A Real-Time Garbage Collection Mechanism for Flash-Memory Storage Systems in Embedded Systems

Li-Pin Chang and Tei-Wei Kuo
{d6526009,ktw}@csie.ntu.edu.tw
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan, ROC 106

Abstract

Flash memory technology is becoming critical in building embedded systems applications because of its shock-resistant, power economic, and non-volatile nature. Because flash memory is a write-once and bulk-erase medium, a translation layer and a garbage collection mechanism is needed to provide applications a transparent storage service. In this paper, we propose a real-time garbage collection mechanism, which provides a deterministic performance, for hard real-time systems. A wear-levelling method, which serves as a non-real-time service, is presented to resolve the endurance issue of flash memory. The capability of the proposed mechanism is demonstrated by a series of experiments over our system prototype.

1 Introduction

Flash memory is not only shock-resistant and power economic but also non-volatile. With the recent technology breakthroughs in both capacity and reliability, more and more (embedded) system applications now deploy flash memory for their storage systems. For example, the manufacturing systems in factories must be able to tolerate severe vibration, which may cause damage to hard-disks. As a result, flash memory is a good choice for such systems. Flash memory is also suitable to portable devices, which have limited energy source from batteries, to lengthen their operating time.

There are two major issues for the flash memory storage system implementation: the nature of flash memory in (1) write-once and bulk-erasing, and (2) the endurance issue. Because flash memory is write-once, the existing data cannot be overwritten directly. Instead, the newer version of the data will be written to some available space elsewhere. The old version of the data is then invalidated and considered as "dead". The latest version of the data is considered as "live". As a result, physical locations of data change from time to time because of the adoption of the out-place-update scheme. A bulk erasing could be initiated when flash-memory storage systems have a large number of live and dead data mixed together. A bulk erasing could involve a significant number of live data copyings since the live data on the erased region must be copied to somewhere else before the erasing. That is so called *garbage collection* to recycle the space occupied by dead data. However, each erasable unit of flash memory has a limited cycle count in erasing, which imposes another restrictions on flash memory management strategies (the endurance issue).

In the past work, various techniques were proposed to improve the performance of garbage collection for flash memory, e.g., [1, 7, 8]. In particular, Kawaguchi, et al. proposed the *cost-benefit* policy [1], which uses a value-driven heuristic function as a block-recycling policy. Chiang, et al. [8] refined the above work by considering the locality in the run-time access patterns. Kwoun, et al. [7] proposed to periodically move live data among blocks so that blocks have more even life-times. Although researchers have proposed excellent garbage-collection policies, there is little work done in providing a deterministic performance guarantee for flash-memory storage systems. It has been shown that garbage collection could impose almost 40 seconds of blocking time on time-critical tasks without proper management [10]. *This paper is motivated by the needs of a predictable garbage collection mechanism to provide a deterministic garbage collection performance in time-critical systems.*

The rest of this paper is organized as follows: Section 2 introduces the system architecture of a real-time flash memory storage system. Section 3 presents our real-time block-recycling policy, free-page replenishment mechanism, and the supports for non-real-time tasks. The proposed mechanism is analyzed in Section 4. Section 5 summarizes the experimental results. Section 6 is the conclusion and the future research.

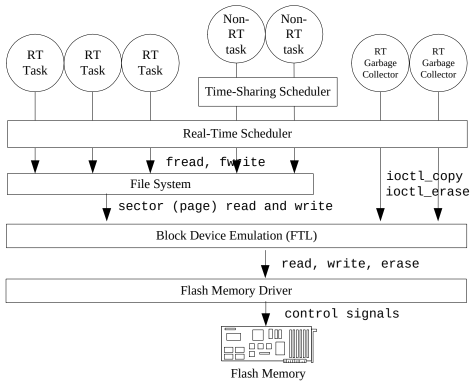


Figure 1: System Architecture.

2 System Architecture

This section proposes the system architecture of a real-time flash-memory storage system, as shown in Figure 1. We selected NAND flash to realize our storage system. There are two major architectures in flash memory design: NOR flash and NAND flash. NOR flash is a kind of EEPROM, and NAND flash is designed for data storage. We study NAND flash in this paper because it has a better price/capacity ratio, compared to NOR flash.

A NAND flash memory chip is partitioned into blocks, where each block has a fixed number of pages, and each page is of a fixed size byte array. Due to the hardware architecture, data on a flash are written in a unit of one page, and the erase is performed in a unit of one block. A page can be written only if it is erased, and a block erase will clear up all data on its pages. Logically, a page that contains live data is called a "live page", and a "dead page" contains dead (invalidated) data. Furthermore, each block has an individual limit (theoretically equivalent in the beginning) on the number of erase cycles. A worn-out block will suffer from frequent write errors. The block size of a typical NAND flash is 16KB, and the page size is 512B. The endurance of each block is usually 1,000,000 under the current technology.

There are several different approaches in managing flash memory for storage systems. We must point out that NAND flash prefers the block device emulation approach because NAND flash is a block-oriented medium (Note that a NAND flash page fits a disk sector in size). "Flash memory Translation Layer" (FTL) is introduced to emulate a block device for flash memory so that applications could have transparent storage service over flash memory. There is an alternative approach which builds a native flash memory file system over NOR flash, we refer interested readers to [5] for more details.

