Real-Time Access Guarantees for NAND Flash Using Partial Block Cleaning

Siddharth Choudhuri and Tony Givargis

Center for Embedded Computer Systems,
School of Information and Computer Sciences,
University of California, Irvine CA, USA
{sid,givargis}@uci.edu

Abstract. Increasing use of NAND flash in newer application domains has been possible due to lowering cost per GB, consumer demands for storage and advantages of NAND flash over traditional disks. However, NAND flash has its idiosyncrasies resulting in asymmetric read/write times due to garbage collection and wear leveling requirements. Such asymmetric (non-deterministic) read/write times poses a challenge for the adoption of NAND flash in real-time systems.

We present the implementation details of a flash translation layer called *GFTL* that guarantees strict upper bounds on read/write times that are comparable to a theoretical ideal case. Such guarantees are made possible by dividing the source of non-determinism into deterministic intervals using our proposed approach called partial block cleaning. Using partial block cleaning, the process of garbage collection is divided into several smaller, deterministic steps. Partial block cleaning comes with an overhead of additional space requirements. We provide a proof on the limit of the additional space requirements.

Keywords: NAND flash, real-time, file system, embedded systems.

1 Introduction

The use of NAND flash as a storage subsystem is on the rise. NAND flash manifests itself in a wide variety of embedded systems such as mp3 players, digital camera cards, USB based flash drives, set-top boxes, routers to name a few. The driving forces behind the widespread adoption of NAND have been – *(i)* The advantages of NAND flash over hard disk drives such as small form factor, shock resistance and fast access times; *(ii)* The falling cost per GB of NAND flash [1] [2]; and *(iii)* The push from end users for increased storage in consumer electronics. With lowering cost per GB, NAND flash is poised to be used in newer application domains that impose timing guarantees on storage accesses. For example, the One Laptop Per Child (OLPC) project, Canon's HD camcoder use NAND flash as the only non-volatile storage medium [3][4]. While the economics of price has been favorable, the use of NAND flash in mission critical and real-time applications that demand determinism, has been a challenge due to NAND flash idiosyncrasies.

U. Brinkschulte, T. Givargis, and S. Russo (Eds.): SEUS 2008, LNCS 5287, pp. 138–149, 2008.
© IFIP International Federation for Information Processing 2008

NAND flash has certain unique characteristics that are atypical of either RAM or hard disk drives. Specifically, NAND flash does not support in-place updates, i.e., an update (re-write) to a *page* (the minimum of write) is not possible, unless a larger region containing the page (known as a *block*) is first erased. Erase operation on a block is an order of magnitude slower, making it undesirable. Further, a block has a limited erase lifetime (typically 100,000) after which a block becomes unusable. Such characteristics require special handling of NAND flash using either a dedicated file system or wrapping the NAND flash with a layer of hardware/software known as the *flash translation layer* (FTL). The FTL performs three important functions *(i)* Exports a view of NAND flash that resembles a disk drive, thereby hiding the peculiarities of NAND flash. Thus, an FTL translates a read/write request from the file system (*sector*) into a specific ⟨*block, page*⟩ of the NAND flash; *(ii)* Reclaims space by erasing obsoleted blocks (due to out of place updates), also known as *garbage collection*; *(iii)* Performs *wear leveling* to make sure that blocks across a flash get evenly erased.

NAND flash management (wear leveling and garbage collection) is workload dependent resulting in asymmetric read/write times. Therefore, typically FTLs do not provide service guarantees. Such asymmetric read/write latency might be tolerable for single-threaded or dedicated applications. However, as we move towards newer application domains, a deterministic service guarantee becomes desirable to design and run applications.

In this paper, we present implementation details of an FTL called as *GFTL* (for Guarantee Flash Translation Layer) based on the concept of "partial block cleaning". An FTL based on partial block cleaning is capable of providing strict service guarantees for file system accesses (reads/writes) independent of the state or utilization of the flash. Partial block cleaning comes at a cost of additional flash storage. It is our opinion that with rising capacities and lowering cost per GB, additional NAND flash overhead (less than 20% across benchmarks) to provide deterministic guarantee is tolerable. The following are the contributions of this paper:

- GFTL algorithms for read/write access to a NAND flash which provides strict service guarantees due to partial block cleaning.
- Proof for determining the limit on additional space requirements for GFTL.

