Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems

LI-PIN CHANG, TEI-WEI KUO, and SHI-WU LO
National Taiwan University

Flash-memory technology is becoming critical in building embedded systems applications because of its shock-resistant, power economic, and nonvolatile nature. With the recent technology breakthroughs in both capacity and reliability, flash-memory storage systems are now very popular in many types of embedded systems. However, because flash memory is a write-once and bulk-erase medium, we need a translation layer and a garbage-collection mechanism to provide applications a transparent storage service. In the past work, various techniques were introduced to improve the garbage-collection mechanism. These techniques aimed at both performance and endurance issues, but they all failed in providing applications a guaranteed performance. In this paper, we propose a real-time garbage-collection mechanism, which provides a guaranteed performance, for hard real-time systems. On the other hand, the proposed mechanism supports non-real-time tasks so that the potential bandwidth of the storage system can be fully utilized. A wear-leveling method, which is executed as a non-real-time service, is presented to resolve the endurance problem of flash memory. The capability of the proposed mechanism is demonstrated by a series of experiments over our system prototype.

Categories and Subject Descriptors: B.3.2 [**Design Styles**]: Mass Storage; D.4.2 [**Storage Management**]: Garbage Collection; D.4.7 [**Organization and Design**]: Real-Time Systems and Embedded Systems

General Terms: Algorithms, Design

Additional Key Words and Phrases: Embedded systems, flash memory, garbage collection, real-time system, storage systems

1. INTRODUCTION

Flash memory is not only shock resistant and power economic but also nonvolatile. With the recent technology breakthroughs in both capacity and reliability, more and more (embedded) system applications now deploy flash

This paper is an extended version of a paper appeared in the 8th International Conference on Real-Time Computing Systems and Applications, 2002.
Author's Address: Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, 106; email: {d6526009,ktw,d89015}@csie.ntu.edu.tw.

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.

A complicated management method is needed for flash-memory storage systems. There are two major issues for the system implementation: the nature of flash memory in (1) write once and bulk erasing and (2) the endurance problem. Because flash memory is write once, the flash memory management software could not overwrite existing data. Instead, the newer version of data will be written to 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." Flash-memory translation layer (FTL) is introduced to emulate a block device for flash memory so that applications could have transparent access to data that might dynamically move around different locations. Bulk erasing, which involves significant live data copying, could be initiated when flash-memory storage systems have a large number of live and dead data mixed together. That is, so-called garbage-collection with the intension in recycling the space occupied by dead data. Garbage collection directly over flash memory could result in performance overheads and, potentially, unpredictable run-time latency. It is the price paid for robustness considerations. Note that many industrial applications, such as industrial control systems and distributed automation systems, do require reliable storage systems for control data and system configurations that can survive over various kinds of power failures and operate in extreme environments (e.g., under severe vibration considerations). No matter what garbage-collection or data-writing policies are adopted, the flash-memory storage system should consider the limit of possible erasings on each erasable unit of flash memory (the endurance problem).

In the past work, various techniques were proposed to improve the performance of garbage collection for flash memory [e.g., Kawaguchi et al. 1995; Kim and Lee 1999; Wu and Zwaenepoel 1994; Chiang et al. 1997]. In particular, Kawaguchi et al. proposed the *cost–benefit* policy [Kawaguchi et al. 1995], which uses a value-driven heuristic function based on the cost and the benefit of recycling a specific block. The policy picks up the block that has the largest value of $(a \times \frac{2u}{1-u})$ to recycle, where u is the capacity utilization (percentage of fullness) of the block, and a is the time elapsed so far since the last data invalidation on the block. The cost–benefit policy avoids recycling a block that contains recently invalidated data because the policy surmises that more live data on the block might be invalidated soon. Chiang et al. [1997] refined the above work by considering a fine-grained hot–cold identification mechanism. They proposed to keep track of the hotness of each LBA, where LBA stands for the Logical Block Address of a block device. The hotness of an LBA denotes how often the LBA is written. They proposed to avoid recycle a block that contains many live and hot data because any copying of the live and hot data is usually considered inefficient. Chang and Kuo [2002, 2004] investigated garbage-collection issues over a novel striping architecture and large-scale flash-memory storage systems, in which the challenges faced are to maximize the parallelism and to

reduce the management overheads, respectively. Wu et al. [2003] proposed a native spatial index implementation over flash memory. Kim and Lee [1999] 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. For a time-critical system, such as manufacturing systems, it is highly important to predict the number of free pages reclaimed after each block recycling so that the system will never be blocked for an unpredictable duration of time because of garbage collection. It has been shown that garbage collection could impose almost 40 s of blocking time on real-time tasks without proper management [Malik 2001a, 2001b]. This paper is motivated by the needs of a predictable block-recycling policy to provide real-time performance to many time-critical systems. We shall not only propose a block-recycling policy with guaranteed performance but also resolve the endurance problem. A free-page replenishment mechanism is proposed for garbage collection to control the consumption rate of free pages so that bulk erasings only occur whenever necessary. Because of the real-time garbage-collection support, each time-critical task could be guaranteed with a specified number of reads and/or writes within its period. Non-real-time tasks are also serviced with the objective to fully utilize the potential bandwidth of the flash-memory storage system. The design of the proposed mechanism is independent of the implementation of flash-memory-management software and the adoption of real-time scheduling algorithms. We demonstrate the performance of the system in terms of a system prototype.

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. We provide the admission control strategy and the justification of the proposed mechanism in Section 4. Section 5 summarizes the experimental results. Section 6 presents the conclusion and future research.

2. SYSTEM ARCHITECTURE

This section proposes the system architecture of a real-time flash-memory storage system, as shown in Figure 1. The system architecture of a typical flash-memory storage system is similar to that in Figure 1, except that a typical flash-memory storage system does not have or consider real-time tasks, a real-time scheduler, and real-time garbage collectors.

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 either

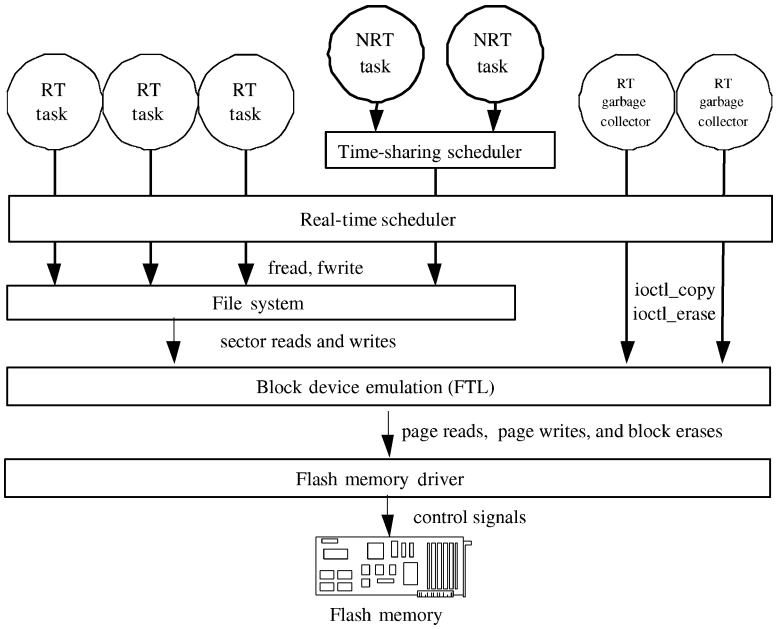

Fig. 1. System architecture.

programmable or unprogrammable, and any page initially is programmable. Programmable page are called "free pages." A programmable page becomes unprogrammable once it is written (programmed). A written page that contains live data is called a "live page"; it is called "dead page" if it contains dead (invalidated) data. A block erase will erase all of its pages and reset their status back to programmable (free). Furthermore, each block has an individual limit (theoretically equivalent in the beginning) on the number of erasings. A worn-out block will suffer from frequent write errors. The block size of a typical NAND flash is 16 KB, and the page size is 512 B. The endurance of each block is usually 1,000,000 under the current technology.

