Deterministic Service Guarantees for NAND Flash using Partial Block Cleaning

Siddharth Choudhuri and Tony Givargis
University of California, Irvine, USA
Email: {sid, givargis}@uci.edu

***Abstract*— Current generation embedded systems are capable of running applications that were realm of desktop systems a few years ago. Along with sophisticated processors, affordable storage due to NAND flash continues to be one of the enabling technologies driving the proliferation of embedded systems. NAND flash has its idiosyncrasies (eg: bulk erase, wearleveling) which results in a non-linear and unpredictable read/write access times. In case of application domains such as streaming multimedia and real-time systems, a deterministic read/write access time is desired during design time.**

We propose a novel NAND flash translation layer called *GFTL* that guarantees fixed upper bounds (worst case service rates) for reads and writes that are comparable to a theoretical ideal case. Such guarantees are made possible by eliminating sources of non-determinism in GFTL design and using partial block cleaning. GFTL performs garbage collection in partial steps by dividing the garbage collection of a single block into several chunks, thereby interleaving and hiding the garbage collection latency while servicing requests. Further, GFTL guarantees are independent of flash utilization, size or state. Along with theoretical bounds, benchmark results show the efficacy of our approach. Based on experiments, GFTL requires an additional 16% of total blocks for flash management. A proof for additional blocks required is provided for a general case. GFTL service guarantees can be calculated from flash specifications. Thus, with GFTL a designer can determine the service guarantees and size requirements apriori, during design time.

Index Terms NAND flash, Embedded Systems, Storage, QoS, Determinism, Real-Time, File Systems

I. Introduction

The proliferation of embedded systems has led to wide spread use of NAND flash as a storage medium. [1]. While the use of flash memory for secondary storage in mobile embedded systems has been known for over a decade [9], large scale adoption has only been possible recently due to affordable cost. With lowering cost per GB, NAND flash is poised to be used in newer application domains [13][14]. For example, the One Laptop Per Child (OLPC) project, Canon's HD camcoder use NAND flash as the only non-volatile storage medium[16][5]. 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.

[1]There are two kinds of flash memories − NOR flash and NAND flash. NOR flash is mostly used for small amounts of code storage. NAND flash is widely used as a data storage

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 by using either a dedicated file system or wrapping the NAND flash with a layer of hardware/software known as the *flash translation layer* or FTL (Figure 1). 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 $\langle block, page \rangle$ of the NAND flash; *(ii)* Reclaims space by erasing obsoleted blocks (due to out of place updates), also known as *garbage collection*; *(iii)* Performs *wearleveling* to make sure that blocks across a flash get evenly erased.

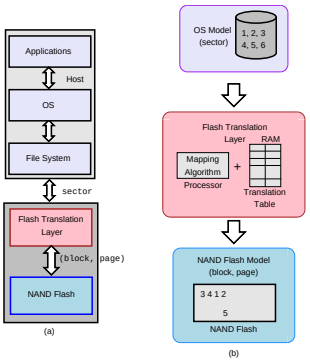

Figure 1. (a) NAND FTL converts OS view of sectors to NAND (block, page), (b) FTL consisting of a low-end processor and RAM

NAND flash management (wearleveling, garbage collection) is workload dependent resulting in asymmetric read/write times. Therefore, typically FTLs do not provide service guarantees. For instance, consider a scenario in which an FTL is busy performing garbage collection over several blocks. During this time period I/O requests experience a high latency. Such latency may be tolerable for single-threaded applications. However, as we move towards newer application domains, a deterministic service guarantee becomes desirable to run applications.

In this paper, we propose a NAND flash translation

doi:10.4304/jsw.4.7.728-737

layer called as *GFTL* (for Guarantee Flash Translation Layer) that provides strict service guarantees for reads and writes that are close to an ideal case (to be described in Section 3). GFTL achieves this using a two fold approach. First, it uses a mapping from sectors to pages on flash that eliminates any dependency on flash utilization or state (i.e., provides determinism). Second, it uses partial block cleaning to hide the flash management latencies. Partial garbage collection is a scheme where the basic unit of garbage collection is a single block. Further, the garbage collection of each such block is divided into smaller states such that the garbage collection and the file system read/write requests are *interleaved*, resulting in a responsive systems that hides garbage collection latency. The following are contributions of this paper −

(i) An FTL that provides strict service time guarantees for reads and writes independent of the workload, utilization or the state of NAND flash. The FTL is validated based on file system benchmarks and substantiated with proof of deterministic guarantee for a general case.

(ii) Partial garbage collection, where the garbage collection of a single block is divided into chunks that are no greater than the largest non-interruptible flash operation; thereby providing a responsiveness that is close to a theoretical limit.