The rest of the paper is organized as follows: Section 2 briefly describes the NAND flash characteristics. Section 3 presents our problem formulation followed by Section 4 which describes the read/write algorithms for GFTL and presents a proof on the space overhead of GFTL. Section 5 describes the benchmarks used followed by results. Section 7 summarizes related work followed by conclusion.

2 Preliminaries

A NAND flash consists of multiple *erase blocks*. Each such erase block is further divided into multiple *pages*, a page being the minimum unit of data transfer (read/write). Associated with each page is a spare area known as the *Out Of*

Band (OOB) area, primarily meant to store the Error Correction Code (ECC) of the corresponding page (also used to store meta-data such as inverse page table). A page is 512 bytes for older, small block NAND flash and 2 KB for newer large block NAND flash. Three basic operations can be performed on a NAND flash. An *erase* operation "wipes" an entire erase block turning every byte into all 1s i.e., `0xff`. A *write* operation works on either a page or an OOB area, selectively turning desired 1s into 0s. A *read* operation reads an entire page or an OOB area. Updates (re-writes) are out-of-place i.e., directed to a different page unless the entire block is erased. Table 1 depicts NAND flash specifications for the basic operations. There are two possible mappings between a sector and

Table 1. NAND flash specifications

Characteristics	Samsung 16MB Small Block	Samsung 128MB Large Block
Block size	16384 (bytes)	65536 (bytes)
Page size	512 (bytes)	2048 (bytes)
OOB size	16 (bytes)	64 (bytes)
Read Page	36 (usec)	25 (usec)
Read OOB	10 (usec)	25 (usec)
Write Page	200 (usec)	300 (usec)
Write OOB	200 (usec)	300 (usec)
Erase	2000 (usec)	2000 (usec)

a ⟨block, page⟩. A page based mapping where a translation table maps each sector to a ⟨block, page⟩ pair. However, the size of translation table can become a limiting factor as flash size increases. In order to deal with such a problem, a block based translation layer is widely used. For instance, in one of the popular block based translation layers known as NFTL [5], a sector is divided into a virtual block and an offset. The virtual block maps to a physical block (known as the primary block) on the NAND flash. In case of a rewrite (or if the primary block is full), a new physical block called a secondary block is chosen to perform the writes. When the two blocks become full, an operation known as fold merges the primary and the replacement blocks into a new primary block and freeing the old primary and replacement block. Garbage collection is invoked either when the NAND flash runs out of space (which does a fold across several blocks) or using a heuristic. Interested reader can find more details on mapping and garbage collection heuristics in [6] [7]. For the rest of the paper, the term flash refers to NAND flash. Table 2 denotes the terminology used throughout the paper (to be described in later sections)

3 Problem Formulation

We model I/O request (incoming from file system to the FTL) as a real-time task $\tau = \{p, e, d\}$ where p is the periodicity, e is the execution time and d is the

Table 2. Terminology

Symbol	Definition
T_{wrpg}	Time to write a page and OOB area
T_{rdpg}	Time to read a page
T_{rdoob}	Time to read an OOB area
T_{er}	Time to erase a block
π	Pages per block
N	Number of blocks
L	Length of the write pending queue

deadline. Without loss of generality, we assume that p is equal to d. We have two kinds of tasks: a read request task $\tau_r = \{p_r, e_r\}$, and a write request task $\tau_w = \{p_w, e_w\}$. p_r and p_w denote "how often" a read or write request arrives from the file system. e_r is the time taken to search for a given sector, read the corresponding $\langle$*block, page*$\rangle$ of the flash, and return a success/failure to the file system. Similarly, e_w is the time taken to write a sector to a given $\langle$*block, page*$\rangle$. The bounds on p and e are determined by the FTL. Specifically, a *lower bound* on p (denoted by $\mathcal{L}(p)$) determines the maximum request arrival rate that an FTL can handle. The worst case execution time, i.e., an *upper bound* on e (denoted by $\mathcal{U}(e)$), determines the worst case rate at which requests are serviced by the FTL. For a file system, $\mathcal{U}(e)$ represents the *average memory access time* (AMAT) for read/write and $\mathcal{L}(p)$ represents the maximum rate at which requests are issued to the flash.