Over a real-time flash-memory storage system, user tasks might read or/and write the FTL-emulated block device through the file system,[1] where the FTL is to provide transparent accesses to flash memory through block-device emulation. We propose to support real-time and reasonable services to real-time and 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

[1]Note that there are several different approaches in managing flash memory for storage systems. In this paper, we focused on the block-device emulation approach. There are also alternative approaches that build native flash-memory file systems, we refer interested readers to [JFFS] and [YAFFS] for more details. Note that one of the major advantages of log-structured file systems [JFFS; YAFFS] is on robustness. Although the approach is very different from the block-device emulation approach for flash memory, it does provide robustness to storage systems to survive over power failures. Garbage collection might be needed to remove unnecessary logs (that correspond to dead pages for the block-device emulation approach).

Table I. Performance of NAND Flash Memory

	Page Read 512 bytes	Page Write 512 bytes	Block Erase 16K bytes
Performance (μs)	348	919	1881
Symbol	t_r	t_w	t_e

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 erase, and *ioctl_copy* which performs atomic copying. Each atomic copying, that consists of a page read then a page write, is designed to realize the live-page copying in garbage collection. In order to ensure the integrity of each atomic copying, its read-then-write operation is non-preemptible to prevent the possibility of any race condition. The garbage collection for non-real-time tasks are handled inside the FTL. 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. Note that a write by any task invocation does not necessarily make the data written by previous invocations of the same task stale. Unless the same data are updated, data written by previous invocations remain valid.

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 I. Block erases take a much longer time, compared to others. Because the garbage-collection facility in FTL might impose unbounded blocking time on tasks, we propose to adopt two FTL-supported control services to process real-time garbage collection: the block erase service (ioctl_erase) and the atomic copying service (ioctl_copy). The atomic copying operation copies data from one page to another by the specified page addresses, and the copy operation is non-preemptible to avoid any possibility of race conditions.[2] The main idea in this paper is to have a real-time garbage collector created for each real-time task to handle block erase requests (through ioctl_erase) and live-page copyings (through ioctl_copy) for real-time garbage collection. We shall illustrate the idea in more detailed in later sections.

Flash memory is a programmed-I/O device. In other words, flash-memory operations are very CPU consuming. For example: To handle a page write request, the CPU first downloads the data to the flash memory, issues the address and the command, and then monitors the status of the flash memory until the command is completed. Page reads and block erases are handled in a similar fashion. As a result, the CPU is fully occupied during the execution of flash-memory operations. On the other hand, flash-memory operations are non-preemptible

[2]Some advanced NAND flash memory had a native support for the atomic copying. The overhead of the atomic copying is significantly reduced since the operation can be done internally in flash memory without the redundant data transfer between RAM and flash memory.

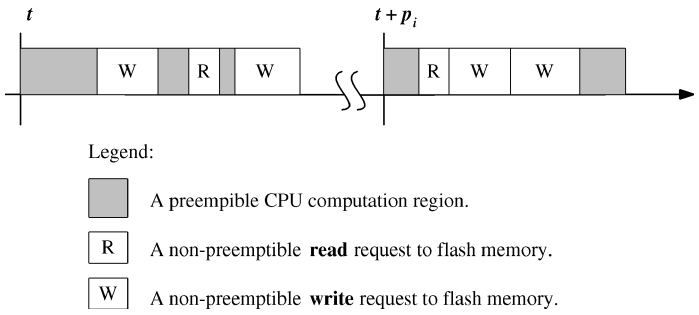

Fig. 2. A task T_i which reads and writes flash memory.

because the operations cannot be interleaved with one another. We can treat flash-memory operations as non-preemptible portions in tasks. As shown in Table I, the longest non-preemptible operation among them is a block erase, which takes 1881 μs to complete.

3.2 Real-Time Task Model

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 $c_{T_i}^{\text{cpu}}$ 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 $c_{T_i}^{\text{cpu}} + i \times t_r + j \times t_w$ (t_r and t_w can be found in Table I). 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).

Any file-system access over a block device might not only access the corresponding files but also touch their meta-data at the same time, where the meta-data of a file contains the essential information of the file, for example, the file owner, the access permissions, and some block pointers. For example, file accesses over an FAT file system might follow cluster chains (i.e., parts of the corresponding meta-data) in the FAT table, where file accesses over an UNIX file system are done over direct/indirect block pointers of the corresponding meta-data. Meta-data accesses could result in unpredictable latencies for their corresponding file-system accesses because the amount of the meta-data accessed for each file-system access sometimes could not be predicted precisely, such as those activities on the housekeeping of the entire device space. Molano et al. [1998] proposed a meta-data prefetch scheme, which loads necessary meta-data into RAM to provide a more predictable behavior on meta-data accesses. In this paper, we assume that each real-time task has a predictable behavior in the accessing of the corresponding meta-data so that we can focus on the real-time support issue for flash-memory storage systems.

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 $c_{T_1}^{\text{cpu}} = 1$ ms and $p_{T_1} = 20$ ms. Let T_1 wish to read a 2K-sized fragment from a data file, write 256 bytes of machine status back to a status file. Assume that the status file already exists, and we have one-page ancillary file-system meta-data to read and one-page meta-data to write. According to these information, task T_1 can be defined as $(1000 + (1 + \frac{2048}{512}) \times t_r + (1 + \lceil \frac{256}{512} \rceil) \times t_w, 20 \times 1000, 1 + \lceil \frac{256}{512} \rceil) = (4578, 20000, 2)$. Note that the time granularity is 1 μs. As shown in the example, it is very straightforward to describe a real-time task in our system.

3.3 Real-Time Garbage Collection

This section is meant to propose a real-time garbage-collection mechanism to prevent each real-time task from being blocked because of an insufficient number of free pages. 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 for the free-page replenishment mechanism 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 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 that in Section 4.1). Note that the number of reclaimed free pages for a block recycling is identical to the number of dead pages on the block before the block recycling. 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:

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

Equation (1) is based on the assumption that G_i can reclaim at least α pages for T_i for every period p_{G_i}. The CPU demand c_{G_i} consists of at most $(\pi - \alpha)$ live-page copyings, a block erase, and computation requirements $c_{G_i}^{\text{cpu}}$. All real-time garbage collectors have the same worst-case CPU requirements because α is a constant lower bound. Obviously, the estimation is based on a worst-case analysis, 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

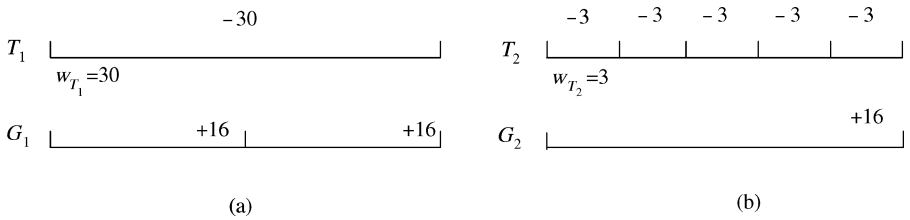

Fig. 3. Creation of the corresponding real-time garbage collectors.

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 free-page replenishment mechanism proposed in this section is to resolve the over-reclaiming issue for free pages during real-time garbage collection. The over-reclaiming occurs because the real-time garbage collection is based on the worst-case assumption of the number of reclaimed free pages per erasing. Furthermore, in many cases, real-time tasks might consume free pages slower than what they declared. A coordinating mechanism should be adopted to manage the free pages and their reclaiming more efficiently and flexibly.