	Page Read 512 bytes	Page Write 512 bytes	Block Erase 16K bytes
Performance(μs)	348	909	1,881
Symbol	t_r	t_w	t_e

Table 1: Performance of the NAND Flash Memory (controlled through the ISA bus)

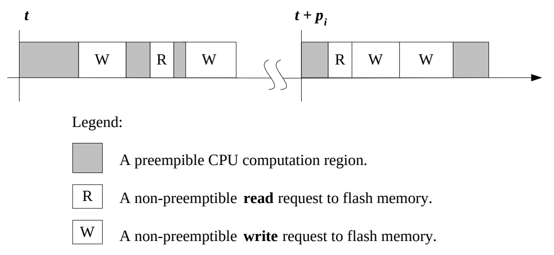
We propose to support real-time tasks and reasonable services to non-real-time tasks through a real-time scheduler and a time-sharing scheduler, respectively. A real-time garbage collector is initiated for each real-time task which might write data to flash memory to reclaim free pages for the corresponding task. Real-time garbage collectors interact directly with the FTL though the special designed control services. The control services includes *ioctl_erase* which performs the block erasing, and *ioctl_copy* which performs the atomic copying. Each atomic copy, that consists of a page read then a page write, is designed to realize the live-page copying in garbage collection. The read-then-write operation is non-preemptible in order to prevent the possibility of any race condition. Note that there is no garbage collection tasks created for the non-real-time tasks. Instead, FTL must reclaim an enough number of free pages for non-real-time tasks, similar to typical flash-memory storage systems, so that reasonable performance is provided.

3 Real-Time Garbage Collection Mechanism

3.1 Characteristics of Flash Memory Operations

A typical flash memory chip supports three kinds of operations: Page read, page write, and block erase. The performance of the three operations measured on a real prototype is listed in Table 1. The prototype has an average read and write speed at 1.4MB/sec and 540KB/sec, respectively, if no garbage collection activities are involved. Block erases take a much longer time, compared to others. As we mentioned in Section 2, we adopted a special (ioctl_copy) control services to process live-page copying for real-time garbage collection. The atomic copy operation copies data from one page to another by the specified page addresses. From the perspective of FTL's clients, the FTL is capable in processing page read, page write, block erase, and atomic copy.

Flash memory is a programmed-I/O device. In other words, flash memory operations are very CPU-consuming. As a result, the CPU is fully occupied during the execution of each flash memory operation.



t
t + p_i
W
R
W
R
W
W
Legend:
A preempible CPU computation region.
A non-preemptible read request to flash memory.
A non-preemptible write request to flash memory.

Figure 2: A task T_i which reads and writes flash memory.

On the other hand, flash memory operations are non-preemptible since the operations can not be interleaved with one another. We can treat flash memory operations as non-preemptible portions in tasks. As shown in Table 1, the longest non-preemptible operation among them is a block erase, which takes 1,881 μs to complete.

3.2 Real-Time Task Model

This section illustrate how a real-time task represents its requirements for flash memory storage service and timing constraint: Each periodic real-time task T_i is defined as a triple $(c_{T_i}, p_{T_i}, w_{T_i})$, where c_{T_i}, p_{T_i}, and w_{T_i} denote the CPU requirements, the period, and the maximum number of its page writes per period, respectively. The CPU requirements c_i consists of the CPU computation time and the flash memory operating time: Suppose that task T_i wishes to use CPU for $c^{cpu}_{T_i}$ μs, read i pages from flash memory, and write j pages to flash memory in each period. The CPU requirements c_{T_i} can be calculated as $c^{cpu}_{T_i} + i * t_w + j * t_r$ (t_w and t_r can be found in Table 1). If T_i does not wish to write to flash memory, it may set $w_{T_i} = 0$. Figure 2 illustrates that a periodic real-time task T_i issues one page read and two page writes in each period (note that the order of the read and writes does not need to be fixed in this paper).

For example, in a manufacturing system, a task T_1 might periodically fetch the control codes from files, drive the mechanical gadgets, sample the reading from the sensors, and then update the status to some specified files. Suppose that $c^{cpu}_{T_1} = 1ms$ and $p_{T_1} = 20ms$. Let T_1 wish to read a 2K-sized fragment from a data file and write 256 bytes of machine status back to a status file. Assume that the status file already exists, and we have one-page spontaneous file system meta-data to read and one-page meta-data to write. As a result, task T_1 can be defined as $(1000+(1+\frac{2048}{512})*t_r + (1+\lceil\frac{256}{512}\rceil)*t_w, 20*1000, 1+\lceil\frac{256}{512}\rceil) = (4558, 20000, 2)$. Note that the time granularity is 1μs.