A flash needs to perform flash management (wear leveling and garbage collection) which involves erasing atleast one or more blocks. T_{er} is the longest *atomic* operation on a flash, i.e., when a block is being erased, the flash is locked and hence non-interruptible. Therefore, T_{er} is the limiting factor which decides the inter-arrival time (periodicity) of requests. Therefore, in an ideal case, $\mathcal{L}(p)$ is at least T_{er}. The latency due to T_{er} could be hidden by having buffers in the RAM. However, while this solution works in an average case, in a worst case scenario (i.e., when every access results in a block erase), one would require an infinitely large buffer in RAM as the arrival rate would exceed the service rate. Table 3 depicts the bounds guaranteed by GFTL (details in the next section).

Table 3. Service guarantee bounds

Bounds	Ideal	GFTL
$\mathcal{U}(e_w)$	T_{wrpg}	T_{wrpg}
$\mathcal{U}(e_r)$	$T_{rdpg} + T_{rdoob}$	$\pi T_{rdoob} + T_{rdpg}$
$\mathcal{L}(p_r)$ $\mathcal{L}(p_w)$	T_{er}	$T_{er}+\max\{\mathcal{U}(e_w), \mathcal{U}(e_r)\}$

GFTL guarantees (Table 3) a worst case execution time for writes that is as good as an ideal case and a worst case execution time for reads that is marginally ($(\pi - 1)T_{rdoob}$) larger than an idea case. Further, GFTL provides service

guarantees for requests that have an inter-arrival time $[\mathcal{L}(p)]$ that is only slightly larger than an ideal case while still performing garbage collection.

4 Technical Approach

GFTL is a block based approach. A sector is treated as a *logical address* and a *logical block* is derived from the most significant bits of the logical address (Figure 1). A *block mapping table* is used to map a logical block to a physical block of flash. For a given flash with N blocks, there is a $1:1$ mapping between the logical blocks and the physical blocks, resulting in N entries in the block mapping table. Further, GFTL requires an additional Q blocks for a *write queue*.

4.1 GFTL Writes

The first write to a given virtual block is written to a free physical block. Due to a $1:1$ mapping, a free physical block is guaranteed to be available. Once a physical block is found, pages are written sequentially starting from page 0. The sector number is written in the OOB area and serves as an inverse page table. After π writes, the physical block becomes full. The full physical block is added to a garbage collection queue called as *GCQ*. Additional writes that map to a full physical block are written to pages in the write queue (shown as dark gray in Figure 1). The write queue serves as a buffer for writes from the time a physical block becomes full until that physical block is garbage collected. A *write queue tail* serves as the index to the next available page in the write queue. There is only one write queue for the entire flash, thus, there exists a *write queue map* which maps the logical address (sector) to a $\langle block, page\rangle$ of the write queue. Algorithm 1 shows that the bounds on taken by write is T_{wrpg}.

4.2 GFTL Reads

A read to a given sector is first searched in the write queue map since it holds the most recent copy. In case of a write queue map miss, the block mapping table is used to determine the physical block corresponding to the sector. The

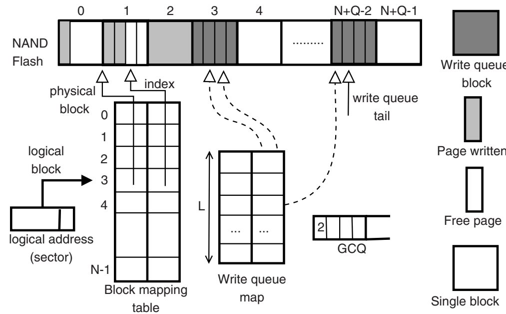

Fig. 1. GFTL Data Structures

Algorithm 1. GFTL write

```
1: writesect(sector, buffer)
2: Input: Function writesect, Sector sect, Buffer buf
3: Output: return status
4: vba ← sector/blocksize
5: if (fsm.state = READ ∨ ERASE) ∧ fsm.blk = vba then
6:     cached ← true
7:     writebuffer(buffer); /* Write to RAM O(1) */
8:     goto PartialGC
9: end if
10: if ¬ cached then
11:     pba ← blockmap[vba].block /* RAM lookup O(1) */
12:     if pba = NULL then
13:         pba = find_free_blk() /* RAM lookup O(1) */
14:         nandwrite(pba, 0, buf) /* Write to flash O(T_wrpg) */
15:         goto PartialGC
16:     end if
17:     if pba.status = BLOCK_FULL then
18:         pba ← writequeue.block
19:         page ← writequeue.tail
20:     else
21:         pba ← blockmap[vba].index
22:     end if
23:     nandwrite(pba, page, buf) /* Write to flash O(T_wrpg) */
24: end if
25: PartialGC:
26: if  GCQ.size > 0 then
27:     do_fsm() /* Invoke partial GC FSM to determine next state */
28: end if
29: return
```

OOB area of the physical block is searched backwards starting from the page pointed to by the index field of the block mapping table.