In this section, we propose a token-based mechanism to coordinate the free-page usage and reclaiming. In order not to let a real-time garbage collector reclaim too many unnecessary free pages, here a token-based "free-page replenishment mechanism" is presented:

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 $(w_i \times \sigma_i / p_{T_i})$ 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 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 G_i, respectively. (G_i also needs tokens for live-page copyings, and it will be addressed later.) The symbols for token counters are summarized in Table II.

Initially, T_i and G_i are given $(w_{T_i} \times \sigma_i / p_{T_i})$ 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 garbage collection. The replenishment of tokens are

Table II. Symbol Definitions

Symbol	Description	Value
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	32
Θ	The total number of pages on flash	$\Lambda + \Delta + \Phi$
	The block size	32 KB
	The page size	512 B
α	The pre-defined 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	

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 1. When G_i reclaim a free page and, thus, create a token, ρ_{G_i} and ρ are both increased by 1. Basically, by the end of each period (of G_i), G_i provides T_i the created tokens in the period. When T_i terminates, its tokens (and those of G_i) 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. Second, the real-time garbage collectors might reclaim too many free pages than what we estimate in the worst case. Because G_i always replenish T_i with all its 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 illustrates the consuming and supplying of tokens between T_i and G_i (based on the example in Figure 3(a)), and Figure 5 provides the algorithm of the free-page replenishment mechanism. We shall justify that G_i and T_i always have free pages to write or perform live-page copyings in Section 4.4. All symbols used in this section is summarized in Table II.

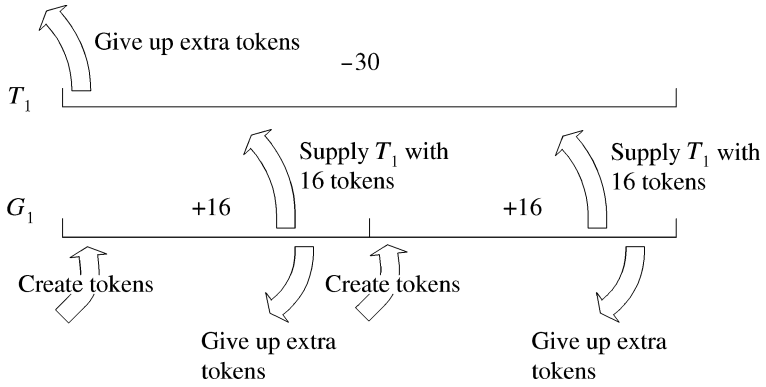

Fig. 4. The free-page replenishment mechanism.

```
Ti()
{
    if(beginningOfMetaPeriod())
    {
      // give up the extra tokens
      x=ρTi−(wTi*σi)/pTi;
      ρTi=ρTi−x;ρ=ρ−x;
    }
    ..... /* job of Ti */
}
```

```
Gi()
{
    if((Φ−ρ)≥α)
    {
        // create tokens from the
        // existing free pages
        ρGi=ρGi+α;ρ=ρ+α;
    }
    else
    {
        // Recycle a block
        recycleBlock();
        // Note that Φ,ρ,ρGi change
    }
    // Supply Ti with α tokens
      ρTi=ρTi+α;ρGi=ρGi−α;
    // Give up the residual tokens
     z=ρGi−(π−α);ρGi=ρGi−z;ρ=ρ−z;
}
```

Fig. 5. The algorithm of the free-page replenishment mechanism.

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 block-recycling policies proposed in the past work [Kawaguchi et al. 1995; Kim and Lee 1999; Wu and Zwaenepoel 1994; Chiang et al. 1997] usually adopted sophisticated heuristics with objectives in low garbage-collection overheads and a longer overall flash-memory lifetime. With unpredictable performance on garbage collection, these block-recycling policies could not be used for time-critical applications. 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 that has the largest number of dead pages with the objective in predicting the worst-case performance. 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:

$$\left\lceil \pi \times \frac{\Delta}{\Theta} \right\rceil \tag{2}$$

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 II 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 total number of dead pages to the total number of 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 that properly manages Δ would result in a better low bound on the number of free-pages reclaimed in a block recycling.*

Because $\Theta = \Delta + \Phi + \Lambda$, Equation (2) could be rewritten as follows:

$$\left\lceil \pi \times \frac{\Theta - \Phi - \Lambda}{\Theta} \right\rceil \tag{3}$$

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). For the rest of this section, we shall show the relationship between the utilization of flash memory and the minimum number of free pages obtained for each recycle of a block based on Equation (3): Suppose that we have a 64 MB NAND flash memory, and live data occupy 48 MB. The rest 16 MB are occupied by dead or free pages. Let each block consist of 32 pages. The worst-case performance of the greedy policy is $\lceil 32 \times \frac{131,072-100-98,304}{131,072} \rceil = 8$, where 131, 072 and 98, 304 are the numbers of 512 B pages for 48 and 64 MB, respectively. As shown in Equation (3), the worst-case performance of the block-recycling policy can be controlled by bounding Φ and Λ.

The major challenge in guaranteeing the worst-case performance is how to properly set a bound for Φ for each block recycling because the number of free pages in system might grow and shrink from time to time.[3] We shall further discuss this issue in Section 4.1. As astute readers may point out, the above greedy policy does not consider wear leveling because the consideration of wear leveling could result in an unpredictable behavior in block recycling. We should address this issue in the next section.

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

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 leveler will then be proposed to resolve the endurance problem.

3.4.1 *Free-Page Replenishment for Non-Real-Time Tasks.* We propose to extend the free-page replenishment mechanism to support non-real-time tasks as follows: Let all non-real-time tasks be first scheduled by a time-sharing scheduler. When a non-real-time task is scheduled by the time-sharing scheduler, it will execute as a background task under the real-time scheduler (please see Figure 1).

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 ($\rho_{\text{nr}} = \pi$) initially because FTL might need up to π tokens to recycle a block full of live pages (e.g., due to wear-leveling considerations). Note that non-real-time tasks depends on FTL for garbage collection. Before a non-real-time task issues a page write, it should check up if $\rho_{\text{nr}} > \pi$. If the condition holds, the page write is executed, and one token is consumed (ρ_{nr} and ρ are decreased by 1). Otherwise (i.e., $\rho_{\text{nr}} \leq \pi$), 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 $\Phi > \rho$, then there might not be any needs for any block recycling.

3.4.2 *A Non-Real-Time Wear Leveler.* In the past work, researchers tend to resolve the wear-leveling issue in the block-recycling policy. A typical wear-leveling-aware block-recycling policy might sometimes recycle the block that has the least number of erasing, regardless of how many free pages can be reclaimed. This approach is not suitable to a real-time application, where predictability is an important issue.

We propose to use non-real-time tasks for wear leveling and separate the wear-leveling policy from the block-recycling policy. We could create a non-real-time wear-leveler, which sequentially scans a given number of blocks to see if any block has a relatively small erase count, where the erase count of a block denotes the number of erases that has been performed on the block so far. 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). When the wear leveler finds a block with a relatively small erase count, the wear leveler first copy live pages from the block to some other blocks and then sleeps for a while. As the wear leveler repeats the scanning and live-page copying, dead pages on the blocks with relatively small erase counts would gradually increase. As a result, those blocks will be selected by the block-recycling policy sooner or later.

Note that one of the main reasons why a block has a relatively small erase count is because the block contains many live-cold data (which had not been invalidated/updated for a long period of time). The wear leveling is done by

"defrosting" blocks with relatively small erase counts by moving cold data away. The capability of the proposed wear leveler is evaluated in Section 5.3.