Molano, et al. [2] pointed out that the access of file system meta-data may introduce unbounded delay. They proposed a meta-data pre-fetch scheme which loads necessary meta-data into RAM to provide a deterministic behavior in accessing meta-data. In this paper, we assume that each real-time task has a deterministic behavior in accessing meta-data so that we can focus on the real-time support issue for the flash memory storage systems.

symbol	Description
Λ	the total number of live pages currently on flash
Δ	the total number of dead pages currently on flash
Φ	the total number of free pages currently on flash
π	the number of pages in each block
Θ	the total number of pages on flash ($=\Lambda + \Delta + \Phi$)
α	the constant lower-bound of the number of reclaimed free pages after each block recycling
ρ	the total number of tokens in system
ρ_{free}	the number of unallocated tokens in system
ρ_{T_i}	the number of tokens given to T_i
ρ_{G_i}	the number of tokens given to G_i
ρ_{nr}	the number of tokens given to non-real-time tasks

Table 2: Symbol Definitions.

3.3 Real-Time Garbage Collection

In this section, we shall present our real-time garbage collection mechanism. We shall first propose the idea of real-time garbage collectors and the free-page replenishment mechanism for garbage collection. We will then present a block-recycling policy to choose appropriate blocks to recycle.

3.3.1 Real-Time Garbage Collectors

For each real-time task T_i which may write to flash memory ($w_{T_i} > 0$), we propose to create a corresponding real-time garbage collector G_i. G_i recycles a block in each period, and it should reclaim and supply T_i with enough free pages. Let a constant α denote a lower-bound on the number of free pages that can be reclaimed for each block recycling (we will show how to derive the lower-bound in Section 4.1). Let π denote the number of pages per block. Given a real-time task $T_i = (c_{T_i}, p_{T_i}, w_{T_i})$ with $w_{T_i} > 0$, the corresponding real-time garbage collector is created as follows:

$$\begin{aligned} c_{G_i} &= (\pi - \alpha) * (t_r + t_w) + t_e + c^{cpu}_{G_i} \\ p_{G_i} &= \begin{cases} p_{T_i} / \lceil \frac{w_{T_i}}{\alpha} \rceil & \text{, if } w_{T_i} > \alpha \\ p_{T_i} * \lfloor \frac{\alpha}{w_{T_i}} \rfloor & \text{, otherwise} \end{cases} . \end{aligned} \quad (1)$$

The CPU demand c_{G_i} consists of at most $(\pi - \alpha)$ live-page copyings, a block erase, and computation requirements $c^{cpu}_{G_i}$. The worst-case CPU requirements of

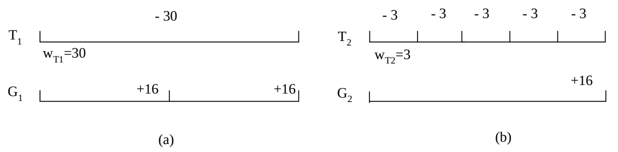


Figure 3: Creation of the Corresponding Real-Time Garbage Collectors.

all real-time garbage collectors are the same since α is a constant lower-bound. Obviously, the estimation is conservative, and G_i might not consume the entire CPU requirements in each period. The period p_{G_i} is set under the guideline in supplying T_i with enough free pages. The length of its period depends on how fast T_i consumes free pages. We let G_i and T_i arrive the system at the same time.

Figure 3 provides two examples for real-time garbage collectors, with the system parameter $\alpha = 16$. In Figure 3.(a), because $w_{T_1} = 30$, p_{G_1} is set as one half of p_{T_1} so that G_1 can reclaim 32 free pages for T_1 in each p_{T_1}. As astute readers may point out, more free pages may be unnecessarily reclaimed. We will address this issue in the next section. In Figure 3.(b), because $w_{T_2} = 3$, p_{G_2} is set to five-times of p_{T_2} so that 16 pages are reclaimed for T_2 in each p_{G_2}. The reclaimed free pages are enough for T_1 and T_2 to consume in each p_{T_1} and p_{G_2}, respectively. We define the **meta-period** σ_i of T_i and G_i as p_{T_i} if $p_{T_i} \geq p_{G_i}$; otherwise, σ_i is equal to p_{G_i}. In the above examples, $\sigma_1 = p_{T_1}$ and $\sigma_2 = p_{G_2}$. The meta-period will be used in the later sections.

3.3.2 Free-Page Replenishment Mechanism

The consuming and reclaiming of free pages during run-time usually do not behave in the worst-case. In this section, we provide a token-based "free-page replenishment mechanism" to manage the free pages and their reclaiming more efficiently and flexibly:

Consider a real-time task $T_i = (c_{T_i}, p_{T_i}, w_{T_i})$ with $w_{T_i} > 0$. Initially, T_i is given $(w_i * \sigma_i / p_{T_i})$ tokens, and one token is good for executing one page-write (Please see Section 3.3.1 for the definition of σ_i). We require that each (real-time) task could not write any page if it does not own any token. Note that a token does not correspond to any specific free page in the system. Several counters of tokens are maintained: ρ_{init} denotes the total number of available tokens when system starts. ρ denotes the total number of tokens that are currently in system (regardless of whether they are allocated and not). ρ_{free} denotes the number of unallocated tokens currently in system, ρ_{T_i} and ρ_{G_i} denote the numbers of tokens currently given to task T_i and

```
Ti()
{
    if(beginningOfMetaPeriod())
    {
      // give up the extra tokens
      $x = \rho_{Ti} - \frac{(w_{Ti} * \sigma_i)}{p_{Ti}};$
      $\rho_{Ti} = \rho_{Ti} - x; \rho = \rho - x;$
    }
    ....... /* job of Ti */
}
```

```
Gi()
{
    if( $(\Phi - \rho) \geq \alpha$ )
    {
        // create tokens from the
        // existing free pages
        $\rho_{Gi} = \rho_{Gi} + \alpha; \rho = \rho + \alpha;$
    }
    else
    {
        // Recycle a block
        recycleBlock();
        // Note that $\Phi, \rho, \rho_{Gi}$ change
    }
    // Supply Ti with $\alpha$ tokens
     $\rho_{Ti} = \rho_{Ti} + \alpha; \rho_{Gi} = \rho_{Gi} - \alpha;$

    // Give up the residual tokens
     $z = \rho_{Gi} - (\pi - \alpha); \rho_{Gi} = \rho_{Gi} - z; \rho = \rho - z;$

}
```

Figure 4: The Algorithm of the Free-Page Replenishment Mechanism.

G_i, respectively. The symbols for token counters are summarized in Table 2.

Initially, T_i and G_i are given $(w_{T_i} * \sigma_i / p_{T_i})$ and $(\pi - \alpha)$ tokens, respectively. It is to prevent T_i and G_i from being blocked in their first meta-period. During their executions, T_i and G_i are more like a pair of consumer and producer for tokens. G_i creates and provides tokens to T_i in each of its period p_{G_i} because of its reclaimed free pages in block-recycling. The replenishment of tokens are enough for T_i to write pages in each of its meta-period σ_i. When T_i consumes a token, both ρ_{T_i} and ρ are decreased by one. When G_i reclaim a free page and, thus, create a token, ρ_{G_i} and ρ are both increased by one. Basically, by the end of each period (of G_i), G_i provides T_i the created tokens in the period. When T_i and G_i terminates, their tokens must be returned to the system.

The above replenishment mechanism has some problems: First, real-time tasks might consume free pages slower than what they declared. Secondly, the real-time garbage collectors might reclaim too many free pages than what we estimate in the worst-case. Since G_i always replenish T_i with at least α created tokens, tokens would gradually accumulate at T_i. The other problem is that G_i might unnecessarily reclaim free pages even though there are already sufficient free pages in the system. Here we propose to refine the basic replenishment mechanism as follows:

In the beginning of each meta-period σ_i, T_i gives up $\rho_{T_i} - (w_{T_i} * \sigma_i / p_{T_i})$ tokens and decreases ρ_{T_i} and ρ by the same number because those tokens are beyond the needs of T_i. In the beginning of each period of G_i, G_i also checks up if $(\Phi - \rho) \geq \alpha$, where a variable Φ denotes the total number of free pages currently in the system. If the condition holds, then G_i takes α free pages from the system, and ρ_{G_i} and ρ is incremented by α, instead of actually performing a block recycling. Otherwise (i.e., $(\Phi - \rho) < \alpha$), G_i initiates a block recycling to reclaim free pages and create tokens. Suppose that G_i now has y tokens (regardless of whether they are done by a block erasing or a gift from the system) and then gives α tokens to T_i. G_i might give up $y - \alpha - (\pi - \alpha) = y - \pi$ tokens because they are beyond its needs, where $(\pi - \alpha)$ is the number of pages needed for live-page copyings (done by G_i). Figure 4 provides the algorithm of the refined free page replenishment mechanism. Again, readers could find all symbols used in this section in Table 2.

3.3.3 A Block-Recycling Policy

A block-recycling policy should make a decision on which blocks should be erased during garbage collection. The previous two sections propose the idea of real-time garbage collectors and the free-page replenishment mechanism for garbage collection. The only thing missing is a policy for the free-page replenishment mechanism to choose appropriate blocks for recycling. The purpose of this section is to propose a greedy policy that delivers a predictable performance on garbage collection:

We propose to recycle a block which has the largest number of dead pages. Obviously the worst-case in the number of free-page reclaiming happens when all dead pages are evenly distributed among all blocks in the system. The number of free pages reclaimed in the worst-case after a block recycling can be denoted as:

$$\lceil \pi * \frac{\Delta}{\Theta} \rceil, \quad (2)$$

where π, Δ, and Θ denote the number of pages per block, the total number of dead pages on flash, and the total number of pages on flash, respectively. We refer readers to Table 2 for the definitions of all symbols used in this paper.

Formula 2 denotes the worst-case performance of the greedy policy, which is proportional to the ratio of the numbers of dead pages and pages on flash memory. That is an interesting observation because we can not obtain a high garbage collection performance by simply increasing the flash memory capacity. We must emphasize that π and Θ are constant in a system. A proper greedy policy which properly manages Δ would result in a better lower-bound on the number of free pages reclaimed in a block recycling.