A read from the write queue will result in one OOB read and one page read. A read from block mapping table on the other hand will result in π OOB reads in the worst case followed by the actual page read. Therefore, the best case AMAT for reads is $T_{rdpg} + T_{rdoob}$ and the worst case is $\pi T_{rdoob} + T_{rdpg}$.

A write either goes to the next available location pointed to by the index field of block mapping table (Figure 1) or into the write queue in case of a full physical block. In either case the time taken is constant i.e., T_{wrpg}. Due to the $1:1$ mapping between virtual and physical blocks, a physical block is guaranteed available for the very first write. Further, in case of a full block, the size of write queue is such that a page is guaranteed to be available.

4.3 GFTL Flash Management

The only flash management performed in GFTL is based on partial block cleaning which takes care of both garbage collection and wear leveling. The idea

Algorithm 2. GFTL read

```
1: readsect(sector, buffer)
2: Input: Function readsect, Sector sect, Buffer buf
3: Output: return status
4: if sector ∈ writequeue then
5:    pba ← writequeue[sector].block
6:    page ← writequeue[sector].page
7: else
8:    pba ← blockmap[vba].block /* RAM lookup O(1) */
9:    for all page ∈ pba do
10:      nand_read_oob(pba, page, oob) /* O(π × T_rdoob) */
11:      if sector = oob.sec then
12:         nand_read_page(pba, page, buf) /* O(T_rdpg) */
13:      end if
14:   end for
15: end if
16: if GCQ.size > 0 then
17:    do_fsm() /* Invoke partial GC FSM to determine next state */
18: end if
```

behind partial block cleaning is to perform garbage collection on a single block at a time. Further, each such single block garbage collection is divided into "partial" steps such that the time taken to perform each step is *no longer* than the longest atomic flash operation i.e., T_{er}. The partial steps are interleaved between servicing read/write requests. The garbage collection of a single block, say B_i, amounts to the following phases:

1. *Block Read:* In this phase, the pages that belong to B_i are first read from the write queue followed by reading the remaining valid pages out of the block B_i. In a worst case, this step can result in reading $(\pi - 1)$ pages from the write queue followed by π OOB reads of B_i to search the remaining valid page. Thus, the worst case time is $(2\pi - 1)T_{rdoob} + \pi T_{rdpg}$.
2. *Block Erase:* Block B_i is erased in time T_{er}.
3. *Block Write:* The pages that were read in phase 1 are written to a free block, say, B_{new}. In a worst case, π pages will be written resulting in a worst case time of πT_{wrpg}.

The idea behind partial block cleaning is to divide the block read and block write phases into partial steps, each of which is of a duration equal to T_{er} as shown in Figure 2(a). Let $\alpha = \lceil (2\pi - 1)T_{rdpg}/T_{er} \rceil$ denote the number of partial steps into which a read phase can be split as multiple of T_{er}. Similarly, $\beta = \lceil \pi T_{wrpg}/T_{er} \rceil$ denotes the number of partial steps that a block write can be broken into. Thus partial block cleaning divides the three block cleaning phases into $(\alpha + 1 + \beta)$ steps, each of a duration T_{er}.

The core of GFTL acts as a real-time executive that implements the finite state machine shown in Figure 2(b). As shown in Figure 2(a), GFTL first dispatches

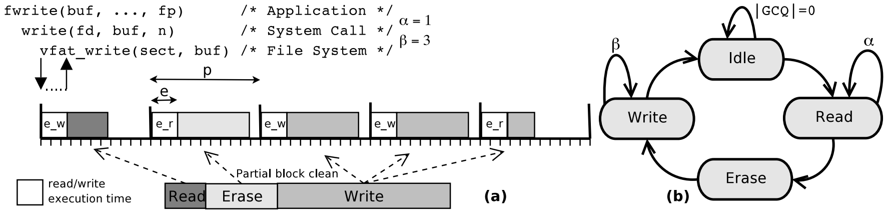

Fig. 2. Partial block cleaning and FSM

any read/write request followed by performing a step of partial block cleaning (if the GCQ is non-empty). This approach lets GFTL provide read/write service guarantees shown in Table 3 while accepting requests at a rate equal to $\mathcal{L}(p)$. The wear level is taken care of in GFTL due to a round robin approach to allocating free blocks.