4. SYSTEM ANALYSIS

In the previous sections, we had proposed the idea of the real-time garbage collectors, the free-page replenishment mechanism, and the greedy block-recycling policy. 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, that is, a lower bound on the number of free pages that can be reclaimed for each block recycling. As shown in Section 3.3.3, it is not trivial in guaranteeing the performance of the block-recycling policy because the number of free pages on flash, that is, 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:

$$\left\lceil \frac{\Delta}{\Theta} \times \pi \right\rceil \geq \alpha. \tag{4}$$

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

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

The formula shows that the number of free pages, that is, Φ, must be controlled under Equation (5) if the lower bound α on the number of free pages reclaimed for each block recycling 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 $(\alpha - 1) + \rho$. Let $\Phi = (\alpha - 1) + \rho$, Equation (5) can be again rewritten as follows:

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

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 $\rho_{\max}$ which is the largest possible number of tokens in the system under the proposed garbage collection mechanism. Because $\rho_{\max}$ should also satisfy Equation (6), we will derive the relationship between α and Λ (the number of live pages on

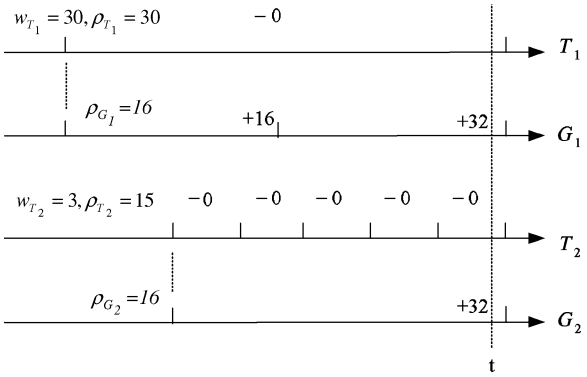

Fig. 6. An instant of having the maximum number of tokens in system (based on the example in Figure 3.)

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 that 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 $2 \times \pi$ 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 reserved $(w_{T_i} \times \sigma_i / p_{T_i})$ tokens. On the other hand, G_i replenishes T_i with $(\alpha \times \sigma_i / p_{G_i})$ 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 $(w_{T_i} \times \sigma_i / p_{T_i}) + (\alpha \times \sigma_i / p_{G_i})$ tokens, and G_i could hold up to $2 \times (\pi - \alpha)$ tokens, where $(\pi - \alpha)$ tokens are the extra tokens created, and the other $(\pi - \alpha)$ tokens are reserved for live-page copying. The system will have the maximum number of tokens accumulated when all real-time tasks T_i and their corresponding real-time garbage collectors G_i behave in the same way as described above, and all of the meta-periods end at the same time point. Figure 6 illustrates the above example based on the task set in Figure 3. The largest potential number of tokens in system $\rho_{\max}$ could be as follows, where ρ_{free} is the number of unallocated tokens

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

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

$$\rho_{\text{free}} + 2\pi + \sum_{i=1}^{n} \left(\frac{w_{T_i} \times \sigma_i}{p_{T_i}} + \frac{\alpha \times \sigma_i}{p_{G_i}} + 2(\pi - \alpha) \right) < \left(1 - \frac{(\alpha - 1)}{\pi} \right) \times \Theta - \Lambda - \alpha + 1 \tag{8}$$

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, that is, the utilization of the flash Λ. We shall address the admission control of real-time tasks in the next section, based on α.

4.2 Initial System Configuration

In the previous section, we derived a sufficient condition to verify whether the promised block-recycling policy performance (i.e., α) could be delivered. However, the setting of α highly depends on the physical parameters of the selected flash memory and the target capacity utilization. In this section, we shall provide guidelines for the initial system configuration of the proposed flash-memory storage systems: the values of α and ρ_{init} (i.e., the number of initially available tokens).

According to equation (4), the value of α could be set in-between $(0, \lceil \pi \times \frac{\Delta}{\Theta} \rceil]$. The objective is to set α as large as possible, because a large value for α means a better block-recycling performance. With a better block-recycling performance, the system could accept tasks that are highly demanding on page writes. For example, if the capacity utilization is 50%, we could set the value of α as $\lceil 32 \times 0.5 \rceil = 16$.

ρ_{init} could be set as follows: A large value for ρ_{init} could help the system in accepting more tasks because tokens must be provided for real-time tasks and real-time garbage collectors on their arrivals. Equation (6) in Section 4.1 suggests an upper bound of ρ, which is the total number of tokens in the current system. Obviously, $\rho = \rho_{\text{init}}$ when there is no task in system, and ρ_{init} must satisfy equation 6. We have shown that $\rho_{\max}$ (which is derived from Equation (7)) must satisfy equation (6) because the value of ρ might grow up to $\rho_{\max}$. It becomes interesting if we could find a relationship between ρ_{init} and $\rho_{\max}$, so that we could derive an upper bound for ρ_{init} based on equation (6). First, equation (7) could be rewritten as follows:

$$\rho_{\max} = \rho_{\text{free}} + 2\pi + \sum_{i=1}^{n} \left(\left(\frac{w_{T_i} \times \sigma_i}{p_{T_i}} + (\pi - \alpha) \right) + \left(\frac{\alpha \times \sigma_i}{p_{G_i}} + (\pi - \alpha) \right) \right). \quad (9)$$

Obviously the largest possible value of $\rho_{\max}$ occurs when all available tokens are given to real-time tasks and real-time garbage collectors. In other words, we only need to consider when $\rho_{\text{free}} = 0$. Additionally, we have $\rho_{T_i} = (\frac{w_{T_i} \times \sigma_i}{p_{T_i}})$, $\rho_{G_i} = (\pi - \alpha)$, and $(\frac{\alpha \times \sigma_i}{p_{G_i}}) \geq (\frac{w_{T_i} \times \sigma_i}{p_{T_i}})$ by the creation rules in equation (1). Therefore, equation (9) can be further rewritten as follows:

$$\rho_{\max} \geq 2\pi + 2 \sum_{i=1}^{n} (\rho_{T_i} + \rho_{G_i}). \quad (10)$$

Because we let $\rho_{\text{free}} = 0$, we have $\rho_{\text{init}} = \sum_{i=1}^{n} (\rho_{T_i} + \rho_{G_i})$. The relationship between $\rho_{\max}$ and ρ_{init} could be represented as follows:

$$\rho_{\max} \geq 2\pi + 2 \times \rho_{\text{init}}. \quad (11)$$

This equation reveals the relationship between $\rho_{\max}$ and ρ_{init}, and the equation is irrelevant to the characteristics (e.g., their periods) of real-time tasks and real-time garbage collectors in system. In other words, we could bound the value of $\rho_{\max}$ by managing the value of ρ_{init}. When we consider both equation (6) and (11), we have the following equation:

$$\rho_{\text{init}} < \frac{\left(\left(1 - \frac{(\alpha-1)}{\pi}\right) \times \Theta - \Lambda - \alpha + 1 - 2\pi\right)}{2}. \tag{12}$$

Equation (12) suggests an upper bound for ρ_{init}. If ρ_{init} is always under the bound, then the total number of tokens in system will be bounded by equation (6) (because $\rho_{\max}$ is also bounded). As a result, we do not need to verify equation (8) for every arrival of any new real-time tasks and real-time garbage collectors. We only need to ensure that equation (12) holds when the system starts. That further simplifies the admission control procedure to be proposed in the next section.

4.3 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 provide the procedures for admission control:

Given a set of real-time tasks $\{T_1, T_2, \ldots, T_n\}$ and their corresponding real-time garbage collectors $\{G_1, G_2, \ldots, G_n\}$, 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).

