NAND Flash Memory Garbage Collection

Johannes Schilling
FAU Erlangen-Nürnberg
johannes.schilling@fau.de

ABSTRACT

This report provides an overview and discussion of address translation and free memory management mechanisms for NAND flash memory, with special attention to real-time requirements.

NAND flash memory pages can only be written once, but must be erased in larger blocks. This means that storage management involes copying still used data pages to other blocks before triggering an erase operation. The copy duration depends on the number of pages that must be moved, while the erase operation is relatively time-consuming and non-interruptible. Both these factors pose difficulties especially for the implementation of real time systems, but (average case) general performance optimizations are still an active topic as well. This report focuses on one representative work of each part.

KEYWORDS

AKSS, real-time storage requirements, flash memory garbage collection

1 INTRODUCTION

NAND flash memory has been the default in many mobile and embedded systems with a range of real-time constraints. Whereas mobile phones or entertainment hardware typically have softer requirements, other embedded controllers have very strict timing requirements to ensure safety.

1.1 NAND Flash Memory

NAND is the predominant type of flash memory today, due to its versatile features such as non-volatility, solid-state reliability, low cost and high density [13]. It has advantages over classical spinning hard drives like physical shock resistance and read-access times independent of seeking, due to no moving parts.

NAND flash memory is organized in *pages*, which are typically 512 to 4096 bytes in size [3]. Pages are grouped into *blocks* of typically 32 or 64 pages.

NAND flash memory has the restriction that a page, which is the basic unit of read and write operations, must be erased before being rewritten in the same location. This characteristic is sometimes called *erase-before-write*. Moreover, the erase operations must be performed on entire blocks, not single pages. As traditional file systems expect to be able to freely write and re-write chunks of data onto their underlying block device, an intermediate software layer called a *flash translation layer* (FTL) is usually employed to hide this erase-before-write limitation of flash devices [5].

The FTL redirects write requests for a page into a previously erased block, and keeps an internal mapping table of logical to physical page locations. This incurs some overhead of storage space for the mapping and of erased blocks that are kept as spare for future write requests. When additional writable pages are required, the FTL must find a block to be erased. In general, this involves copying pages that are still required to some other block and updating the mapping table. This process is known as *garbage collection*. It is discussed in detail in Section 2.

1.2 Real-Time Requirements

Kopetz describes a real-time system as one in which the correctnes of the system behaviour depends not only on the logical results of the computations, but also on the physical instant at which these results are produced [8]. This means that results have a deadline by which they must arrive, in order to be useful.

Depending on the characteristics and requirements of the concrete application, two types of real-time systems can be differentiated: those where it is required that deadlines are always met—so-called *hard real-time* systems—and those where missing a deadline will not immediately result in e.g., safety problems—these are called *soft real-time* systems.

A hard real-time system must execute a set of concurrent real-time tasks in such a way that all time-critical tasks meet their specified deadlines. For some workloads this means that a schedule of the execution sequence of each process in the system is computed at compile time. This is known as *offline* or *pre-run-time* scheduling; the converse is called *online* scheduling. Computing an offline schedule requires complete prior knowledge of the task-set characteristics like maximum execution times, mutual exclusion constraints et cetera. The scheduler then only has to enforce the scheduling decisions that have already been made prior to runtime.

For system components such as the storage access and management, this means that deterministic access times with guaranteed maximum latencies are necessary in order to satisfy real-time requirements.

1.3 Potential for Optimization

The garbage collection time and the trade-offs made in implementing the flash translation layer leave some room for optimizations where different choices and trade-offs can be made.

Zhang et al. optimize garbage collection performance by delaying some steps and lazily performing them only when requested, further discussed in Section 4. Their approach guarantees upper bounds for the execution time of requests, so it is suitable for real-time applications.

In Section 5 an approach based on Reinforcement Learning proposed by Kang et al. is discussed. Here the goal is not to provide guaranteed upper bounds, but to minimize the typical latency, using a predictor for the length of idle periods between requests based on reinforcement learning.