II. 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 I depicts NAND flash specifications for the basic operations.

TABLE I.
NAND FLASH DATA SHEET 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)

The above properties of NAND flash result in out-of-place updates and garbage collection. A page P starts off in a *free* (erased) state. Once data is written into page P, its state changes to a *valid* state. However, an update (re-write) to page P is not possible. In order to overcome this limitation, an update is made out-of-place i.e., to another page Q that is in free state. Following this, the state of page P changes from valid $\rightarrow$ obsolete and the state of page Q changes from free $\rightarrow$ valid.

Garbage collection is the process of reclaiming space by erasing the blocks that contain obsolete pages. Note that not all pages in a block might be obsolete, hence garbage collection takes care of moving valid pages into a different block before erasing the whole block. Due to the out-of-place updates and garbage collection, it is not possible to have a fixed association between a sector and a $\langle$*block, page*$\rangle$.

There are two possible mappings between a sector and a $\langle$*block, page*$\rangle$. A page based mapping where a translation table maps each sector to a $\langle$*block, page*$\rangle$ 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 [3], 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 [10] [6]. For the rest of the paper, the term flash refers to NAND flash. Table II denotes the terminology used throughout the paper (to be described in later sections)

TABLE II.
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

III. 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 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.

We now present a hypothetical ideal case that serves as a baseline for comparison. In an ideal case, the read/write access takes constant time. The bounds on $\mathcal{U}(e)$ for such an ideal case is shown in Table III, i.e., there are no additional flash management overheads other than the actual page read/write [2].

A flash needs to perform flash management (wearleveling, garbage collection) which involves erasing atleast one or more blocks. Note that, 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 that 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. This leads us to the following axiom:

"*In the presence of flash management in a single chip flash, the block erase time T_{er} provides the lower bound on inter-arrival request time*".

TABLE III.
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$\{\mathcal{U}(e_w), \mathcal{U}(e_r)\}$

Although non-realtime block based FTLs like NFTL provide a write time close to T_{wrpg} in an average case, flash management results in a drastic, unpredictable variation. Figure 2 depicts this scenario for a set of synthetic benchmarks. Note there are two distinct variations on the y-axis due to folds and garbage collection.

The motivation behind GFTL is to reduce this variation thereby enabling flash to be used in real-time applications. GFTL guarantees (Table III) 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 ideal 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 performing garbage collection. Next section explains the technical details behind providing guarantees in Table III.

[2]For simplicity, we do not include the flash controller processor execution time as it is at least an order of magnitude lesser than a typical flash access time

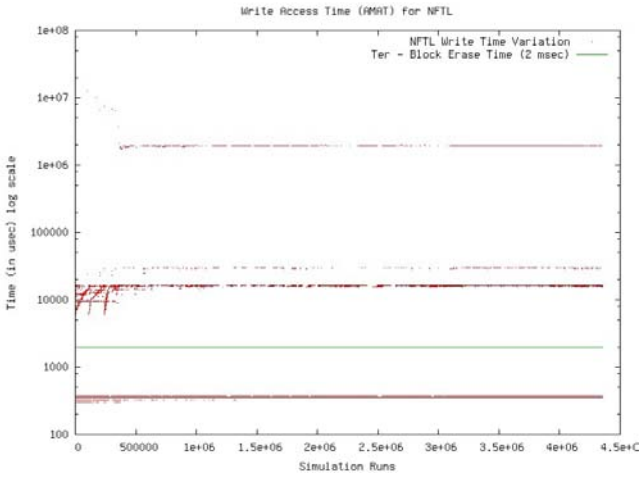

Figure 2. Write time variation for NFTL

IV. 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 3). A *block mapping table* is used to map a logical block to a physical block on the 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*.

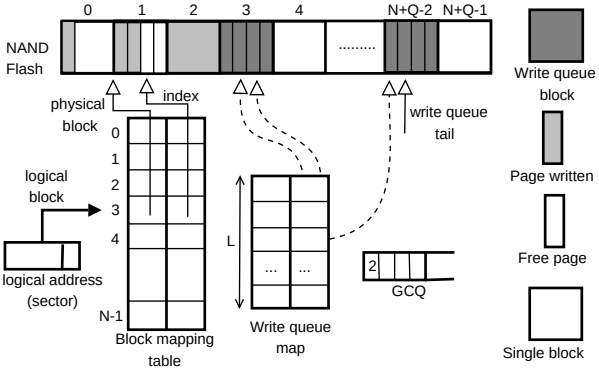

Figure 3. GFTL Data Structures

A. 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. Note that the advantage of writing pages sequentially is that GFTL can be adopted to the newer MLC NAND flash which imposes a restriction on random page writes within a block. 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 3). 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. Write queue blocks that do not have any valid pages (i.e., no logical address in write queue map points to a given write queue block) are added to the GCQ to be erased in future. A write queue block can become obsolete if its pages either got rewritten to a different block in the write queue or the block to which the pages belonged to have been garbage collected.