Because 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:

$$\sum_{i=1}^{n} \left(\frac{w_{T_i} \times \sigma_i}{p_{T_i}} + (\pi - \alpha) \right) \leq \rho_{\text{free}}. \tag{13}$$

Note that $\rho_{\text{free}} = \rho_{\text{init}}$ when the system starts (and there are no tasks in system), where ρ_{init} is the number of initially reserved tokens. The evaluation of the above test could be done in a linear time. Beside the test, engineers are supposed to verify the relationship between α and Λ by means of equation (8), which serves as a sufficient condition to guarantee α. However, in the previous section, we have shown that if the value of ρ_{init} is initially configured to satisfy equation (12), the relation in equation (8) holds automatically. As a result, it is not necessary to verify equation (8) at each arrival of any real-time task.

Other than the above tests, engineers should verify the schedulability of real-time tasks in terms of CPU utilization. Suppose that the earliest deadline first algorithm (EDF) [Liu and Layland 1973] is adopted to schedule all real-time tasks and their corresponding real-time garbage collectors. Because 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 other system overheads could be ignorable:

$$\frac{t_e}{min\_p()} + \sum_{i=1}^{n} \left(\frac{c_{T_i}}{p_{T_i}} + \frac{c_{G_i}}{p_{G_i}} \right) \leq 1 \tag{14}$$

where $min\_p()$ denotes the minimum period of all T_i and G_i, and t_e is the duration of a block erase. Equation (14) is correct because Liu and Layland [1973] show that all independent tasks scheduled by EDF are schedulable if their total CPU utilization (i.e., $\sum_{i=1}^{n}(\frac{c_{T_i}}{p_{T_i}} + \frac{c_{G_i}}{p_{G_i}})$) is not over 100%, and $\frac{t_e}{min\_p()}$ denotes the ratio of CPU utilization sacrificed due to the blocking of non-preemptive operations [Baker 1990]. It is because a real-time task might be blocked by a block erase in the worst case. The maximum blocking factor is contributed by the task which has the shortest period, i.e., $\frac{t_e}{min\_p()}$. Equation (14) could be derived in a similar way as that for Theorem 2 in Baker [1990]. We must emphasize that the above formula only intends to deliver the idea of blocking time, due to flash-memory operations.

Note that there exists a relationship between the utilization of flash memory and the minimum number of free pages obtained for each recycle of a block. If we want to guarantee the minimum number of available pages recovered from the recycle of a block, then we could set a bound on the utilization of flash memory. The minimum number of available pages recovered from the recycle of a block could have an impact on the admission control of real-time tasks (i.e., equations (13) and (14)) because a smaller number of available pages obtained for each block recycling would result in a larger number of block recycles in order to obtain enough free pages for tasks.

4.4 The Performance Justification of the Free-Page Replenishment Mechanism

The purpose of this section is to justify that all real-time tasks always have enough tokens to run under the free-page replenishment mechanism, and no real-time garbage collector will be blocked because of an insufficient number of free pages.

Because each real-time task T_i consumes no more than $(w_{T_i} \times \sigma_i / p_{T_i})$ tokens in a meta-period, the tokens initially reserved are enough for its first meta-period. Because the corresponding real-time garbage collector G_i replenishes T_i with $(\alpha \times \sigma_i / p_{G_i})$ tokens in each subsequent meta-period, the number of replenished tokens is larger than or equal to the number that T_i needs, according to equation (1). We conclude that all real-time tasks always have enough tokens to run under the free-page replenishment mechanism.

Because real-time garbage collectors also need to consume tokens for live-page copying, we shall justify that no real-time garbage collector will be blocked forever because of an insufficient number of free pages. Initially, $(\pi - \alpha)$ tokens are given to each real-time garbage collector G_i. It is enough for the first block recycling of G_i. Suppose that G_i decides to erase a block that has x dead pages (note that $x \geq \alpha$). The rest $(\pi - x)$ pages in the block could be either live pages and/or free pages. For each nondead page in the block, G_i might need to consume a token to handle the page, regardless of whether the page is live or free. After

Table III. The Basic Simulation Workload

Task	c^{cpu} (ms)	Page Reads	Page Writes (w)	c μs	p (ms)	c/p
T_1	3	4	2	6354	20	0.3177
T_2	5	2	5	8738	200	0.0437

the copying of live pages, a block erase is executed to wipe out all pages on the block, and π tokens (free pages) are created. The π created tokens could be used by G_i as follows: G_i replenishes itself with $(\pi - x)$ tokens, replenishes T_i with α tokens, and finally gives up the residual tokens. Because $x \geq \alpha$, we have $(\pi - x) + \alpha \leq \pi$ so G_i will not be blocked because of an insufficient number of free pages.

5. PERFORMANCE EVALUATION

5.1 Simulation Workloads

5.1.1 *Overview.* The real-time garbage-collection mechanism proposed in Section 3 provides a deterministic performance guarantee to real-time tasks. The mechanism also targets at simultaneous services to both real-time and non-real-time tasks. The purpose of this section is to show the usefulness and effectiveness of the proposed mechanism. We compared the behaviors of a system prototype with or without the proposed real-time garbage-collection mechanism. We also show the effectiveness of the proposed mechanism in the wear-leveling service, which is executed as a non-real-time service.

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, and all files were stored 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 control files, did some computations, and updated their own (small) status files. The non-real-time task emulated a file downloader, which downloaded the 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 16 MB NAND flash [NAND]. The block size was 16 KB, and the page size was 512 B ($\pi = 32$). 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 III. We varied the capacity utilization of flash memory in some experiments to observe the system behavior under different capacity utilizations. For example, a 50% capacity utilization stand for the case that one half of the total pages were live pages. An 8 MB block device was emulated by a 16 MB flash memory to generate a 50% capacity utilization. Before each experiment started, the emulated block device was written so that most of pages on the flash memory were live pages. The duration of each experiment was 20 min.

5.1.2 *System Configuration and Admission Control for Simulation Experiments.* In this section, we used the basic simulation workload to explain how to configure the system and how to perform the admission control.

Table IV. The Simulation Workload for Evaluating the Real-Time Garbage-Collection Mechanism ($\alpha = 16$, capacity utilization = 50%)

Task	c^{cpu}	Page Reads	Page Writes (w)	c	p	c/p
T_1	3 ms	4	2	623 μs	20 ms	0.3115
T_2	5 ms	2	5	10291 μs	200 ms	0.0514
G_1	10 μs	16	16	20282 μs	160 ms	0.1267
G_2	10 μs	16	16	20282 μs	600 ms	0.0338

In order to configure the system, system engineers need to first decide the expected block-recycling performance α. According to equation (4), the value of α could be set as a value between $(0, \lceil \pi \times \frac{\Delta}{\Theta} \rceil]$. Under a 50% capacity utilization, the value of α could be set as $\lceil 32 \times 0.5 \rceil = 16$ because a large value is preferred. Secondly, we use equation (12) to configure the number of tokens initially available in system (i.e., ρ_{init}). In this example, we have $\alpha = 16$, $\pi = 32$, $\Lambda = 16{,}384$, and $\Theta = 32{,}768$. As a result, we have $\rho_{\text{init}} < 472.5$. Intuitively, ρ_{init} should be set as large as possible because we must give tokens to tasks at their arrival, and a larger value of ρ_{init} might help system to accept more tasks. However, if we know the token requirements of all tasks in a priori, a smaller ρ_{init} could help in reducing the possibility to recycle a block that still have free pages inside. In this example, we set $\rho_{\text{init}} = 256$, which is enough for the basic simulation workload. Once α and ρ_{init} are determined, the corresponding real-time garbage collectors G_1 and G_2 could be created under the creation rules (equation (1)). The CPU requirements, the flash memory requirements, and the periods of G_1 and G_2 were shown in Table IV.