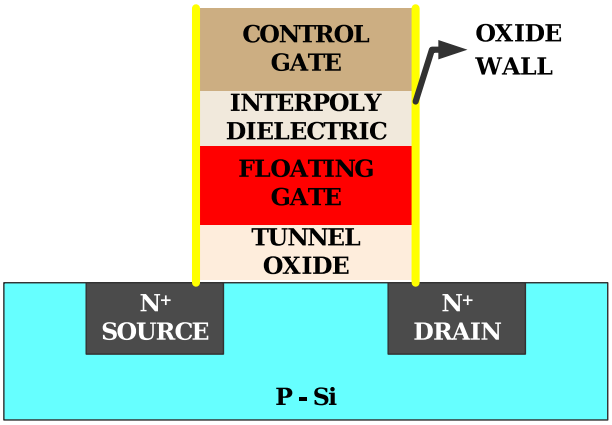

Figure 1: Schematic structure of a single NAND memory cell; reproduced from [10], Figure 2.3

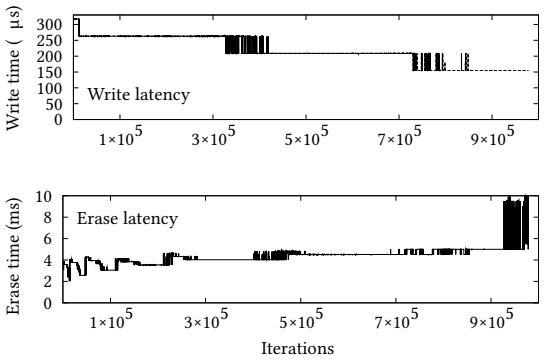

Figure 2: Wear-related changes in NAND page write and block erase latency; reproduced from [3], Figure 6.

2 BACKGROUND

NAND flash memory is organized as blocks of pages of data. A page can only be written when it was previously in erased state. Then—after being written—it keeps its value until the whole block of pages is erased again. Pages that are programmed with some data and still in use are called *live* pages, while those that are still programmed but not in use anymore are called *dead*.

2.1 NAND Flash Memory

In [10], Mohan describes the structure and inner workings of NAND flash memory.

In Figure 1, the schematic structure of a single NAND memory cell, called a *floating gate transistor* (FGT) is shown. It mimics the structure of a MOSFET transistor, with one important difference: there is an additional *floating gate* between the control gate and the source and drain gates. The floating gate is electrically insulated so that current can't flow to or from it, at least not directly.

Writing and Erasing. To store information, charge is transferred to or from the floating gate. When applying big enough voltage (typically around 18 V to 20 V) between the control gate and the substrate (cyan in Figure 1), electrons tunnel through the insulator layer into or out of the floating gate. This effect is called Fowler-Nordheim tunneling.

Reading. If the floating gate contains charge, these electrons will cause an increase the threshold voltage of the gate, so the presence or absence is used to encode one bit of information. To read the value stored in a FGT, a voltage between the unprogrammed state threshold voltage and the programmed state threshold voltage is applied at the control gate. Now the connection from source to drain will conduct exactly if the floating gate contained no charge.

Pages and Blocks. The units of writing and erasing for NAND flash memory are a page and a block, respectively. This is not due to inherent limitations of floating gate transistors, but due to economic considerations.

Power Both writing and erasing require relatively high voltages. In addition to the required voltage to trigger Fowler-Nordheim tunneling, it is also required to protect neighbouring FGTs from being charged as well by applying a counter-voltage to the substrate. These voltages typically have to be generated from lower voltage supplies. It is more efficient to batch the occasions when high voltages are required. Also, chips are layouted such that interferences only affect gates that are programmed together.

Chip space All FGTs in one block share the same substrate. This way erasing will necessarily work on the whole block, but a lot of chip space and connections are saved.

This is why practically all NAND flash storage media must be written at page granularity, and only erased at block granularity. Further details including the internal addressing of memory cells, the build-up of strings and pages of cells into blocks and finally a full flash memory device, may be found in Chapter 2 of [10].

Typical Characteristic Values. As discussed, writing and erasing is much more complex than just reading a cell's value. Desnoyers in 2010 surveyed characteristic values of available NAND flash memory devices [3]. He reports page sizes of 2K or 4K, with 16 to 128 pages forming a block. Additionally, there are usually some bytes in spare for each page, allowing to store some 32 or 64 bytes of checksums or other related data for each page.

Similarly, for the read, write and erase timings, Desnoyers found typical page read timings of 25 μs for single level cell (SLC) based and around 50 μs for multi level cell (MLC) based devices, with some early small-page SLC devices rated at 12 μs.

Write and erase timings vary with age, as seen in Figure 2. The write timings reported are largely in the 200 μs to 300 μs spectrum, with some outliers towards longer latencies for devices with bigger page sizes.

Erasing—being subject to wear as well—takes between 2000 μs and 4000 μs, with latencies up to 10 000 μs including wear-induced delays.