Because $\Theta = \Delta + \Phi + \Lambda$. Equation 2 could be re-written as follows:

$$\lceil \pi * \frac{\Theta - \Phi - \Lambda}{\Theta} \rceil, \quad (3)$$

where π and Θ are constants, and Φ and Λ denote the numbers of free pages and live pages, respectively, when the block-recycling policy is activated. As shown in Equation 3, the worst-case performance of the block-recycling policy can be controlled by bounding Φ and Λ.

Because it is not necessary to perform garbage collection if there are already sufficient free pages in system. The block-recycling policy could be activated only when the number of free pages is less than a threshold value (a bound for Φ). Furthermore, in order to bound the number of live pages in system (i.e., Λ), we propose to reduce the total number of the FTL-emulated LBA's (where LBA stands for "Logical Block Address" of a block device). For example, we can emulate a 48MB block device by using a 64MB NAND flash memory. We argue that this intuitive approach is affordable since the capacity of flash memory chip is increasing very rapidly. [1] For example, suppose that the block-recycling policy is always activated when $\Phi \leq 100$, and there are 32 pages in a block, the worst-case performance of the greedy policy is $\lceil 32 * \frac{131{,}072-100-98{,}304}{131{,}072} \rceil = 8$. *The major challenge in guaranteeing the worst-case performance is how to properly set a bound for Φ for each block recycling since the number of free pages in system might grow and shrink from time to time.* We shall further discuss this issue in Section 4.1.

3.4 Supports for Non-Real-Time Tasks

The objective of the system design aims at the simultaneous supports for both real-time and non-real-time tasks. In this section, we shall extend the token-based free-page replenishment mechanism in the previous section to supply free pages for non-real-time tasks. A non-real-time wear-leveller will then be proposed to resolve the endurance issue.

3.4.1 Free-Page Replenishment for Non-Real-Time Tasks

Different from real-time tasks, let all non-real-time tasks share a collection of tokens, denoted by ρ_{nr}. π tokens are given to non-real-time tasks ($\rho_{nr} = \pi$) initially for the purpose of live-page copying during garbage collection for non-real-time tasks. Before a non-real-time task issues a page write, it should check up if $\rho_{nr} > \pi$. If the condition holds, the page write is executed, and one token is consumed (ρ_{nr} and ρ are decreased by one). Otherwise (i.e., $\rho_{nr} \leq \pi$), the system must replenish itself with tokens for non-real-time tasks. The token creation for non-real-time tasks is similar to the strategy adopted by real-time garbage collectors: If $\Phi \leq \rho$, a block recycling is initiated to reclaim free pages, and tokens are created. If $\Phi > \rho$,

[1]The capacity of a single NAND flash memory chip had grown to 256MB when we wrote this paper.

then there might not be any needs for any block recycling.

3.4.2 A Non-Real-Time Wear-Leveller

As we mentioned in Section 2, each block has an individual limit on the number of erase cycles. In order to lengthen the overall life-time of flash memory, the management software should try to wear each block as evenly as possible. It is so-called "wear-levelling". In the past work, researchers tend to resolve the wear-levelling issue in the block-recycling policy. A typical wear-levelling-aware block-recycling policy might sometimes recycle the block which has the least erase count, regardless of how many free pages can be reclaimed. Note that the erase count denotes the number of erases had been performed on the block so far. This approach is not suitable to a time-critical application, where predictability is an important issue.

We propose to use non-real-time tasks for wear-levelling and separate the wear-levelling policy from the block-recycling policy. We could create a non-real-time wear-leveller, which sequentially scans a given number of blocks to see if any block has a relatively small erase count. We say that a block has a relatively small erase count if the count is less than the average by a given number, e.g., 2. One of the main reasons why some blocks have relatively small erase counts is because the blocks contain many live-cold data (which had not been invalidated / updated for a long period of time). Those blocks obviously are not good candidates for recycling, hence they are seldom erased. When the wear-leveller finds a block with a relatively small erase count, the wear-leveller first copy live pages from the block to some other blocks and then sleeps for a while. As the wear-leveller repeats the scanning and live-page copying, dead pages on the target blocks would gradually increase. As a result, those blocks will be selected by the block-recycling policy sooner or later. The idea is to "defrost" the blocks by moving live-cold data away.

4 System Analysis

The purpose of this section is to provide a sufficient condition to guarantee the worst-case performance of the block-recycling policy. As a result, we can justify the performance of the free-page replenishment mechanism. An admission control strategy is also introduced.

4.1 The Performance of the Block-Recycling Policy

The performance guarantees of real-time garbage collectors and the free-page replenishment mechanism are based on a constant α, i.e., a lower-bound on the number of free pages that can be reclaimed for each block recycling. As shown in Section 3.3.3, guaranteeing the performance of the block-recycling policy is not trivial since the number of free pages on flash, i.e., Φ, may grow or shrink from time to time. In this section, we will derive the relationship between α and Λ (the number of live pages on the flash) as a sufficient condition for engineers to guarantee the performance of the block-recycling policy for a specified value of α.