A write either goes to the next available location pointed to by the index field of block mapping table (Figure 3) 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. (to be shown in subsection "Write Queue Limit"). Thus, both the best and worst cast AMAT for writes is T_{wrpg}.

Algorithm 1 depicts the write operation of GFTL. The algorithm takes as input the sector number and a pointer to a buffer to be written. Lines $5-9$ updates the in memory buffer if the sector to be written belongs to a block that is being garbage collected. Lines $10-24$ depict the more general case. The `nandwrite` function is a low level function that writes to the flash. Lines $25-28$ depict the partial garbage collection process. The `do_fsm` invokes the next state (or step) of partial garbage collection process. This process is described in more details in the GFTL flash management subsection.

Algorithm 1 GFTL write

```
 1: writesect (sector, buffer)
 2: Input: 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 flash O(T_wrpg)
24: end if
25: PARTIALGC:
26: if GCQ.size > 0 then
27:     do_fsm() // Invoke partial GC FSM for next state
28: end if
29: return status
```

Algorithm 2 GFTL read

```
 1: readsect (sector, buffer)
 2: Input: 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 for next state
18: end if
```

B. 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 OOB area of the physical block is searched backwards starting from the page pointed to by the index field of the block mapping table. The search is done backwards to determine the most recently written copy of the sector.

The page corresponding to a read is found either in the write queue map or in the physical block pointed to by 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 $\pi T_{rdoob} + T_{rdpg}$ (Table III).

Algorithm 2 depicts the read operation of GFTL. The algorithm takes as input a sector number and an empty buffer to be read. Lines $4-6$ depict the case when the sector to be read belongs to a block that is currently being garbage collected. Lines $7-15$ depict the case when a sector is searched sequentially in the flash depending on the block map. Lines $16-18$ invoke the `do_fsm` function which is same as in the write algorithm.

Algorithm 3 depicts the `do_fsm` function. This algorithm does the state transition between read, write and erase such that the duration of each call is equal to time T_{er}. The current page that was left off (during read/write) at each call is stored in global memory to restart at the following page at next invocation of the `do_fsm` call. The details of saving and restoring the current page pointer are implementation specific and left out for brevity and keeping the algorithm generic.

C. 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 behind partial block cleaning is to perform garbage collection on

Algorithm 3 GFTL FSM

```
1: do_fsm (sector, buffer)
2: Input: Sector sect, Buffer buf
3: Output: return status
4: if GCQ.size = 0 then
5:     state ← idle
6:     return state // Remain in idle state
7: end if
8: if state = read then
9:     for all pg = current, pg ∈ [current + range] do
10:        nandread(blk, pg, buf) // Read for T_er time
11:    end for
12:    if current = end of page then
13:        state ← write // Switch next state after π reads
14:    end if
15:    return state
16: end if
17: if state = write then
18:    for all pg = current, pg ∈ [current + range] do
19:        nandwrite(blk, pg, buf) // Write for T_er time
20:    end for
21:    if current = end of page then
22:        state ← erase // Switch next state after π writes
23:    end if
24:    return state
25: end if
26: if state = erase then
27:    nanderase(blk)
28: end if
29: state ← idle
30: return state
```

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:

(i) *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 $(2\pi - 1)T_{rdoob} + \pi T_{rdpg}$.

(ii) *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 in a worst case time of πT_{wrpg}.

(iii) *Block Erase:* Block B_i is erased in time T_{er}.

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 4(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 equal to the erase time i.e., T_{er}.

The core of GFTL acts as a real-time executive that implements the finite state machine shown in Figure 4(b). As shown in Figure 4(a), GFTL first dispatches 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 III while accepting requests at a rate equal to $\mathcal{L}(p)$. The order of performing partial garbage collection is read followed by write followed by erase. This order (Figure 4(b)) ensures that data integrity in case of abrupt power failure i.e., data is written first before doing an erase.

The wearlevel is taken care of GFTL due to a round robin approach to allocating free blocks. This approach reduces data structure overhead in selecting a free block. However, more sophisticated approaches such as moving blocks based on age can be implemented (by inserting blocks into GCQ) to further improve the wearlevel. Also, wear level can be implemented independent of GFTL using an approach similar to Ubifs i.e., a layer that acts as a volume manager for the flash and hides the complexities of wear level and bad block management from the FTL [4]. There exists substantial research in this area, hence we chose not to delve into it.

Over a period of time, blocks that belong to the write queue need to be erased. This is due to the fact that every page that belongs to a write queue block has been garbage collected. GFTL determines such blocks by scanning the write queue map. If a write queue block (other than the write queue block which is currently being written to) has no pointers pointing to it from the write queue map, the block is added to GCQ. The cost of garbage collecting a write queue block is only T_{er}.