2.2 The Flash Translation Layer (FTL)

Traditional file systems expect to read, write, overwrite or erase blocks freely. Hardware interfaces for storage devices, like SATA

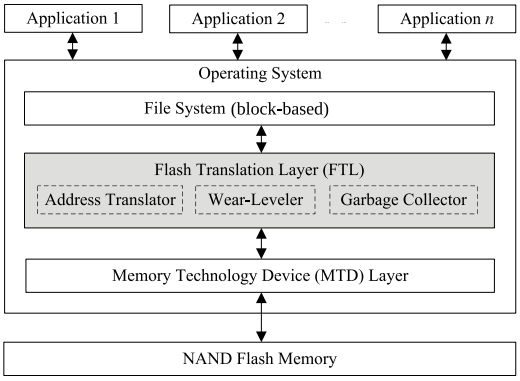

Figure 3: Flash Translation Layer overview and embedding into the OS driver stack; reproduced from [15], Figure 1

and SAS, operate on the block layer as well, where blocks are expected to be overwritable. To fit what NAND flash offers to these requirements, an adaption typically called *flash translation layer* is employed, either as part of the device driver for a NAND flash memory device, or in the case of SSDs as part of the disk controller firmware that exports a standard SATA or SAS interface.

The Real-Time Flash Translation Layer paper [15] contains a schematic overview of the layers involved when accessing data on a NAND flash memory device, reproduced with minor adjustments as Figure 3.

There exist quite a number of approaches in the literature as to how flash translation layers are designed [2, 9, 12, 18]. As there are various criteria to optimize by (like real-time suitability, but also space efficiency, good crash recovery, good wear-leveling, low runtime memory usage, ...) the focus is on papers that previously concerned themselves with real time suitability as well.

Figure 3 is very instructive, in that the main task of the flash translation layer is to translate addresses between the NAND flash device and the block-based file system. Ideally it will arrange for balanced wear over all flash memory pages, although it is not obvious which pages are going to fail soon, so there is some leeway in what a "good" wear-leveling algorithm is. And lastly, the FTL has to find a "good" (whatever that means in the specific application context) garbage collection mechanism to provide new free blocks as needed. These parts are of course inter-connected, and often cannot hold their promises without relying on each other. This is why all FTL parts are inter-connected and influence each others' behaviour.

An interesting trade-off is the granularity of the mapping. Both block- and page-granularity FTLs exist. Mapping at the granularity of whole blocks significantly reduces the amount of RAM required for the mapping table while potentially incurring overhead when only small portions of data are changed and a whole block of data (instead of just a page) needs to be written to a new flash location with the updated data. Hu et al. construct a flash translation layer that aims to achieve the performance of page-mapping with only the RAM cost of block-mapping, discussing this trade-off further along the way.

Flash File Systems. Special file systems exist for NAND flash storage media, that might be able to combine steps and therefore optimize performance, durability or power consumption, but still have to work with the interface the NAND flash memory offers. The focus here is on the flash translation layer as a non-integrated, separate abstraction below file systems.

2.3 Garbage Collection Duration

When a new request to write data arrives at the flash translation layer, it must check to see if enough pages are free, and if necessary start garbage collection.

Once a NAND flash memory block has been designated to be garbage collected, all remaining live pages in that block need to be moved elsewhere. The selected block is also called *victim block*. Moving pages includes copying their data and adjusting any references to the old page location. This takes linear time in the number of pages that have to be copied. Increasing wear of the device actually lowers the time required to write a page (while it has no effect on the read performance), so an upper bound here is easy to find. Erasing has a more or less constant cost per block; at least it can be approximated by the worst-case expected time cost at the highest wear level for the device.

What makes the garbage-collection duration hard to statically determine is the fact that it's not usually known ahead of time how full each block is, that is the linear time in moving live pages around has to either be estimated very conservatively as one less than the number of pages per block, or can't be simply known statically at all.

3 REAL-TIME FLASH MEMORY GARBAGE COLLECTION

Stankovic defines a real-time system as one, in which "the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced" [17]. For NAND flash memory garbage collection, this mainly means that the garbage collection time needs to have an upper bound, while guaranteeing that write requests can be serviced, as long as the storage is not full.

3.1 Performance Indicators for Garbage Collection Algorithms

Note that it is not obvious what metrics to apply when comparing garbage collection mechanisms. As discussed in Subsection 2.2, there are many aspects to optimize for; equally many use-cases and therefore access patterns can be found. It is clear that mechanisms will likely fare a lot better in the domain that they were intended for; it is a lot less clear, what a "fair" benchmark is, so against which workload or set of requests to compare implementations.

Also, it is not always clear what optimizations authors were going for from their proposed methods, as are applicable to a broad range of systems, from very resource-limited embedded systems where scraping kilobytes off the RAM usage is actually worth an increase in the complexity of a scheme over industrial control systems where limits are less constrained to high-end server systems that aim to fairly distribute resources among their tenant virtual systems.