4.4 Write Queue Limit

In order to determine the write queue limit (i.e., the limit on L), we consider a worst case write request arrival sequence. The following is a worst case write request arrival sequence: $N \times \pi$ write requests arrive such that each request is to a unique page. Thus, at the end of $N \times \pi$ write requests, we have a full flash. Now, each subsequent request will start filling the write queue. Note that if each request filling up a write queue belongs to a unique logical block, garbage collecting such write queue block cannot be started until each block whose page is written to the write queue block has been reclaimed. For example, if a write queue block Q_i has π pending writes that belong to unique logical blocks $\{B_1, B_2, ..., B_\pi\}$, the write queue block Q_i cannot be reclaimed (garbage collected) until each block in $\{B_1, B_2, ..., B_\pi\}$ has been reclaimed. Therefore, the worst case sequence of logical blocks to which writes arrive are $\{0, 1, 2, ..., N-1, 0, 1, 2, ..., N-1, ...\}$ (Figure 3 "Block Numbers Arrival Sequence"). This results in each write queue block being filled with π pending writes, each of which belongs to a unique logical block. Therefore, a write queue block cannot be reclaimed until π blocks are first garbage collected (i.e., worst case for a write queue block). Thus, the write request grows at a rate equal to $1/\mathcal{L}(p)$ (Figure 3 "Arrival Rate"). However, every $(\alpha + \beta + 1) \times \mathcal{L}(p)$ time units, a block is garbage collected (Figure 3 "Service Rate") resulting in a net growth of write queue (Figure 3 "Theoretical Write Queue Length"). In this case the arrival rate $1/\mathcal{L}(p)$ is greater than the service rate $1/(\alpha + \beta + 1)\mathcal{L}(p)$ (Figure 3) leading to an infinite queue length. However, in our worst case arrival model, after N writes, every incoming write request already has at least one other pending write in the write queue that belongs to the same logical block as the incoming write. Similarly, after $2N$ writes, every write request has 2 pending requests that belong to the same logical block. Thus, with time, the growth of the write queue length decreases every N requests reaching a steady state value (Figure 3 "Write Queue Length"). Specifically, the write

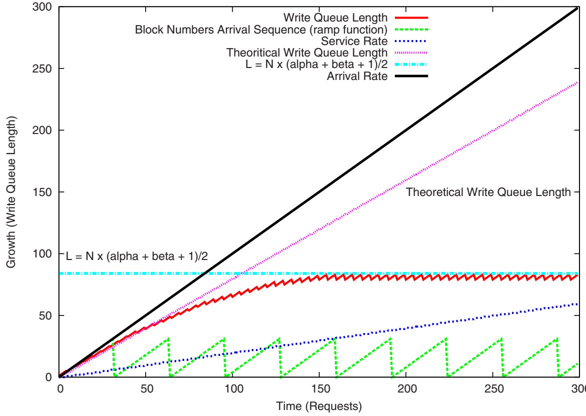

Fig. 3. Write Queue Length Growth

queue length reaches a maximum value of $L = [N \times (\alpha + \beta + 1)/2]$ after which the write queue attains a steady state. Figure 3 depicts the growth of write queue buffer with $\mathcal{L}(p) = 1$.

The following proof provides a limit on the upper bound of the write queue length. The proof is derived for the worst case arrival sequence mentioned above (i.e., the write queue pages fill up such that each page belongs to a different logical block and the distance between two pages in the write queue that belong to the same block is N).

In Figure 3, the ramp function denotes the growth of write queue in terms of the logical block numbers. The actual growth is denoted by the curve entitled "Write Queue Length". Assuming $\kappa = (\alpha + \beta + 1)$, the service rate is given by $y(x) = x/\kappa$.

Every κ interval, a physical block B_i is reclaimed. Since the block B_i is reclaimed, every page $p \mid p \in writequeue \wedge physical_block(p) = B_i$ is also rendered obsolete. Every N^{th} interval, the number of such write queue pages $\mid p \mid$ (that are rendered obsolete) increases by 1 until κ times. This can be seen as the intersection of "Service Rate" and the ramp function in Figure 3. After κ times, the growth of the write queue reaches a steady state as the number of pages that are rendered obsolete i.e., $\mid p \mid$ equals κ. Therefore, the write queue length reaches a steady state where it grows by an amount κ and then decreases by the same amount every κ intervals due to multiple pages in the write queue being rendered obsolete.