The admission control procedure is as follows: First, we verify if the system has enough tokens for the real-time garbage-collection mechanism under equation (13): The token requirements of T_1, T_2, G_1, G_2, and the non-real-time garbage collection are 16, 15, 16, 16, and 32, respectively. Because $\rho_{\text{init}} = 256$, the verification can be verified by the formula $(15 + 16 + 16 + 16 + 32) < 256$ (equation (13)). Because the system satisfies equation (12), it implies the condition in equation (8) also holds, and we need not to check it again. Second, the CPU utilization is verified by equation (14): The CPU utilization of each real-time task is shown in Table IV, and the verification is verified by the formula $(0.3115 + 0.0514 + 0.1267 + 0.0338 + \frac{1.881}{20}) = 0.6176 \leq 1$ (equation (14)).

5.2 The Evaluation of the Real-Time Garbage-Collection Mechanism

This section is meant to evaluate the proposed real-time garbage-collection mechanism. We shall show that a non-real-time garbage-collection mechanism might impose an unpredictable blocking time on time-critical applications, and the proposed real-time garbage-collection mechanism could prevent real-time tasks from being blocked due to an insufficient number of tokens.

We evaluated a system that adopted a non-real-time garbage-collection mechanism [Kawaguchi et al. 1995] under the basic workload and a 50% flash-memory capacity utilization. We compared our real-time garbage-collection mechanism with the well-known *cost–benefit* block-recycling policy [Kawaguchi et al. 1995] in the non-real-time garbage-collection mechanism.

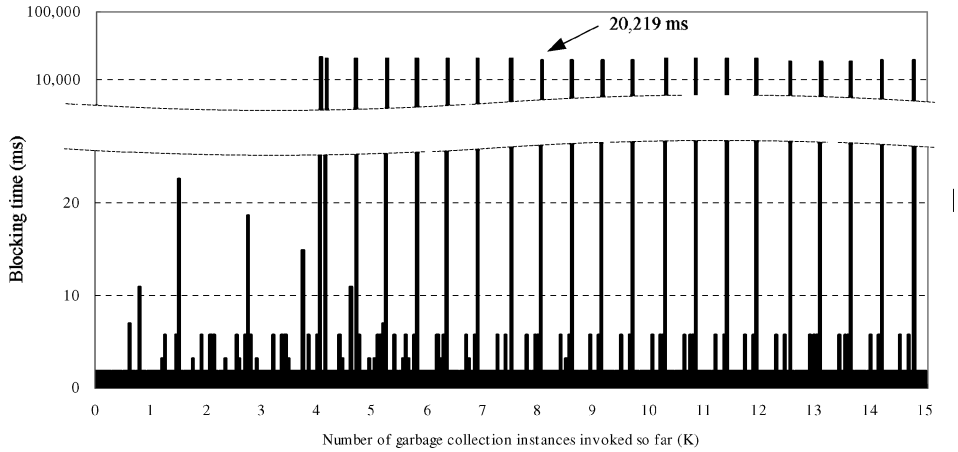

Fig. 7. The blocking time imposed on each page write by a non-real-time garbage-collection instance.

The non-real-time garbage-collection mechanism was configured as follows: A page write could only be processed if the total number of free pages on flash was more than 100. Note that any number larger than 32, that is, the number of pages in a block of the system prototype (as described in Section 3.4.1), was acceptable. 100 was chosen because it was reasonable for the experiments. If there were not enough free pages, garbage collection was activated to reclaim free pages. The cost–benefit block-recycling policy picked up a block which had the largest value of $a \times \frac{1-u}{2u}$, as summarized in Section 1. But in order to perform wear leveling, the block-recycling policy might recycle a block which had an erase count less than the average erase count by 2, regardless how many free pages could be reclaimed. Because garbage collection was activated on demand, the garbage-collection activities directly imposed blocking time on page writes. 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. An instance of garbage collection might consist of recycling several blocks consecutively because the non-real-time garbage-collection policy might recycle a block without dead pages, due to wear leveling.

We measured the blocking time imposed on a page write by each instance of garbage collection, because a non-real-time garbage-collection instance blocked the whole system until the garbage-collection instance finished. Figure 7 shows the blocking times of page writes, where the X-axis denotes the number of garbage-collection instances invoked so far in a unit of 1000, and the Y-axis denotes the blocking time in ms. The results show that the non-real-time garbage collection activities indeed imposed a lengthy and unpredictable blocking time on page writes. The blocking time could even reach 21.209 s in the worst case. The lengthy blocking time was mainly caused by wear leveling. The block-recycling policy might recycle a block that had a low erase count, regardless how many dead pages were on the block. As a result, the block-recycling policy might consecutively recycle many blocks until at least one free page was reclaimed.

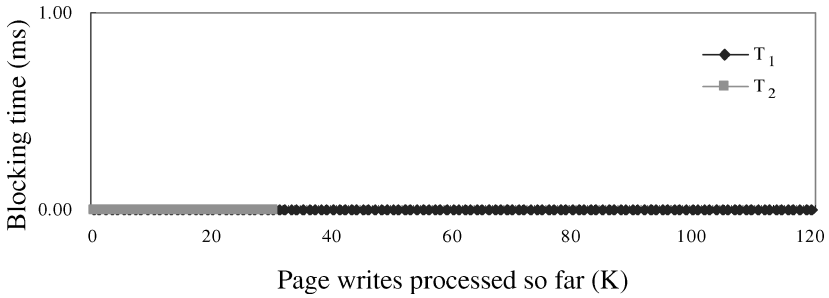

Fig. 8. The blocking time imposed on each page write of a real-time task by waiting for tokens (under the *real-time* garbage-collection mechanism).

Although the maximum blocking time could be reduced by decreasing the wear-leveler activities or performing garbage collection in an opportunistic fashion, the non-real-time garbage-collection did fail in providing a deterministic service to time-critical tasks.

The next experiment intended to observe if the proposed real-time garbage collection mechanism could prevent each real-time task from being blocked, due to an insufficient number of tokens. To compare with the non-real-time garbage-collection mechanism, a system adopted the real-time garbage-collection mechanism was evaluated under the same basic configurations. Additionally, according to the basic workload, the real-time garbage-collection mechanism created two (corresponding) real-time garbage collectors and a non-real-time wear leveler. In particular, the non-real-time wear leveler performed live-page copyings whenever a block had an erase count less than the average erase count by 2. The wear-leveler slept for 50 ms between every two consecutive live-page copyings. The workload is summarized in Table IV. Note that the garbage-collection activities (real-time and non-real-time) did not directly block a real-time task's page write request. A page write request of a real-time task would be blocked only if its corresponding real-time garbage collector did not supply it with enough tokens. Distinct from the non-real-time garbage collection experiment, we measured the blocking time of each real-time task's page write request to observe if any blocking time ever existed. The results are shown in Figure 8, where the X-axis denotes the number of page writes processed so far in a unit of 1000, and the Y-axis denotes the blocking time imposed on each page write (of real-time tasks) by waiting for tokens. Note that T_1 and T_2 generated 12,000 and 3000 page writes, respectively, in the 20-min experiment. Figure 8 shows that the proposed real-time garbage-collection mechanism successfully prevented T_1 and T_2 from waiting for tokens, as expected.