Accordingly, some sample access patterns to the stored data—also called *traces*—exist from various sources, but it's hard to find one that all FTL implementations were tested against. Also, it is questionable whether algorithms that purportedly were designed with tough resource constraints in mind are adequately showcased by desktop web-browsing file access patterns.

3.2 Real-Time Flash Translation Layer (RFTL)

Qin et al. proposed a Real-Time Flash Translation Layer (RFTL) [15] mechanism for NAND flash memory in 2012. For each block exported to the layers above, three physical blocks on the NAND medium were reserved: a primary block, a replacement block, and a buffer block. When the primary block is full, data is written to the replacement block instead, and a number of pages is copied to the buffer block[1]. This way the primary block will have no more live pages once the replacement block is full, so only the cost of performing an erase on the primary block is left. Then the blocks are swapped around, so as to always have one block filling up, one possibly being emptied and one spare. With that, they get an inherent reduction in capacity to 33 % of the original memory capacity, but can guarantee response times.

To avoid high RAM cost, RFTL makes use of the so-called *OOB area*, the spare memory cells that accompany each page of flash memory, to store page mapping information.

Evaluation. Due to the ahead-of-time reservation of three physical blocks per data block, RFTL can only make use of $\frac{1}{3}$ of the available space of the NAND flash memory device.

By using the OOB area, RAM usage is limited, at the cost of an additional read/write cycle for each access/write. This makes sense for very memory-constrained systems. On the other hand, the benchmarks presented in the paper use the "Financial" and "Multimedia" file access traces; and a lot of good numbers come out of average case analyses. This both makes a lot more sense in a desktop environment than in embedded real-time scenarios. It remains unclear what the actual goals of the authors were here.

4 LAZY REAL-TIME GARBAGE COLLECTION (LAZY-RTGC)

Building on RFTL and others, Zhang et al. propose an algorithm called Lazy Real-Time Garbage Collection (Lazy-RTGC) [19] that works by

- splitting up one large garbage collection pass into separate, time-bounded steps
- delaying garbage collection until the point where it is necessary

This way they can both the garbage collection time and bound latency. They show that not only does their algorithm improve the absolute times, but also that their method always ensures enough free pages, within a bounded response time.

[1]The number of pages is not clearly specified, but they make it so that the copying takes at most as much time as an erase cycle would, so the erase cycle length still dominates the worst-case latency. This depends on the fact that reading and writing is much faster than erasing.

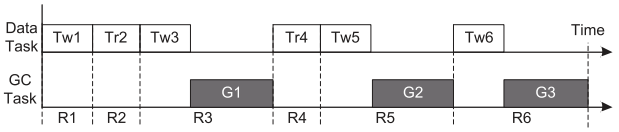

Figure 4: Scheduling of garbage collection tasks after write tasks in Lazy-RTGC; reproduced from [19], part of Figure 2.

Basic Operation. The basic mapping is page-based, to allow for maximum flexibility. The page mapping table is kept in RAM, to allow updates without NAND write overhead[2]

For each task that writes to flash memory, a garbage collection task is scheduled if necessary, i.e. if the garbage collection free pages threshold is reached. This is illustrated in Figure 4.

Garbage collection tasks each do one of two things: either move a number of live pages, or run one erase cycle. As erase cycles take up to an order of magnitude longer, but are not interruptible, so they force a relatively high lower bound for the worst-case latency. Therefore, the number of pages copied is picked (with respect to hardware specifications) such that as many pages are copied as it would take time to do an erase.

If the number of free pages drops below the configured threshold, the garbage-collection tasks gradually restoring the number of free pages are scheduled after the next write tasks, and the final garbage collection task performing the block erase restores the desired state. This can be afforded, because Lazy-RTGC only advertises a fraction (like 90 %) of the actual storage capacity to the layers above. That fraction is also what guarantees that there will always be blocks that are not completely full, i.e. that garbage collection of the block with least live pages will actually yield a net positive free page count (in this case at least 10 %) as result.

The free page requirements for each task are assumed to be known, and you basically have to choose your hardware in a way the bounds match.

4.1 Evaluation

Compared to RFTL, Lazy-RTGC can use a much larger fraction of the memory for actual user data. The authors have gone a long way in comparing their mechanism to four previously proposed methods, improving the worst-case response time in comparison with three of them, while staying level with RFTL, and improving the average response time compared to RFTL while staying comparable or ahead of the other three methods here as well. Sadly, only graphs, not exact numbers, were provided here. The graphs for the worst- and average-case response times are included as Figure 5. Each of these subgraphs is an average over 5 tested usage scenarios.

This is a very solid result, and probably the reason why the Lazy-RTGC algorithm is the only (serious) comparison in the benchmarks for the reinforcement learning based approach proposed by Kang et al., discussed in the next section. Even more so, as Lazy-RTGC is suitable for real-time environments, while the reinforcement learning-supported approach offers no real-time guarantees.