Thus, the upper bound on the length of the write queue can be obtained by summing the growth of write queue (given the arrival rate) and the decrease in write queue due to partial garbage collection. The write queue increases monotonically in the worst case. The decrease due to block cleaning is given by intersection of the service rate with the ramp function. The first intersection is found at $y = x/\kappa$ for $x = N$. The second intersection is found at $y = 2x/\kappa$ for

$x = 2N$ and so on. The summation until the steady state gives the worst case bound on the write queue length L

$$
\begin{array}{lll}
\text{End of } 1^{st} \text{ interval} & L_1 & = N - \lfloor N/\kappa \rfloor \\
\text{End of } 2^{nd} \text{ interval} & L_2 & = N - \lfloor 2N/\kappa \rfloor \\
\ldots & & \\
\text{End of } \kappa - 1^{th} \text{ interval} & L_{\kappa-1} & = N - \lfloor (\kappa - 1)N/\kappa \rfloor
\end{array}
$$

$$
\begin{array}{lll}
\text{Summing,} & \Sigma_i^{\kappa-1} L_i & = (N \times (\kappa - 1)) - (N \times (\kappa - 1)/2) \\
& \Sigma L & = N \times (\kappa - 1)/2
\end{array}
$$

To this summation, we add N additional entries to accommodate the floor function rounding off as a buffer. Thus, the upper bounds on write queue limit is

$$
\begin{aligned}
L &= N \times (\kappa - 1)/2 + N \\
&= \boldsymbol{N(\kappa + 1)/2}
\end{aligned}
$$

Note that though L is greater than N (total blocks), the actual write queue length in terms of the number of additional blocks is $[N(\kappa+1)/2]/\pi$ as each block can store π pending writes. Thus, for a given flash the write queue length (L), can be calculated at design time by looking at the flash specs and independent of workload or flash state.

5 Results

We used the following benchmarks representing a variety of workloads. The *Andrew* benchmark [8] consists of five phases involving creating files, copying files, searching files, reading every byte of a file and compiling source files. The *Postmark* benchmark measures performance of file systems running networked applications like e-mail, news server and e-commerce [9]. The *iozone* benchmark [10] is a well known synthetic benchmark. We ran iozone to do read, write, rewrite, reread, random read, random write, backward read, record rewrite and stride read. The file sizes ranged from 64KB to 32MB in strides of 2× (i.e., 64, 128, ... 32768). Besides these standard benchmarks, we used our own benchmark called *consumer* which simulates flash activities used in consumer electronics devices such as image manipulation, data transfer, audio and video playback.

A set of benchmarks were run in sequence to generate a file system *trace*. The first trace, called the *synthetic* trace was generated by running the following sequence: format flash → andrew → postmark → iozone. Similarly, consumer trace was generated by formating a flash followed by running the consumer benchmark. In order to perform a rigorous evaluation of GFTL, each read/write in the trace was simulated with a periodicity of $\mathcal{L}(p)$ i.e., there is *no idle period.* Further, the synthetic trace consists of 4.3 million writes and 27,841 reads and

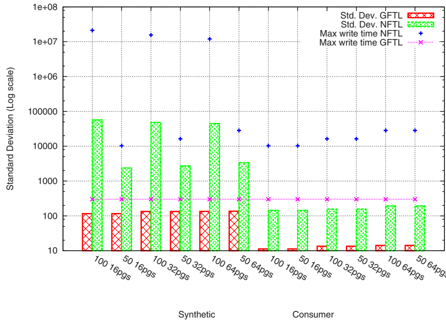

Fig. 4. NFTL vs. GFTL writes

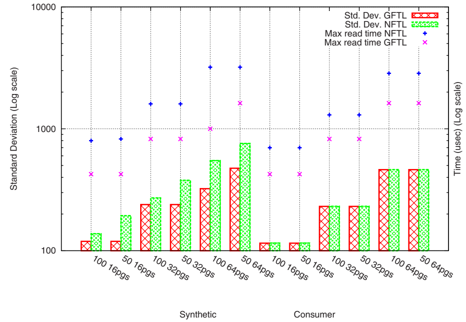

Fig. 5. NFTL vs. GFTL reads

the consumer trace consists of 125, 596 writes and 76, 479 reads. The flash size at 100% utilization for synthetic trace is 136 and 260 MB for the consumer trace.

