FLASH MEMORY GARBAGE COLLECTION IN HARD REAL-TIME SYSTEMS

A Thesis

by

CHIEN-AN LAI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2011

Major Subject: Computer Science

FLASH MEMORY GARBAGE COLLECTION IN HARD REAL-TIME SYSTEMS

A Thesis

by

CHIEN-AN LAI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2011

Major Subject: Computer Science

ABSTRACT

Flash Memory Garbage Collection in Hard Real-Time Systems. (August 2011)

Chien-An Lai, B.S., National Taiwan University

Chair of Advisory Committee: Dr. Riccardo Bettati

Due to advances in capacity, speed, and economics, NAND-based flash memory technology is increasingly integrated into all types of computing systems, ranging from enterprise servers to embedded devices. However, due to its unpredictable update behavior and time consuming garbage collection mechanism, NAND-based flash memory is difficult to integrate into hard-real-time embedded systems. In this thesis, I propose a performance model for flash memory garbage collection that can be used in conjunction with a number of different garbage collection strategies. I describe how to model the cost of reactive (lazy) garbage collection and compare it to that of more proactive schemes. I develop formulas to assess the schedulability of hard real-time periodic task sets under simplified memory consumption models. Results show that I prove the proactive schemes achieve the larger maximum schedulable utilization than the traditional garbage collection mechanism for hard real-time systems in flash memory.

To my parents, Li-Shu and Chin-Hsing, you are my strong foundation

ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. Riccardo Bettati. He continuously and tirelessly guided me through the years and I deeply appreciate all his guidance, care, and sacrifice. He taught me the thrill of pursuing the unknown.

I would also like to thank my committee members, Dr. Anxiao (Andrew) Jiang and Dr. Narasimha Reddy, for their support and feedback. I am grateful for the time they committed to helping me grow as a researcher.

I also thank the many collaborators I've enjoyed the opportunity to work with these years, both on this work and on other research projects: Dr. Youngwoo Ahn, Timothy G. Nix, and Hutson Betts. I am grateful for working with them through most of my graduate career. As we've shared an office together and collaborated on various research endeavors, we've also celebrated in each other's victories and helped each other through difficult challenges. We have accomplished much by working together.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Due to advances in capacity, speed, and economics, NAND-based flash memory technology is increasingly been integrated into all types of computing systems, ranging from enterprise servers [1, 2] to embedded devices [3]. The advantages of NAND-based flash memory versus traditional storage technologies lie in its speed and low power consumption, and the lack of mechanical components. The latter in turn leads to benefit in packaging (small size) and environmental parameters (e.g. shock resistance). In particular the benefits in power and packaging have made flash memory very popular in embedded systems.

Unfortunately, the asymmetric performance characteristics of flash memory (in particular its slow *erase* operation) make it difficult to integrate into time-critical and real-time systems: while *read* and *write* operations can be supported at relatively high speed (350 μs and 920 μs per 512 bytes, respectively) [4], *overwrite* operations require a *block-wide erase*, which is significantly more time consuming (typically in the order of 2 ms). In order to amortize the cost of such operations, flash memory designers use out-place update and bulk-erase. For flash memory, the *page*, usually 512 bytes, is the smallest unit for reading and writing. Pages in turn are arranged in *blocks* (typically a block contains 64 pages). Once a page is written, it can not be updated directly. During an *update* request, the original page is labeled as *invalid*,

The journal model is *IEEE Transactions on Automatic Control.*

and the update data is written to the new empty pages.

As update requests are processed, empty pages are consumed. New empty pages are generated by *recycling* invalid pages. This is achieved by erasing the content of the invalid pages and by adding these pages to the set of empty pages. This process is complicated in flash memory because erase operations can be performed *at block level only*. Since any block may contain valid pages in addition to the invalid pages to be recycled, the recycling process must first copy the content of remaining valid pages to empty pages elsewhere before erasing the entire block. Once the block is erased, its pages are added to the available empty pages and become available for subsequent write and update operations. The cost to do so depends on the number of such valid pages. Therefore, the timing behavior of real-time tasks depends critically on the order of flash memory operations, thus making the completion times of such tasks highly unpredictable.

The process of recycling invalid pages is *de-facto* a garbage collection operation [5, 6, 7]. Typically, garbage collection of flash memory is triggered when the flash memory capacity reaches a low watermark, and it recycles one block at a time [5, 8]. This form of garbage collection (which we call *reactive garbage collection*) naturally causes problems for real-time systems, as time-critical tasks may get blocked at unpredictable points in time for unpredictable lengths of time due to garbage collection. From a schedulability point of view, flash memory garbage collection poses two problems: First, the worst-case blocking time caused by a single garbage collection round must be represented and bounded. Second, tasks experience what we call ”delayed-effect priority inversion”, where the processing of low-priority tasks consumes memory and so can lead to resource starvation and of blocking of high-priority tasks *some time later* in their execution. Examples of such delayed effects occur in many other resource-constraint settings, such as caches [9] or garbage collected memory [10], or in thermally constrained systems [11].

In many cases this form of priority inversion can be addressed by appropriate partitioning of shared resources (see [9] for the case of cache partitioning). This is not possible in the case of thermal constraints or flash memory, however. In the case of thermal control it is naturally impossible to partition the available thermal budget. In the case of flash memory, partitioning is rendered impossible by the fact that the flash memory translation layer (FTL) [12] is typically not accessible from outside of the memory.