[2]Note that this means that a table of maximum size is kept in RAM at all times, even with very low actual space utilization on the flash memory. The authors also propose an on-demand page mapping mechanism as an extension to their initial algorithm.

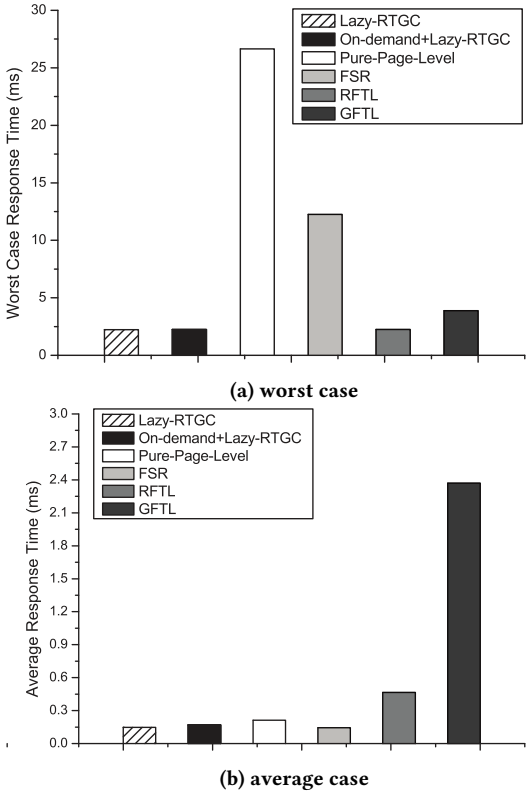

Figure 5: Lazy-RTGC latencies; reproduced from [19], part of Figure 6. Note the different scales!

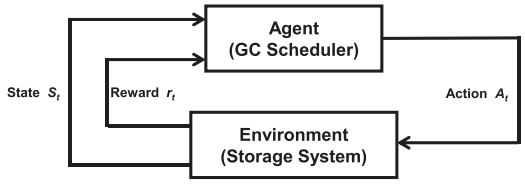

Figure 6: Interaction of the system parts in Reinforcement Learning; reproduced from [6], Figure 5.

5 REINFORCEMENT LEARNING BASED OPTIMIZATION OF TAIL LATENCIES

In their article "Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD" [6], Kang et al. attempt to optimize what they call *long-tail latencies*, that is the relatively long delays when significant garbage collection has to be performed in order to service a request.

5.1 Reinforcement Learning

The core idea of Reinforcement Learning is shown in Figure 6.

The (learning) *Agent* has access to information about the environment and can store its own state. Based on that information, it issues *Action*s onto the environment. Each action will generate a *Reward*, positive or negative, that the agent can feed back into its knowledge, thereby deepening its understanding and ideally making future actions more informed. The agent's goal is to maximize the reward.

5.2 Reinforcement Learning-Assisted Garbage Collection

In this case the agent is the garbage collection scheduler. The environment has states, e.g. active or idle. The agent issues actions of the form "copy *n* pages" or "erase block", and is rewarded according to the actual latency of the request.

The agent is triggered in discrete time steps, in a lazy manner. It is not running e.g. as a background system service that could constantly monitor the flash memory. Instead, when a storage (read or write) request arrives, first the actual request is serviced. Then the agent's next action is performed, and the total latency of servicing the request and the agent's action is fed back to the agent as reward.

The agent has two modes, called *exploration* and *exploitation*. With a very low probability (1 % in their paper), instead of what the internal model deems best, a random action is emitted. This way, new possibilities are explored. Otherwise, the existing knowledge of the agent is used.

Manual Interventions. There are still situations, where the decision (random or otherwise) by the agent is overridden:

- They seem to (wording unclear) always trigger an erase if an erase-ready block exists, and
- They have to manually trigger "intensive GC" when otherwise no free memory exists.

The necessity of both of these manual interventions can be attributed to the agent not being "intelligent" in a meaningful way, so to avoid getting stuck, these sanity measures seem necessary.

5.3 Evaluation

In their paper [6], Kang et al. compare their approach to a baseline page-based FTL with no further cleverness regarding garbage collection, and the Lazy-RTGC method presented in Section 3.

This could indicate that in their eyes, Lazy-RTGC has taken somewhat of a de-facto standard position. In citeZhang2015:LazyRTGC, the Lazy-RTGC authors still took it upon themselves to compare their algorithm to four different others (Pure Page-Level, FSR, RTFL, GFTL).