As indicated by Equation 2, the system must satisfy the following condition:

$$\lceil \frac{\Delta}{\Theta} * \pi \rceil \geq \alpha. \tag{4}$$

Since $\Delta = (\Theta - \Lambda - \Phi)$, and $(\lceil x \rceil \geq y)$ implies $(x > y - 1)$ if y is an integer, Equation 4 can be re-written as follows:

$$\Phi < \Theta * (1 - \frac{(\alpha - 1)}{\pi}) - \Lambda. \tag{5}$$

The formula shows that the number of free pages, i.e., Φ, must be controlled under Equation 5 if α has to be guaranteed (under the utilization of the flash, i.e., Λ). Because real-time garbage collectors would initiate block recyclings only if $\Phi - \rho < \alpha$ (please see Section 3.3.2), the largest possible value of Φ on each block recycling is $(\alpha - 1) + \rho$. Now let $\Phi = (\alpha - 1) + \rho$, Equation 5 can be again re-written as follows:

$$\rho < \Theta * (1 - \frac{(\alpha - 1)}{\pi}) - \Lambda - \alpha + 1. \tag{6}$$

Note that given a specified bound α and the (even conservatively estimated) utilization Λ of the flash, the total number of tokens ρ in system should be controlled under Equation 6. In the next paragraphs, we will first derive ρ_{max} which is the largest possible number of tokens in the system under the proposed garbage collection mechanism. Because ρ_{max} should also satisfy Equation 6, we will derive the relationship between α and Λ (the number of live pages on the flash) as a sufficient condition for engineers to guarantee the performance of the block-recycling policy for a specified value of α. Note that ρ may go up and down, as shown in Section 3.3.1.

We shall consider the case which has the maximum number of tokens accumulated: The case occurs when non-real-time garbage collection recycled a block without any live-page copying. As a result, non-real-time tasks could hold up to $2 * \pi$ tokens, where π tokens are reclaimed by the block recycling, and the other π tokens are reserved for live-page copying. Now consider real-time garbage collection: Within a meta-period σ_i of T_i (and G_i), assume that T_i consumes none of its

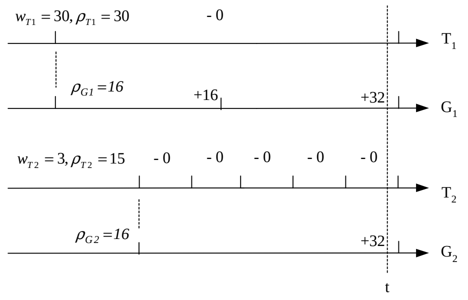


Figure 5: An Instant of Having the Largest Number of Tokens in System (Based on the Example in Figure 3.)

reserved $(w_{T_i} * \sigma_i / p_{T_i})$ tokens. On the other hand, G_i replenishes T_i with $(\alpha * \sigma_i / p_{G_i})$ tokens. In the best case, G_i recycles a block without any live-page copying in the last period of G_i within the meta-period. As a result, T_i could hold up to $(w_{T_i} * \sigma_i / p_{T_i}) + (\alpha * \sigma_i / p_{G_i})$ tokens, and G_i could hold up to $2 * (\pi - \alpha)$ tokens, where $(\pi - \alpha)$ tokens are the extra tokens created, and the other $(\pi - \alpha)$ tokens are reserved for live page copying. If all real-time tasks T_i and their corresponding real-time garbage collectors G_i behave in the same way, as described above. Figure 5 illustrates the above example based on the task set in Figure 3. The largest potential number of tokens in system ρ_{max} could be as follows, where ρ_{free} is the number of un-allocated tokens:

$$\rho_{max} = \rho_{free} + 2\pi + \sum_{i=1}^{n} \left(\frac{w_{T_i} * \sigma_i}{p_{T_i}} + \frac{\alpha * \sigma_i}{p_{G_i}} + 2(\pi - \alpha) \right). \tag{7}$$

When $\rho = \rho_{max}$, we have the following equation by combining Equation 6 and Equation 7:

$$\rho_{free} + 2\pi + \sum_{i=1}^{n} \left(\frac{w_{T_i} * \sigma_i}{p_{T_i}} + \frac{\alpha * \sigma_i}{p_{G_i}} + 2(\pi - \alpha) \right) <$$

$$\Theta * (1 - \frac{(\alpha - 1)}{\pi}) - \Lambda - \alpha + 1 \tag{8}$$

Equation 8 shows the relationship between α and Λ. We must emphasize that this equation serves as as a sufficient condition for engineers to guarantee the performance of the block-recycling policy for a specified value of α, provided the number of live pages in the system Λ, i.e., the utilization of the flash. We shall address the admission control of real-time tasks in the next section, based on α.