D. 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 6 "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 6 "Arrival Rate"). However, every $(\alpha + \beta + 1) \times \mathcal{L}(p)$ time units, a block is garbage collected (Figure 6 "Service Rate") resulting in a net growth of write queue (Figure 6 "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 6) leading to an infinite queue length. However, in our

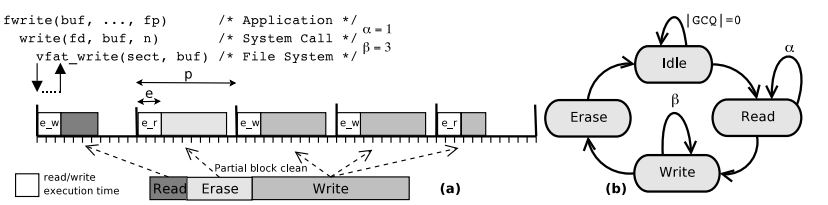

Figure 4. Partial Block Cleaning and FSM

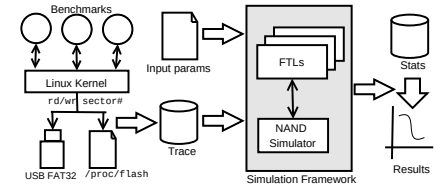

Figure 5. Setup

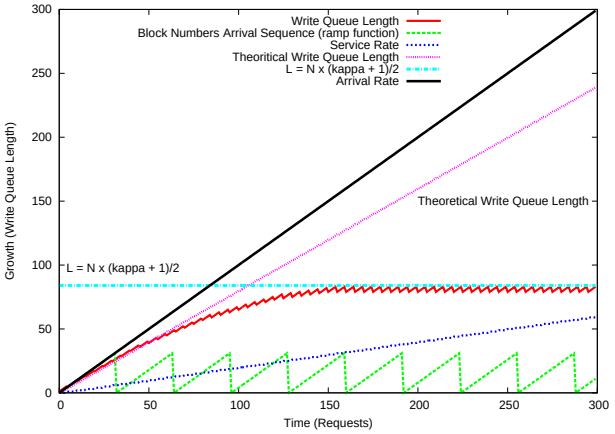

Figure 6. Write Queue Length Growth

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 6 "Write Queue Length"). Specifically, the write queue length reaches a maximum value of $L = [N \times (\alpha + \beta + 1)/2]$ after which the write queue attains a steady state. Figure 6 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 6, 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 6. 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:

End of 1^{st} interval $\quad L_1 = N - \lfloor N/\kappa \rfloor$

End of 2^{nd} interval $\quad L_2 = N - \lfloor 2N/\kappa \rfloor$

...

End of $\kappa - 1^{th}$ interval $\quad L_{\kappa-1} = N - \lfloor (\kappa - 1)N/\kappa \rfloor$

Summing, $\quad \Sigma_i^{\kappa-1} L_i = (N \times (\kappa - 1)) - (N \times (\kappa - 1)/2)$

$$\Sigma L = N \times (\kappa - 1)/2$$

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

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

Although 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 inspecting the flash specs and independent of workload or flash state.

V. Experimental Setup

Figure 5 shows our experimental setup. A USB flash disk, formatted as a FAT 32 file system was connected to a PC running Linux kernel 2.6.16. The kernel was modified to sniff low level file read/write requests being issued to the USB flash and log the requests (sector, read/write operation) into `/proc/flash`. A series of benchmarks were run to generate trace data. The trace, along with input parameters (block size, page size, etc) is fed to our simulation framework.

We used the following benchmarks representing a variety of workloads. The *Andrew* benchmark [11] 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 [12]. The *iozone* benchmark [15] 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 on file sizes ranged from 64KB to 32MB in strides of $2\times$. Besides these standard benchmarks, we used our own benchmark called *consumer*. The consumer benchmark simulates flash activities commonly 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 $\rightarrow$ andrew $\rightarrow$ postmark $\rightarrow$ 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 the consumer trace consists of $125,596$ writes and $76,479$ reads. The flash size at 100% utilization for synthetic trace is 136 MB and 260 MB for the consumer trace. The simulations are based on values for large page flash (Table I).

TABLE IV.
BENCHMARK CHARACTERISTICS

Benchmark	Reads	Writes	Sect Range
Format	15	36	0 - 533
Andrew	2	3126	1 - 2867
Postmark	412	21153	2 - 10000
Postmark long	23694	1238135	1 - 65543
IOzone	3713	3089393	1 - 65588
Consumer	76749	125596	1 - 125060

VI. RESULTS

Table V depicts a summary of runs for the two traces based on varying utilization and pages per block (π). The first two columns under $\bar{e}_{wr}$ and $\bar{e}_{rd}$ denote the average write and read access times (AMAT) for each run of the trace. Similarly, e^{max}_{wr} and e^{min}_{rd} denote the maximum and minimum recorded AMAT. The standard deviation is denoted by σ^{e}_{wr} for writes and σ^{e}_{rd} for reads. The effectiveness of wearleveling is measured using *wear index* denoted by w_i. wear index is a quantitative measure of wearlevel calculated as $(\sigma_{erase}/N)\times 100$, where σ_{erase} is the standard deviation of the number of erases per block. σ_{erase} takes into account the "variation" in number of times a block is erased. To take into account the size of flash in determining wearlevel, we used σ_{erase}/N as an indicator of wearlevel.Note that a flash with larger number of blocks has better wearlevel than a flash with smaller number of blocks for a fixed σ_{erase}. Therefore, we chose σ_{erase}/N as an indicator for wearlevel.