As the reinforcement learning based approach doesn't bound the time required for each operation, they also give percentiles in their evaluation. Note that the numbers in Figure **??** are normalized to the latencies of Lazy-RTGC at the same percentile, so the numbers are highly relative.

The traces "home2", "webmail" and "msnsfs" are credited in [6] as stemming from a set of "six workloads from FIU and two workloads from Microsoft"[3]. These are all write-intensive traces, suitable for benchmarking garbage collection.

Note that the normalization in the table in Figure 8 is relative to the respective value for Lazy-RTGC for that case, so these numbers only represent the relative score at that specific percentile point.

[3]The referenced source is http://iotta.snia.org. The named traces were not available on that website, but Kang et al. specify "write ratio", "avg. interval" and "avg. request size" for "home2" (91 % / 320 548 μs / 9.4 KB), "RBESQL" (82 % / 11 664 KB / 57.85 KB) and "MSNSFS" (67 % / 739 μs / 21.67 KB)

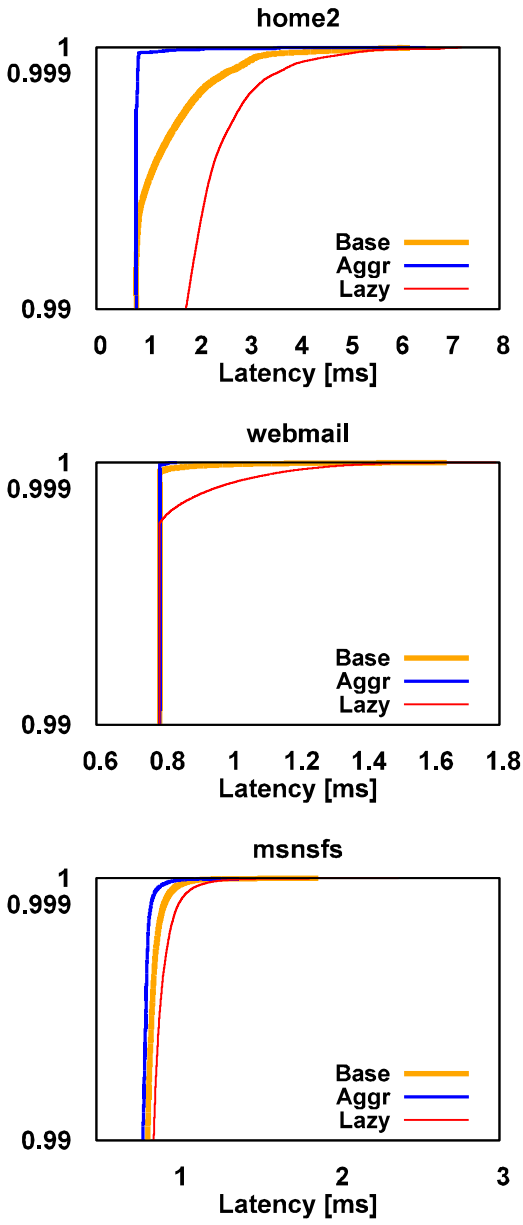

Figure 7: Comparison of write latencies of Base and Aggressive profile of Reinforcement Learning based method with Lazy-RTGC in different usage scenarios; reproduced from [6], part of Figure 7.

6 DISCUSSION

Various tradeoffs are made by the discussed mechanisms for NAND flash memory garbage collection.

Real-Time Flash Translation Layer [15], discussed in Subsection 3.2, only provides a third of the physically available space to the application. It uses the OOB area—typically intended for checksums over a page—to store page mapping information. The authors seem very keen on lowering the RAM usage of their algorithm,

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
99.9999th	Page	465	239	N/A	962	514	697	9353	2246	33.1	1813
	Lazy	1.00	1.00	N/A	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.75	0.86	N/A	0.87	0.82	0.89	0.82	0.81	1.10	0.86
	Aggr	0.58	0.90	N/A	0.83	0.35	0.48	0.88	0.92	1.14	0.76
99.99th	Page	769	292	127	1105	679	848	6396	3435	62.9	1823
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.65	0.90	1.00	0.88	0.66	0.69	0.86	0.84	1.00	0.94
	Aggr	0.50	0.27	1.00	0.88	0.47	0.58	0.84	0.85	1.06	0.71
99th	Page	6.67	3.95	1.00	5.93	6.06	5.79	5077	10.5	2.91	568
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	1.00	0.44	1.00	1.00	0.98	1.00	0.95	1.00	0.99	0.92
	Aggr	1.00	0.44	1.00	1.00	0.93	1.00	0.94	1.00	0.98	0.92

Figure 8: Comparison of write latencies of Reinforcement Learning based method with Lazy-RTGC by latency percentiles, normalized to Lazy-RTGC times; reproduced from [6], Table 5.

which would indicate a target audience of small embedded devices. They trade this for an increase in latency, as all operations need to read mapping information from the NAND memory instead of the RAM. On the other hand, they benchmark their algorithm with typical desktop workloads; a completely different environment which would not require the very strict RAM space reduction. It remains unclear what the intended usage scenario for RFTL is.