4.2 Admission Control

The admission control of a real-time task must consider its resource requirements and the impacts on other tasks. The purpose of this section is to derive a formula for admission control. Given a set of real-time tasks $\{T_1, T_2, ..., T_n\}$ and their corresponding real-time garbage collectors $\{G_1, G_2, ..., G_n\}$, let c_{T_i}, p_{T_i}, and w_{T_i} denote the CPU requirements, the period, and the maximum number of page writes per period of task T_i, respectively. Suppose that $c_{G_i} = p_{G_i} = 0$ when $w_{T_i} = 0$ (it is because no real-time garbage collector is needed for T_i). Since the focus of this paper is on flash-memory storage systems, the admission control of the entire real-time task set must consider whether the system could give enough tokens to all tasks initially. The verification could be done by evaluating the following formula:

$$\sum_{i=1}^{n} \left(\frac{w_{T_i} * \sigma_i}{p_{T_i}} + (\pi - \alpha) \right) \leq \rho_{free}. \tag{9}$$

Note that $\rho_{free} = \rho_{init}$ when the system starts, where ρ_{init} is the number of initially reserved tokens in the system. Beside the test, engineers should also verify the relationship between α and Λ by means of Equation 8. These verifications can be done in a linear time.

Other than the above tests, readers might want to verify the schedulability of real-time tasks in terms of CPU utilization. Suppose that the earliest deadline first algorithm (EDF) [3] is adopted to schedule all real-time tasks and their corresponding real-time garbage collectors. Since all flash-memory operations are non-preemptive, and block erasing is the most time-consuming operation, the schedulability of the real-time task set can be verified by the following formula, provided that other system overheads are ignorable:

$$\frac{t_e}{min\_p()} + \sum_{i=1}^{n} \left(\frac{c_{T_i}}{p_{T_i}} + \frac{c_{G_i}}{p_{G_i}} \right) \leq 1, \tag{10}$$

where $min\_p()$ denotes the minimum period of all T_i and G_i, and t_e is the duration of a block erase. Because every real-time task might be blocked by a block erase issued by either real-time garbage collection or non-real-time garbage collection, as a result, the longest blocking time of every task is the same, i.e., t_e. We could derive Equation 10 from Theorem 2 in T. P. Baker's work [9], which presents a sufficient condition of the schedulability of tasks which have non-preemptible portions. We must emphasize

Task	c^{cpu}	page reads	page writes(w)	c	p	c/p
T_1	3ms	4	2	6,354 μs	20ms	0.32
T_2	5ms	2	5	8,738 μs	200ms	0.05

Table 3: The Basic Simulation Workload

Task	c^{cpu}	page reads	page writes	c	p	c/p
T_1	3ms	4	2	6,354 μs	20ms	0.32
T_2	5ms	2	5	8,738 μs	200ms	0.05
G_1	10 μs	16	16	22,003 μs	160ms	0.14
G_2	10 μs	16	16	22,003 μs	600ms	0.04

Table 4: The Simulation Workload for Evaluating The Real-Time Garbage Collection Mechanism . ($\alpha = 16$, Capacity Utilization = 50%)

that the above formula only intends to deliver the idea of blocking time, due to flash-memory operations.

5 Performance Evaluation

In this section, we compare the behaviors of a system prototype with or without the proposed real-time garbage collection mechanism. We also show the effectiveness of the proposed wear-levelling method, which is executed as a non-real-time service.

5.1 Simulation Workloads

A series of experiments were done over a real system prototype. A set of tasks was created to emulate the workload of a manufacturing system which stored all files on flash memory to avoid vibration from damaging the system. The basic workload consisted of two real-time tasks T_1 and T_2 and one non-real-time task. T_1 and T_2 sequentially read the machine control files, did some computations, and updated their own (small) status files. The non-real-time task emulated a file downloader, which downloaded the machine control files continuously over a local-area-network and then wrote the downloaded file contents onto flash memory. The flash memory used in the experiments was a 16MB NAND flash [6]. The block size was 16KB, and the page size was 512B ($\pi = 32$). The traces of T_1, T_2, and the non-real-time file downloader were synthesized to emulate the necessary requests for the file system. The basic simulation workload is detailed in Table 3. The duration of each experiment was 20 minutes.

5.2 The Evaluation of Garbage Collection Mechanisms

In order to observe the behavior of a non-real-time garbage collection mechanism in a time-critical system, we evaluated a system which adopted a non-real-time garbage collection mechanism under the basic workload and a 50% flash memory capacity utilization. The non-real-time garbage collection mechanism

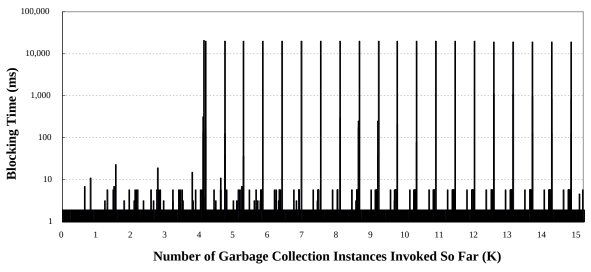


Figure 6: The Blocking Time Imposed on each Page Write by a Non-Real-Time Garbage Collection Instance.

basically followed the well-known $cost-benefit$ block-recycling policy [1], but it might recycle a block which had an erase count less than the average erase count by 2 in order to perform wear-levelling. Note that garbage collection was activated on demand. We defined that *an instance of garbage collection* consisted of all of the activities which started when the garbage collection was invoked and ended when the garbage collection returned control to the blocked page write.