GFTL incurs overhead due to the write queue. This overhead is measured as the percentage increase in flash size denoted by column entitled Δ in Table V. The following observations are made based on Table V: *(i)* As mentioned in Table III, the maximum write time is equal to T_{wrpg}. The average write time is less than T_{wrpg}

TABLE V.
GFTL PERFORMANCE

	%	π	$\bar{e}_{wr}$ usec	e^{max}_{wr} usec	σ^{e}_{wr}	$\bar{e}_{rd}$ usec	e^{max}_{rd} usec	σ^{e}_{rd}	w_i	Δ %
Synthetic	50	16	244	300	116	231	425	119	0.43	16
		32	217	300	133	428	825	239	1.11	14
		64	212	300	136	789	1625	474	3.05	14
	100	16	244	300	116	231	425	119	3.29	16
		32	217	300	133	428	825	239	8.17	14
		64	212	300	136	789	1625	474	14.2	14
Consumer	50	16	299	300	11	237	115	425	0.00	16
		32	299	300	13	237	425	115	0.01	13
		64	299	300	14	836	1625	462	0.01	12
	100	16	299	300	11	237	425	115	0.01	16
		32	299	300	13	437	825	231	0.02	13
		64	299	300	14	836	1625	462	0.02	12

because some of the writes occur during the FSM state change resulting in being written to a block buffer (written later on to the flash and accounted in partial garbage collection time). The maximum read time depends on pages per block for a given flash. This is due to the fact that larger π implies longer chain of OOBs to read. The maximum values observed are equal to $\pi T_{rdoob} + T_{rdpg}$ which is equivalent to searching the entire block for a given page. The standard deviation for reads also shows a similar trend i.e., increasing with larger π *(ii)* The average read/write service times ($\bar{e_{rd}}$/$\bar{e_{wr}}$) are independent of the flash utilization. This result departs from conventional approaches such as NFTL where the AMAT varies as the flash utilization increases. The standard deviation is higher in case of the synthetic trace because of the rewrites and rereads made to the flash by the iozone benchmark. Note that if a block is in the middle of partial garbage collection, additional reads and writes are serviced by the in memory block buffer resulting in an almost zero service time. This leads to the high variation in the average values. *(iii)* The wearlevel index depends on the flash utilization and the number of pages per block, π. As the flash utilization increases, blocks get recycled more often. However, blocks that are read-only are not erased leading to the large gap between minimum and maximum values. This shows as an increase in the wear index. *(iv)* The value of Δ reflects the additional blocks used by GFTL to maintain the write queue. This value was calculated (and used in simulation) based on the equation $L = N \times (\kappa + 1)/2$. Given the lowering cost per GB of NAND flash, such an overhead is tolerable.

Figure 7 shows the distribution of execution time and partial GC in a given period. The total length of a histogram represents a single period i.e., $\mathcal{L}(p)$. The "Rd/Wr request" bar denotes the average service time i.e., $\bar{e}$. The remaining time is spent doing either partial GC or being idle (i.e., when GC queue is empty). Though GFTL guarantees an arrival rate equal to the length of the histogram, a fraction of partial GC time is spent idle because the guarantees are calculated based on worst case scenario. For the given traces, the sum of service time and partial GC is less than the T_{er}. This idle time decreases with increasing π as the value of κ increases which implies that the number of states in the FSM (Figure 4) also increases leading to longer time spent