Lazy Real-Time Garbage Collection [19], discussed in Section 4, has significantly less trouble working around the constraints dictated by the hardware. They fail to make the full size of the NAND flash memory usable as well, but by a much smaller margin.

Despite being targeted at providing real-time suitable latency guarantees, both the paper on RFTL and Lazy-RTGC concern themselves quite a bit with the average-case latencies of the respective algorithms.

Reinforcement Learning based Optimization of Tail Latencies [6] is not a mechanism suitable for real-time workloads. That the only competitor they benchmark against is a real-time mechanism, and they don't outperform it by a very large margin speaks either for the excellent average case performance of Lazy-RTGC or for a very deliberate choice of the point of reference by the authors. Given that the algorithm overrides some of the choices of the reinforcement learning agent—like always erasing a block if that is currently possible—supports the conclusion that the application of reinforcement learning to the NAND flash memory garbage collection problem was only one step in research, with many more to come.

7 FUTURE WORK

For larger, desktop or server-level storage systems, devices with more than one NAND flash channel per die are gaining popularity, and are treated for example by [1, 7, 14]. With that, it would be imaginable to even get guaranteed latencies significantly below the duration of a block erase, by arranging for the channels of a flash memory device to never trigger erases at the same time, and managing pages on sets of channels such that one of a set always has free writable pages available.

Nam et al. have some ideas for future improvements in the single-channel case [11]. They propose that flash memory access scheduling should support priorities for applications or processes, so that more important parts of the system aren't easily blocked by lower-priority processes. Additionally, they imagine some kind of Quality of Service mechanism, so that each application could access the memory for its share of time. This, they argue, would improve the situation for data center workloads, where each virtualised runtime environment should have some service guarantees, so the whole system is some kind of real-time system.

So far i am not aware of any attempts to include the scheduling of flash memory actions into the offline schedule of a hard real-time system, although Nam et al. describe the need for holistic resource scheduling inside a real-time system.

A promising direction that others have explored, but was not the focus of this report, are file systems incorporating (and partly eliminating) the flash translation layer, based on the realization that neither the flash memory device nor the file system really depend on this abstraction from spinning-disk times, and each side can save the expense to adapt to it. One such example is [16].

The paper on Reinforcement Learning based Optimization of Tail Latencies [6] includes (as Figure 2) a distribution of idle times between requests, reproduced here as Figure 9. The authors don't specify exactly which workload this is based on, but reference [19] and [20] whenever they talk about "real world workloads". It seems like an interesting future direction to investigate a constantly-active flash device monitoring service, attempting to utilize these completely idle times for garbage collection. The approach by Kang et al. cannot use these because it is only triggered on requests. Moreover, the partial GC overhead fully counts towards the service time in [6], whereas in a background service model only the time from request arrival until completion of the currently ongoing uninterruptible action would actually cause a service delay. This would still need some kind of oracle, be it Reinforcement Learning based or otherwise, and much of the success would depend on the quality of this oracle.

8 CONCLUSION

NAND flash memory is an interesting technology for a broad range of usage scenarios, from very resource-limited embedded (real-time) systems that need to account for every kilobyte of RAM up to large data center systems.

In principle, NAND flash memory is well suited for real-time applications, due to relatively constant access times (especially compared to spinning disk storages that have seek times) and other properties like shock resistance that may prove beneficial in the target environment. The management of NAND flash memory poses some problems to real-time system engineers. Different approaches to solve them were investigated.

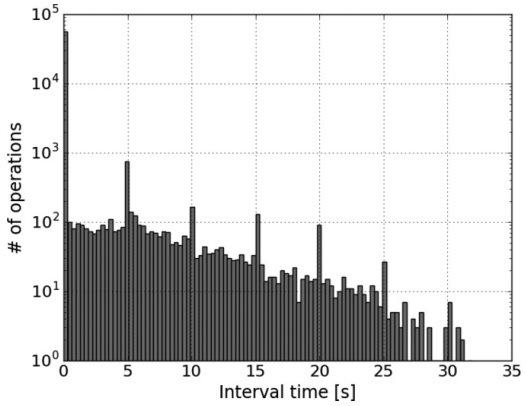

Figure 9: Distribution of idle times between memory requests; reproduced from [6], Figure 2.