Figures 4 and 5 compares GFTL and NFTL in terms of read/write performance. The variation in write times is more than an order of magnitude less for GFTL due to partial block cleaning. The maximum write time of GFTL is constant as opposed to NFTL. The maximum read time is proportional to the number of pages per block i.e., π. This is due to the fact that reads requires a sequential read of the OOB area until a desired sector is found. The average overhead calculated across the traces and across all page, block size combinations is 16%.

6 Related Work

While there have been several block based FTLs, the real time aspect of NAND flash was first investigated by [11]. The authors proposed an innovative approach towards using a garbage collector thread (instance) for each real time task. The garbage collector thread has a execution time of $(\pi - \alpha) \times (T_{rdpg} + T_{wrpg}) + T_{er}$+cpu_time). In [11] each garbage collector invocation is takes at least $(\pi - 1)(T_{rdpg} + T_{wrpg}) + T_{er})$ time (ignoring cpu time) in the *best case*. In our approach, the overhead of partial GC is T_{er} in the *worst case*. Moreover, with GFTL we do not associate an additional GC task thereby avoiding overhead. [11] requires file system support for special *ioctl* calls. GFTL can be run on top any unmodified file system. Results from [11] are based on two tasks $T1 = (3, 20)$ and $T2 = (5, 20)$ resulting in creation of two GC tasks $G1 = (22, 160)$ and $G2 = (22, 600)$ at 50% utilization. The execution time of GC thread is comparable to 10 times T_{er}. GFTL on the other hand provides a delay that is around T_{er}. Moreover, we provide a rigorous where each request is considered a real-time task along with high utilization.

In [12], the authors address soft real-time issues by modifying the file system. The techniques in [12] focus on commonly used access patterns and not strict guarantees. In [6], the authors survey a wide range of garbage collection

algorithms as part of their study. However, the garbage collectors are not aimed at real-time systems. An exhaustive research on flash memories for real time systems was done by [13]. The conclusions in [13], supports our motivation for the lack of real-time, deterministic guarantees for flash. The results on wear level and details on benchmark performance is in[14].

7 Conclusion

In this paper we provided the algorithms to implement an FTL called GFTL that guarantees $O(1)$ write time and a read time that takes π (pages per block) searches of the flash OOB in the worst case. Further, we provided a proof that determines the bounds on space overhead required by GFTL using partial block cleaning. Thus, for a given flash the write queue can be computed at design time independent of flash workload or state. Using the approach of partial block cleaning, real-time guarantees can be provided for NAND flash (that are close to an ideal case). The overhead of partial block cleaning is less than 20% across the benchmarks used in our experiments.

References

1. Lawton, G.: Improved flash memory grows in popularity. Computer 39(1), 16–18 (2006)
2. MemCon: MemCon. (July 2007), `http://linuxdevices.com/news/NS6633183518.html`
3. One Laptop Per Child Project, `http://laptop.org`
4. Canon: Vixia HD Camcoder (January 2008)
5. Ban, A.: Flash file system optimized for page-mode flash technologies. US Patent 5,937,425 (August 10, 1999)
6. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Comp. Surv. 37(2), 138–163 (2005)
7. Chang, L.P., Kuo, T.W.: Efficient management for large-scale flash-memory storage systems with resource conservation. Trans. Storage 1(4), 381–418 (2005)
8. Howard, J.H., et al.: Scale and performance in a distributed file system. ACM Trans. Comput. Syst. 6(1), 51–81 (1988)
9. Katcher, J.: Postmark: A new file system benchmark. Technical report, Net App. Inc. (TR 3022) (1997)
10. Norcutt, W.: IOZONE benchmark, `http://www.iozone.org`
11. Chang, L.P., Kuo, T.W., Lo, S.W.: Real-time garbage collection for flash-memory storage systems of real-time embedded systems. TECS 3(4), 837–863 (2004)
12. New techniques for real-time fat file system in mobile multimedia devices. IEEE Transactions on Consumer Electronics 52, 1–9 (2006)
13. Parthey, D.: Analyzing real-time behavior of flash memories. Diploma Thesis, Chemnitz University of Technology (April 2007)
14. Choudhuri, S., Givargis, T.: Deterministic service guarantees for NAND flash using partial block cleaning. In: CODES+ISSS 2008. ACM, New York (to appear, 2008)