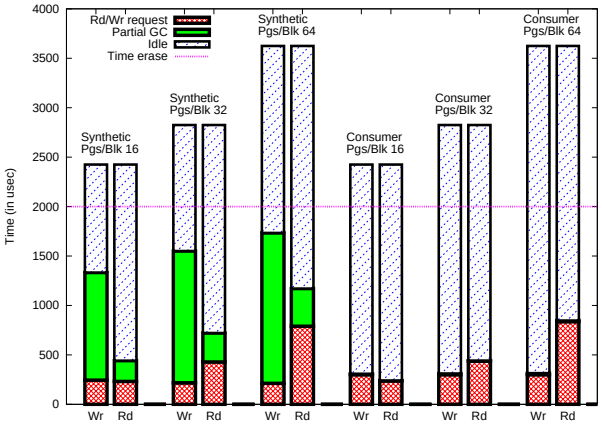

Figure 7. Distribution of time spent in each period

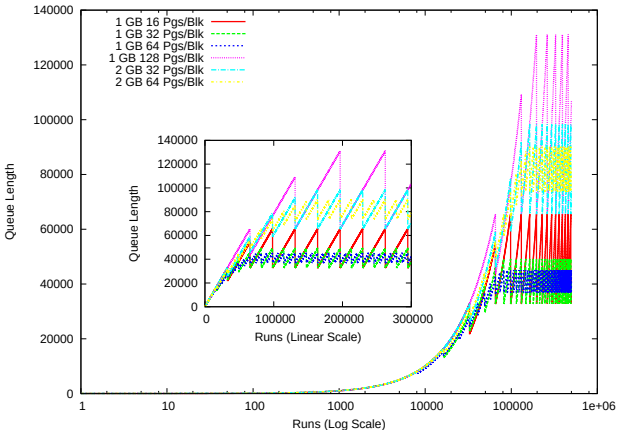

Figure 8. Write Queue Length Growth Simulation

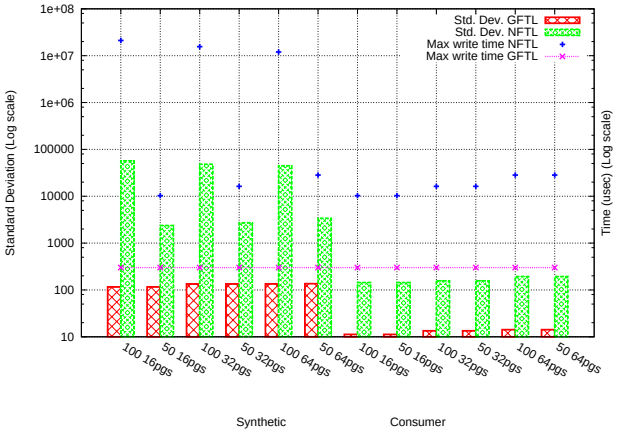

Figure 9. NFTL vs. GFTL writes

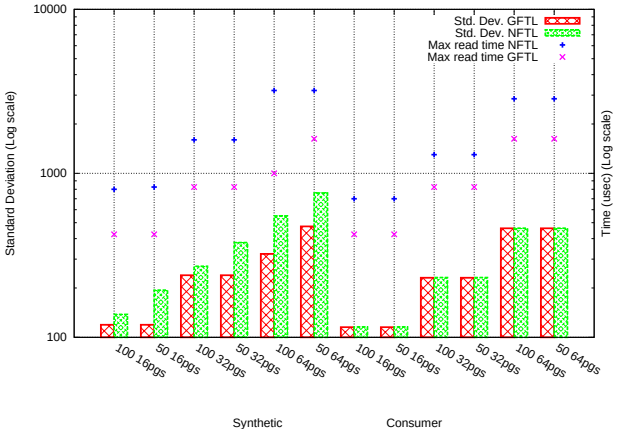

Figure 10. NFTL vs. GFTL reads

in GC. Note that in case of the consumer benchmark, the amount of time spent in performing partial GC is negligible compared to $\mathcal{L}(p)$. Though the consumer trace represents a larger flash size the number of read/writes performed is less than the synthetic trace. The efficacy of GFTL is shown by keeping the flash busy for time that is close to the largest non-interruptible time, T_{er} (Figure 7) while still giving applications a service time that is close to ideal and independent of the flash utilization (Table V).

Figure 8 analyzes the overhead of GFTL in terms of write queue length. The x-axis is the number of runs simulated and the y-axis shows the growth of write queue length L. Initially, the write queue grows at a rate that is close to the incoming request rate and after every N requests, the slope decreases to a final "steady state". During this state, the write queue length varies depending on the value of N and π. Figure 8 shows the growth for a specific large page flash from Table I. The growth depends on the values of T_{er}, T_{wrpg} and T_{rdpg}. Specifically, the larger the difference between the block erase time and the block read/write time, the larger the value of L as GFTL would require more states due to large α and β depending on how many "chunks" of block erase time can the reads and writes be divided into.

Figures VI and VI 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%.

Table VI depicts the representative read/write energy characteristics of GFTL. The columns in Table VI depict varying pages per block. The rows depict the read and write energy per access averaged over the entire simulation runs.

TABLE VI.
AVERAGE ENERGY PER ACCESS

Benchmark	Synthetic ($\pi = 32$)	Synthetic ($\pi = 64$)	Consumer ($\pi = 32$)	Consumer ($\pi = 64$)
Energy Read	14565	35363	7952	10289
Energy Write	43993	48090	10195	14612

The numbers here depict the energy consumption for Samsung large block flash (Table I). With $V_{cc} = 3.3V$ and $I = 10mA$, we have $-$ energy consumption of read page, $E_{rd} = 825nJ$; energy consumption of write page, $E_{wr} = 9900nJ$; and energy consumption of erase page, $E_{er} = 66000nJ$. In case of GFTL, the increase in going from a smaller π to a larger π is significant for reads due to the energy consumed in searching the OOB areas. Also, there is a wide disparity between the energy consumption in case of synthetic trace versus consumer trace. The

synthetic trace stress the flash in terms of heavy rewrites resulting in larger energy consumption. The consumer trace represents a more realistic situation.