5.3 Effectiveness of the Wear-Leveling Method

The second part of experiments evaluated the effectiveness of the proposed wear-leveling policy. A good wear-leveling policy should keep an even distribution of erase cycle counts for blocks. It was because when some blocks of flash memory was worn out, flash memory would start to malfunction. The effectiveness of a wear-leveling policy could be evaluated in terms of the standard

Table V. System Parameters Under Different Capacity Utilizations

	50%	55%	60%	65%	70%	75%
ρ_{init}	256	256	256	256	256	256
α	16	15	13	12	10	8
ρ_{T_1}	16	14	12	12	10	8
ρ_{T_2}	15	15	10	10	10	5
ρ_{G_1}	16	17	19	20	22	24
ρ_{G_2}	16	17	19	20	22	24
ρ_{nr}	32	32	32	32	32	32
ρ_{max}	320	322	326	328	332	336

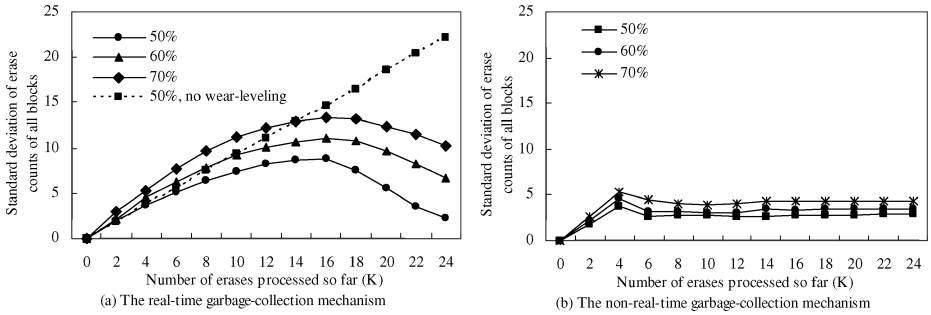

Fig. 9. Effectiveness of the wear-leveling method.

deviation of the erase counts of all blocks and the earliest appearance time of worn-out blocks.

The non-real-time wear leveler was configured as follows: The non-real-time wear-leveler performed live-page copyings whenever a block had an erase count less than the average erase count by 2. The wear-leveler slept for 50 ms between every two consecutive live-page copyings. As the past work showed that the overheads of garbage collection highly depend on the flash-memory capacity utilization [Kawaguchi et al. 1995; Wu and Zwaenepoel 1994; Chiang et al. 1997; Douglis et al. 1994], we evaluated the experiments under different capacity utilizations: 50%, 60% and 70%. System parameters under each capacity utilization are summarized in Table V. Additionally, we disabled the wear leveler to observe the usage of blocks without wear leveling. We also compared the effectiveness of the proposed wear-leveling method with the non-real-time garbage-collection mechanism.

The results are shown in Figure 9, where the *X*-axis denotes that the number of erases had been performed so far in a unit of 1000, and the *Y*-axis denotes the value of the standard deviation. Figure 9(a) and 9(b) show the effectiveness of the proposed wear-leveling method and the wear-leveling method in the non-real-time garbage-collection mechanism. In Figure 9(a), the proposed wear-leveling method gradually leveled the erase count of each block. The results also pointed out that the effectiveness of wear-leveling was better when the capacity utilization was low because the system was not heavily loaded by the overheads of garbage collection. When the system was heavily loaded because of the

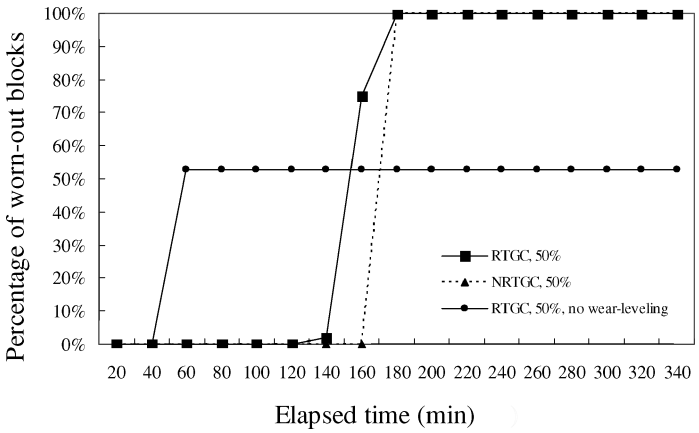

Fig. 10. The percentage of worn-out blocks.

frequent activities of garbage collection, the real-time tasks and the real-time garbage collectors would utilize most of the available bandwidth, so that the effectiveness of the non-real-time wear leveling degraded accordingly. The experiment that disabled the wear leveler was evaluated under a 50% capacity utilization. The results showed that the standard deviation increased very rapidly: Some blocks were constantly erased while some blocks were not ever erased. The frequently erased blocks would be worn-out very quickly, thus the overall lifetime of flash memory was severely reduced. In Figure 9(b), we observed that the standard deviation under the non-real-time garbage collection mechanism was stringently controlled, since the non-real-time garbage collection mechanism performed wear-leveling without the consideration of timing constraint.

Figure 10 shows the percentage of worn-out blocks in the experiments. We set the limit on wear leveling as 200 in the experiments to compare best-effort and real-time memory management methods. We did not use one million as the wear-leveling limit because it could take years to complete the experiments. Based on our experiments, 200 seemed a reasonable number for the wear-leveling limit because similar results were observed for larger numbers. The X-axis of Figure 10 denotes the amount of time elapsed in the experiments, and the Y-axis denotes the percentage of blocks that reached the wear-leveling limit. It was observed that over 50% of blocks quickly reached the wear-leveling limit under the real-time garbage-collection method without a wear-leveling policy. However, when a wear-leveling policy was adopted, the performance of the real-time garbage collection method on wear-leveling control was close to that of the best-effort method (labeled as NRTGC). Note that a good wear-leveling policy tends to keep an even distribution of erase cycle counts for blocks. When some blocks of flash memory was worn out, flash memory would start to malfunction.

5.4 Remark: Overheads

A good real-time garbage-collection mechanism should deliver a reasonably good performance without significant overheads. This section is to compare the overheads of the proposed mechanism with that of the typical non-real-time

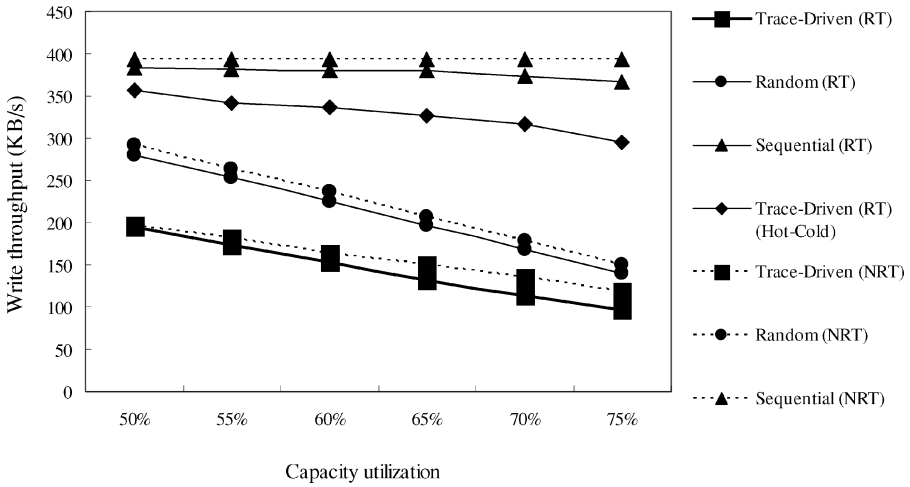

Fig. 11. The overall write throughput under different access patterns and capacity utilizations.

approach (described in previous sections) in terms of their performance degradation (under garbage collection).

The performance metric was the overall write throughput, which was contributed by the real-time tasks and the non-real-time task. The basic simulation workload in Table III was used in this part of experiments. We shall not consider the writes generated internally by garbage collection and wear leveling in the performance metric because they were considered as overheads, instead of parts of the performance index.