Researchers seem to reach agreement that the strategy to break garbage collection up into pieces that take roughly as much time as the biggest uninterruptible operation—erasing a block—is the most promising strategy in order to build a real-time garbage collection mechanism.

In the future, both new hardware—like multi-channel NAND flash devices, that support multiple simultaneously active operations—and ever-changing requirements will trigger further advances in the research into NAND flash memory management software.

REFERENCES

[1] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Trans. Storage* 12, 3, Article Article 13 (May 2016), 39 pages. https://doi.org/10.1145/2818376

[2] H. Chen, C. Li, Y. Pan, M. Lyu, Y. Li, and Y. Xu. 2019. HCFTL: A Locality-Aware Page-Level Flash Translation Layer. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 590–593. https://doi.org/10.23919/DATE.2019.8715252

[3] Peter Desnoyers. 2010. Empirical Evaluation of NAND Flash Memory Performance. *SIGOPS Oper. Syst. Rev.* 44, 1 (March 2010), 50–54. https://doi.org/10.1145/1740390.1740402

[4] Y. Hu, H. Jiang, D. Feng, L. Tian, S. Zhang, J. Liu, W. Tong, Y. Qin, and L. Wang. 2010. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. (May 2010), 1–12. https://doi.org/10.1109/MSST.2010.5496970

[5] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, Jin-Soo Kim, Jin-Soo Kim, Jin-Soo Kim, Jin-Soo Kim, and Joonwon Lee. 2006. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT '06)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/1176887.1176911

[6] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. 2017. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article Article 134 (Sept. 2017), 20 pages. https://doi.org/10.1145/3126537

[7] M. Kim, W. Jung, H. Lee, and E. Chung. 2019. A Novel NAND Flash Memory Architecture for Maximally Exploiting Plane-Level Parallelism. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 8 (Aug 2019), 1957–1961. https://doi.org/10.1109/TVLSI.2019.2905626

[8] Hermann Kopetz. 2011. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media.

[9] Q. Luo, R. C. C. Cheung, and Y. Sun. 2018. Dynamic Virtual Page-Based Flash Translation Layer With Novel Hot Data Identification and Adaptive Parallelism Management. *IEEE Access* 6 (2018), 56200–56213. https://doi.org/10.1109/ACCESS.2018.2872721

[10] Vidyabhushan Mohan. 2010. *Modeling the Physical Characteristics of NAND Flash Memory*. Master's thesis. University of Virginia.

[11] E. H. Nam, K. J. Kim, and K. Kim. 2014. Issues and challenges of real-time flash storage. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*. https://doi.org/10.1109/ISCE.2014.6884543

[12] C. Park, W. Cheon, Y. Lee, M. Jung, W. Cho, and H. Yoon. 2007. A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash based Applications. In *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*. 202–208. https://doi.org/10.1109/RSP.2007.8

[13] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim. 2003. Cost-efficient memory architecture design of NAND flash memory embedded systems. In *Proceedings of the 21st International Conference on Computer Design*. 474–480. https://doi.org/10.1109/ICCD.2003.1240943

[14] S. Park, S. Ha, K. Bang, and E. Chung. 2009. Design and Analysis of Flash Translation Layers for Multi-Channel NAND Flash-based Storage Devices. *IEEE Transactions on Consumer Electronics* 55, 3 (August 2009), 1392–1400. https://doi.org/10.1109/TCE.2009.5278005

[15] Z. Qin, Y. Wang, D. Liu, and Z. Shao. 2012. Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems. In *Proceedings of the 18th IEEE Real Time and Embedded Technology and Applications Symposium*. 35–44. https://doi.org/10.1109/RTAS.2012.27

[16] Seung-Ho Lim and Kyu-Ho Park. 2006. An efficient NAND flash file system for flash memory storage. *IEEE Trans. Comput.* 55, 7 (July 2006), 906–912. https://doi.org/10.1109/TC.2006.96

[17] John A. Stankovic. 1988. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer* 21, 10 (1988), 10–19.

[18] Dong Bin Yeo, Joon-Yong Paik, and Tae-Sun Chung. 2019. Hierarchical Request-Size-Aware Flash Translation Layer Based on Page-Level Mapping. *Journal of Circuits, Systems and Computers* 28, 07 (2019), 1950117. https://doi.org/10.1142/S0218126619501172 arXiv:https://doi.org/10.1142/S0218126619501172

[19] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. 2015. Lazy-RTGC: A Real-Time Lazy Garbage Collection Mechanism with Jointly Optimizing Average and Worst Performance for NAND Flash Memory Storage Systems. *ACM Trans. Des. Autom. Electron. Syst.* 20, 3, Article Article 43 (June 2015), 32 pages. https://doi.org/10.1145/2746236