In case of GFTL, the energy consumption is independent of the utilization i.e., in case of both 50% and 100% flash utilization, the energy consumption remains same. This is due to the fact that GFTL adopts a "proactive" garbage collection approach versus a "reactive" approach in case of NFTL and other traditional flash translation layer. This results in GFTL energy consumption being higher than traditional FTLs for under utilized flash memories. One approach to reduce the energy consumption in GFTL is to have adaptive garbage collection depending on flash utilization.

Since embedded systems are used as devices, they are expected to have a short start up time and do not go through a shutdown process that is typical of desktop systems. GFTL initialization affects the startup time. Specifically, in case of GFTL, three data structures need to be initialized $-$ (i) block mapping table, (ii)write queue map, and (iii) GCQ. The block mapping table can be built by scanning the OOB area of each block. In order to provide a fast startup time, the write queue map and the GCQ can be written to a specific location in the flash before shut down. This would lead to a startup time equal to $\lceil (N \times 8)/pagesize \rceil \times T_{rdpg} + \lceil (L \times 16)/pagesize \rceil \times T_{rdpg} + \pi \times T_{rdpg}$ and a shutdown time equal to $\lceil (N \times 8)/pagesize \rceil \times T_{wrpg} + \lceil (L \times 16)/pagesize \rceil \times T_{wrpg} + \pi \times T_{wrpg}$. Note that the three summations in the startup times are for the time to write the block mapping queue where each entry is 8 bytes; time to write the write queue map where each map entry is 16 byte; and the time to write the GCQ contents. In the case of an abrupt (improper) shutdown, the entire OOB area needs to be scanned during startup resulting in a longer startup time. The worst case startup time in such a case is given by $N \times T_{rdoob} + L \times T_{rdoob}$. The first part of the summation scans the first OOB area of each block in the flash to determine the inverse page mapping; the second part of the summation scans all OOB area of the L blocks to reconstruct the write queue map. The GCQ can be built while scanning for the block mapping table.

For example, in case of 1GB flash with 2K page having flash characteristics from Table I, the following are the startup and shutdown times $-$ normal startup would take 13.6 msec to build the in memory data structures, shutdown would take 163 msec. In case of an abrupt power down, the startup would take 1.84 sec since the block mapping table and write queue map would be reconstructed by scanning the entire flash.

VII. GFTL for Soft Real-time Workloads

GFTL incurs additional overhead in terms of space overhead and energy consumption due to the "reactive" approach in order to meet the strict timing requirements. In this section, we discuss two independent strategies that can be adopted for non-real time (soft real-time tasks). Such a case is useful for media applications where occasional deadline miss is not considered catastrophic.

A. Threshold based Partial Block Cleaning

GFTL performs an aggressive (proactive) block cleaning independent of the flash utilization. In case of a flash that is under utilized during the entire lifetime of the flash, such an aggressive block cleaning can be avoided. The partial block cleaning can be triggered if the number of blocks in the GCQ exceeds a certain threshold. This results in better energy utilization and increased life time of the flash at the cost of increased flash utilization. The FSM in Figure 4 would require changes to the "Idle" state from $|GCQ| = 0$ to $|GCQ| > Threshold$ i.e., initiate partial block cleaning only if the current length of the GCQ exceeds a certain threshold of the total GCQ length i.e., $N(\kappa + 1)/2$. Implementing a threshold of 80% resulted on an average of 4% reduction for every write access to the flash and an increase of 1.4% for every read access to the flash across the entire Consumer benchmark. Note that the increase in read access is due to the fact that since less number of blocks are garbage collected, read access has to look up a larger number of OOB areas. However, since the percentage of reads is at least an order of magnitude less than writes (Table IV), the slight increase in read times is tolerable compared to lower energy in writes.

B. Priority Garbage Collection Queue

In this technique, the garbage collection queue is rearranged during the "Idle fsm" state such that the most dirty block is the first one in the queue instead of the first in first out policy used by GCQ. The advantage in using this approach is that the write length queue grows less slowly as dirty pages are reclaimed faster when using a priority queue versus using a FIFO. However, the disadvantage in this approach is that the garbage collection queue needs to be sorted during every stage of the "idle" state in the FSM (Figure 4). Instead of sorting the entire GCQ, it is possible to use an approximate priority queue by dividing the GCQ blocks into buckets depending on the percentage of dirty pages in each block.