In this part of experiments, we varied the capacity utilization from 50% to 75%. Additionally, we evaluated two more access patterns: sequential access patterns and random access patterns. The flash-memory block device was sequentially (randomly) accessed by the real-time tasks T_1 and T_2 and the non-real-time task write under the sequential access patterns (the random access patterns). The results are shown in Figure 11, where the X-axis denotes the capacity utilization of the flash memory, and the Y-axis denotes the overall write throughput in KB/s. Based on the results shown in Figure 11, we could see that the proposed mechanism delivered a comparable performance with respect to the typical non-real-time approach. In other words, the overheads of the proposed real-time garbage-collection mechanism was affordable. Moreover, we could have two observations: First, a high capacity utilization would impose a significant performance degradation on the write throughput. Second, the system performed better under the sequential access patterns, because the live data were sequentially updated and invalidated.

As astute readers may notice, the system under the trace-driven access patterns had the worst performance in all experiments although people usually considered a random access pattern as the worst access patterns. Such anomaly was also observed in Kawaguchi et al. [1995] and Chiang et al. [1997]. Figure 11(a) shows that the performance of the system under the trace-driven access patterns was significantly boosted if hot-data and cold-data were

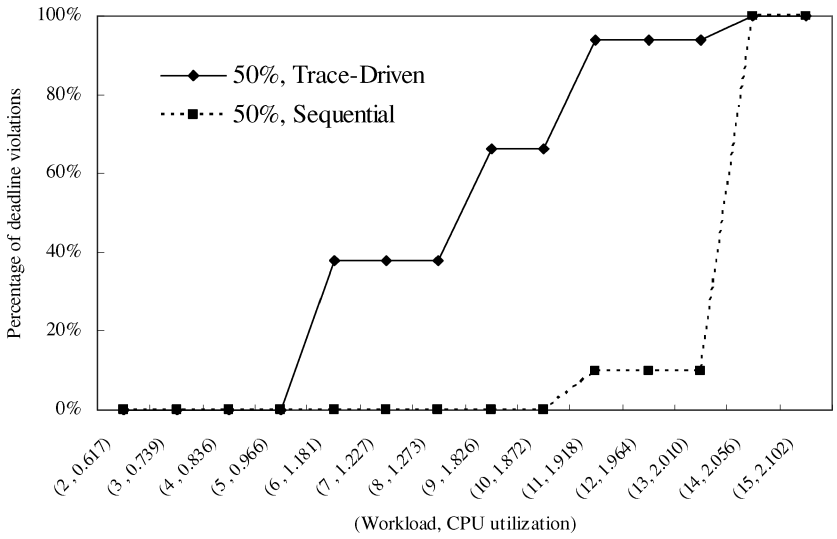

Fig. 12. The percentage of deadline violations under different overloaded workloads.

properly identified and separated. For a better explanation of hot–cold identification and separation mechanisms, we refer interested readers to Kawaguchi et al. [1995] and Chiang et al. [1997].

Figure 12 shows the percentages of deadline violations when the system was overloaded. It was to evaluate how conservative the proposed admission control policy was. Note that the experiments did not reject any workload because we turned off admission control in this part of experiments. (Otherwise, the proposed admission control policy should start to reject additional workloads when the number of writes was larger than 5. That was when the CPU utilization was over 100%.) The number of page writes issued by T_1 in each period was increased from 2 (the original setup) to 15 with a 50% capacity utilization of flash memory. The X-axis of Figure 12 denotes the number of writes per period and the corresponding CPU utilizations, and the Y-axis denotes the percentage of deadline violations under different workloads. It was observed that the proposed admission control was good under trace-driven access patterns because deadline violations were observed right after admission control failed. Note that trace-driven access patterns introduced a lot of live data copyings because trace-driven access patterns resulted in a smaller probability in recycling blocks with a less number of live pages. When sequential access patterns were experimented, admission control seemed very conservative. No deadline violations were observed until the total CPU utilization approached 187.2%. It was because sequential access had very light overheads on live data copying.

6. CONCLUSION

This paper is motivated by the needs of a predictable block-recycling policy to provide real-time performance to many time-critical systems. We propose

a real-time garbage-collection methodology with a guaranteed performance. The endurance problem is also resolved by the proposing of a wear-leveling method. Because of the real-time garbage-collection support, each time-critical task could be guaranteed with a specified number of reads and/or writes within its period. Non-real-time tasks are also serviced with the objective to fully utilize the potential bandwidth of the flash-memory storage system. The design of the proposed mechanism is independent of the implementation of flash-memory-management software and the adoption of real-time scheduling algorithms. We demonstrate the performance of the proposed methodology in terms of a system prototype.

There are several promising directions for future research. We shall extend the concept of the real-time garbage collection to the (disk) log-structured file systems (LFS) [Rosenblum and Ousterhout 1992]. LFS perform out-of-place update like a flash-memory storage system. Note that time-critical applications might wish to use LFS because of its reliability in data consistency and its capability in rapid error recovering. We shall also extend the research results of this paper to power management for portable devices. The goal is to combine the dynamic-voltage-adjustment technique [Chang et al. 2001] with the real-time garbage-collection mechanism to reduce the energy dissipation of a flash-memory storage system. We believe that more research in these directions may prove being very rewarded.

REFERENCES

BAKER, T. P. 1990. A stack-based resource allocation policy for real-time process. In *Proceedings of the 11th IEEE Real-Time System Symposium.*

CHANG, L. P. AND KUO, T. W. 2002. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the The 8th IEEE Real-Time and Embedded Technology and Applications Symposium.*

CHANG, L. P. AND KUO, T. W. 2004. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the ACM Symposium on Applied Computing.*

CHANG, L. P., KUO, T. W., AND LO, S. W. 2001. A dynamic-voltage-adjustment mechanism in reducing the power comsumption in flash memory storage systems for portable devices. In *Proceedings of the IEEE International Conference on Consumer Electronics.*

CHIANG, M. L., PAUL, C. H., AND CHANG, R. C. 1997. Manage flash memory in personal communicate devices. In *Proceedings of IEEE International Symposium on Consumer Electronics.*

DOUGLIS, F., CACERES, R., KAASHOEK, F., LI, K., MARSH, B., AND TAUBER, J. 1994. Storage alternatives for mobile computers. In *Proceedings of the USENIX Operating System Design and Implementation.*

Journaling Flash File System (JFFS). http://sources.redhat.com/jffs2/jffs2-html/.

*K9f2808u0b 16mb*8 NAND Flash Memory Data Sheet*. Samsung Electronics Company.

KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. 1995. A flash memory based file system. In *Proceedings of the USENIX Technical Conference.*

KIM, H. J. AND LEE, S. G. 1999. A new flash memory management for flash storage system. In *Proceedings of the Computer Software and Applications Conference.*

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM 7*, 3 (July).

MALIK, V. 2001a. *JFFS—A Practical Guide*. Available at http://www.embeddedlinuxworks.com/articles/jffs_guide.html.

MALIK, V. 2001b. JFFS2 is broken. In *Mailing List of Memory Technology Device (MTD) Subsystem for Linux.*

MOLANO, A., RAJKUMAR, R., AND JUVVA, K. 1998. Dynamic disk bandwidth management and metadata pre-fetching in a real-time filesystem. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*.

WU, C. H., CHANG, L. P., AND KUO, T. W. 2003. An efficient r-tree implementation over flash-memory storage systems. In *Proceedings of the ACM 11th International Symposium on Advances on Geographic Information Systems*.

WU, M. AND ZWAENEPOEL, W. 1994. envy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Yet Another Flash-Filing System (YAFFS). Aleph One Company.

Received May 2003; accepted February 2004