We measured the blocking time imposed on a page write by each instance of garbage collection. The results in Figure 6 show that the blocking time could even reach 21.209 seconds in the worst-case. The lengthy and un-predictable blocking time was mainly caused by wear-levelling. As astute readers may notice, an instance of garbage collection might consist of recycling several blocks consecutively, because a block without dead pages might be recycled due to wear-levelling. The block recycling-policy continued recycling blocks until at least one free page was reclaimed.

We also observed that if the real-time tasks were ever been blocked due to an insufficient number of tokens, under the adoption of the proposed real-time garbage collection mechanism. The same basic configurations were used in this experiment. The corresponding real-time garbage collectors were created according to the method described in Section 3.3.1, and the workload is summarized in Table 4. T_1 and T_2 generated 12,000 and 3,000 page writes, respectively, in the 20-minute experiment. The results are shown in Figure 7. We observed no blocking time being imposed on page writes of T_1 and T_2 (as we expected).

5.3 Effectiveness of the Wear-Levelling Method

The second part of experiments evaluated the the effectiveness of wear-levelling, which was evaluated in terms of the standard deviation of the erase counts of all blocks. A lower value indicated that all blocks were

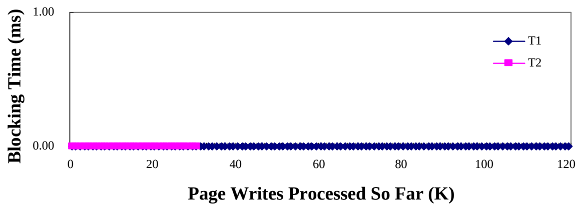


Figure 7: The Blocking Time Imposed on each Page Write of a Real-Time Task by Waiting for Tokens (Under the *Real-Time* Garbage Collection Mechanism).

erased more evenly. The objective of wear-levelling is to maintain a small value of the standard deviation.

The non-real-time wear-leveller was configured as follows: It performed live-page copyings whenever the erase count of a block was less than the average erase count by 2. The wear-leveller slept for $50ms$ between every two consecutive live-page copyings. Since the past work showed that the overheads of garbage collection highly depend on the flash-memory capacity utilization [1, 8, 4], we evaluated the experiments under different capacity utilizations: $50\%, 60\%$ and 70%.

As the results shown in Figure 8, the non-real-time wear-leveller gradually levelled the erase count of each block. The results also pointed out that the effectiveness of wear-levelling was better when the capacity utilization was low, since the system was not heavily loaded by the overheads of garbage collection. The standard deviation increased very rapidly when wear-levelling was disabled. We also observed that the the standard deviation was stringently controlled under the non-real-time block-recycling policy.

6 Conclusion

This paper is motivated by the needs of a garbage collection mechanism which is capable of providing a deterministic performance for time-critical systems. We propose a real-time garbage collection mechanism with a guaranteed performance. The endurance issue is also resolved by the proposing of a wear-levelling method. We demonstrate the performance of the proposed methodology in terms of a system prototype. However, there are some issues not addressed in this work: We should observe the overheads and performance of the proposed mechanism when the system is highly stressed. Furthermore, we shall extend the proposed mechanism to support time-critical applications over a NOR flash-based storage system.

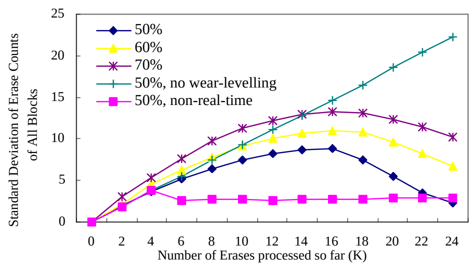


Figure 8: Effectiveness of the Wear-Levelling Method.

References

[1] A. Kawaguchi, S. Nishioka, and H. Motoda,"A Flash Memory based File System," Proceedings of the USENIX Technical Conference, 1995.

[2] A. Molano, R. Rajkumar, and K. Juvva, "Dynamic Disk Bandwidth Management and Metadata Pre-fetching in a Real-Time Filesystem," Proceedings of the 10th Euromicro Workshop on Real-Time Systems , 1998.

[3] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM , 1973.

[4] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," Proceedings of the USENIX Operating System Design and Implementation, 1934.

[5] Journaling Flash File System, http://sources.redhat.com/jffs2/jffs2-html/

[6] "K9F2808U0B 16Mb*8 NAND Flash Memory Data Sheet," Samsung Electronics Company.

[7] K. Han-Joon, and L. Sang-goo, "A New Flash Memory Management for Flash Storage System," Proceedings of the Computer Software and Applications Conference, 1999.

[8] M. L. Chiang, C. H. Paul, and R. C. Chang, "Manage flash memory in personal communicate devices," Proceedings of IEEE International Symposium on Consumer Electronics, 1997.

[9] T. P. Baker, "A Stack-Based Resource Allocation Policy for Real-Time Process," IEEE 11th Real-Time System Symposium, Dec 4-7, 1990.

[10] Vipin Malik, "JFFS2 is Broken," Mailing List of Memory Technology Device (MTD) Subsystem for Linux, June 28th 2001.