VIII. Related Work

While there have been several block based FTLs, the real time aspect of NAND flash was first investigated by [7]. 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). Note that, 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. [7] requires file system support for special *ioctl* calls. GFTL can be run on top any unmodified file system.

Results from [7] 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 [1], the authors address soft real-time issues by modifying the file system. The techniques in [1] focus on commonly used access patterns and not strict guarantees. In [10], 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 [17]. The conclusions in [17], supports our motivation for the lack of real-time, deterministic guarantees for flash.

Dedicated flash file systems such as JFFS2, YAFFS2 are not designed for real-time systems [18] [8]. JFFS2 does garbage collection on a need basis. Only one block is erased when required. This leads to a minimum latency equal to $T_{er} + \pi T_{wr}$ and the additional overhead of reading valid pages. The YAFFS2 garbage collector on the other hand switches to an aggressive garbage collection when the file system is low on free erase blocks. Aggressive garbage collection in YAFFS2 looks at a large region compared to its default garbage collection. This can lead to latency of the order of several seconds in case the flash is highly utilized. Thus both flash file system depend on either the state or the utilization of the flash in order to perform garbage collection. GFTL on the other hand decouples garbage collection with workload or state of the flash. Moreover, due to the fact that GFTL is a flash translation layer, a regular block based file system can be run on top of GFTL. There has been recent development in the area of real-time Java. Notably, an approach from IBM research called the *metronome* [2] uses an approach similar to GFTL. However, garbage collection in Java has the advantage of not having to displace out of place updates since the updates are done in RAM. Also, due to the out of place update nature, GFTL needs to provide guarantees on the length of write queue which is not required for Metronome.

IX. Conclusion

We presented GFTL, an FTL geared towards providing deterministic service guarantees for real-time systems. Specifically, GFTL provides $O(1)$ write time and a read time that takes π (pages per block) searches of the flash OOB in the worst case. Benchmark results from a synthetic trace and a trace representing common embedded and multimedia applications were used to evaluate the efficacy of GFTL.

Partial block cleaning lets requests arrive at a rate comparable to T_{er}, the block erase time, which is a theoretical limit on responsiveness of a flash. Benchmark results show that GFTL sticks to the theoretical limits independent of the flash utilization or state. GFTL lets a developer calculate the service guarantees and size requirements from the flash specifications during design time. The flash overhead from experiments is 16% on average. The comparisons made against NFTL (with increased size) and previous work on real-time garbage collection show the efficacy of our approach. In summary, GFTL enables a highly responsive flash with strict guarantees. Thus, GFTL decouples flash response time from flash utilization or input access pattern.

References

[1] New techniques for real-time fat file system in mobile multimedia devices. *IEEE Transactions on Consumer Electronics*, 52:1–9, 2006.
[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1):285–298, 2003.
[3] A. Ban. Flash file system optimized for page-mode flash technologies. *US Patent 5,937,425*, Aug 10, 1999.
[4] A. Bityutskiy. `http://www.linux-mtd.infradead.org/doc/ubifs.html`, 2008.
[5] Canon. Vixia HD Camcoder, January 2008.
[6] L.-P. Chang and T.-W. Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage*, 1(4):381–418, 2005.
[7] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *TECS*, 3(4):837–863, 2004.
[8] A. O. Company. Yet another flash filing system. 2001.
[9] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *OSDI*, pages 25–37, 1994.
[10] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comp. Surv.*, 37(2):138–163, 2005.
[11] J. H. Howard and et. al. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.
[12] J. Katcher. Postmark: A new file system benchmark. Technical report, Net App. Inc, TR 3022, 1997.
[13] G. Lawton. Improved flash memory grows in popularity. *Computer*, 39(1):16–18, 2006.
[14] MemCon. MemCon, July 2007. `http://linuxdevices.com/news/NS6633183518.html`, 2007.
[15] W. Norcutt. IOZONE benchmark, `www.iozone.org`.
[16] One Laptop Per Child Project. `http://laptop.org`.
[17] D. Parthey. Analyzing real-time behavior of flash memories. *Diploma Thesis, Chemnitz University of Technology*, April, 2007.
[18] D. Woodhouse. Jffs: The journalling flash file system. *Ottawa Linux Symposium*, 2001.

Siddharth Choudhuri received his PhD degree from the University of California, Irvine in 2009. His research interests are in the areas of operating systems, system architecture of embedded systems, and flash file systems. He was a member of the Center of Embedded Computer Systems, University of California from 2003 - 2009.

Tony Givargis is an Associate Professor at University of California, Irvine. He received his Ph.D. from University of California, Riverside in 2001. He is the co-author of a popular text book entitled "Embedded System Design: A Unified Hardware/Software Introduction." His research is in the area of Software for Embedded Systems. He is currently investigating issues related to Real-Time Operating System (RTOS) synthesis, high-confidence embedded software, serializing compilers, and algorithmic code transformation techniques targeting embedded software.