In this thesis we first develop a performance model for flash-memory garbage collection in order to understand the blocking times due to garbage collection. Since the effectiveness of flash memory garbage collection is highly dependent on the number of empty pages (the available capacity of the memory,) several trade-offs exist in the design of garbage collection schemes. We develop a garbage collection effectiveness model, which we then apply to a number of garbage collection schemes to assess their effect on the overall schedulability of the system: We first study *reactive* garbage collection and determine a schedulability bound for this scheme. We then proceed to analyze two *proactive* schemes: (i) We develop schedulability bounds for a proactive scheme that performs garbage collection during idle intervals as well as when the system runs out of pages. (ii) We then describe the effect of an allocated-bandwidth garbage collector, which uses a portion of the CPU utilization to proactively perform garbage collection at all times (in addition to blocking all tasks to perform garbage collection wherever the system runs out of pages).

This thesis is organized as follows: In Chapter II we provide the background for our work. We will describe the operation of flash memory, provide details on the flash memory translation layer and describe related work in the area of flash memory management. Moreover, we review the previous research about garbage collection in real-time system. In Chapter III we will describe a simplified flash memory model that will be used throughout the rest of the thesis. In Chapter IV, we describe the problem of scheduling real-time tasks with flash memory garbage collection. We describe the workload model and provide a simplified memory consumption model. In Chapter V, we analyze the equilibrium state and schedulable utilization for the simple case of identical-period real-time tasks with flash memory. In Chapter VI, we illustrate the constant-bandwidth garbage collection and explain how to calculate the equilibrium speed as a function of the capacity level. In Chapter VII, we show the performance evaluation and demonstrate that the proactive garbage collection, which we propose in this thesis, can achieve higher schedulable utilization comparing to the traditional reactive garbage collection. Chapter VIII is the conclusion.

CHAPTER II

PRELIMINARIES AND RELATED WORK

A. Flash Memory Architecture

SLC (Single Level Cell) and MLC (Multiple Level Cell) flash memory are two major NAND flash memory designs. In SLC NAND flash memory, one cell contains one bit of information, while the same cell could contain n bits in MLC flash memory. The latter is usually denoted by $MLC_{\times n}$. The trade-off between SLC and MLC is not a simple one. For example, while $MLC_{\times n}$ NAND flash memory has the advantage of information density compared to SLC NAND flash memory, pages in $MLC_{\times n}$ flash memory can only be written sequentially in one block, while pages in SLC flash memory can be written randomly across the block [13]. MLC NAND flash memory is usually cheaper than SLC NAND flash memory. Throughout this thesis, we will use SLC since it has better performance, endurance and reliability.

The system architecture of flash memory storage systems consist of two primary layers: The *Memory Technology Device (MTD) layer*, which supports *read* and *write* functions of the file system, and the *Flash Translation Layer* (FTL), which translates addresses between Logical Block Addresses (LBA) and Physical Block Addresses (PBA), and which is responsible for garbage collection. Besides, the FTL also addresses *wear leveling*: It distributes block erase evenly across the memory to extend the flash memory life time.

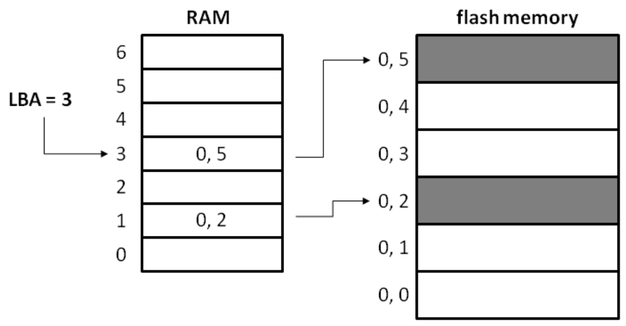


Fig. 1.: Page mapping in the FTL protocol

B. Flash Translation Layers

One of the biggest factors of flash memory performance is the design of the page allocation algorithms in the [12]. There are two main flash memory implementation protocols: (1) FTL (flash translation layer protocol) and (2) NFTL (NAND flash translation layer protocol).

In FTL [14], when a page is accessed in flash memory, the translation layer determines the exact position by looking up the LBA-to-page mapping, which is maintained in RAM. An example of FTL is shown in Figure 1. The FTL maps the page identified by LBA = 3 to Page 5 of Block 0 in flash memory.

In NFTL [15], in order to access a page, the *primary block* is checked first. If the page is labeled as "invalid", then the mapping proceeds to look up the *replacement block* to find the first valid page whose LBA is matched from the bottom to the top.

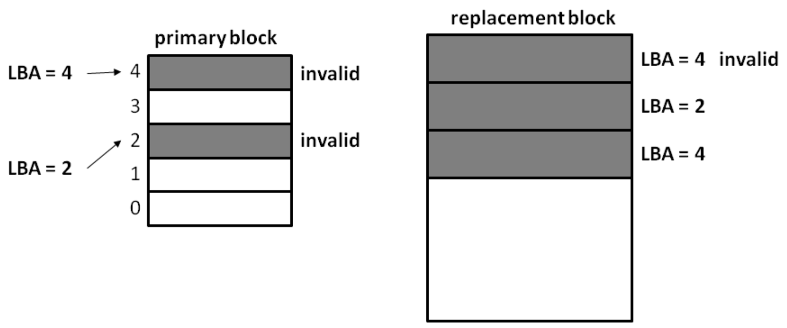


Fig. 2.: Page mapping in the NFTL protocol

If the operation is a write, the NFTL marks the corresponding page in the primary block as invalid and then writes the data in the first empty page from the top of the replacement block. Figure 2 is an example of a mapping in NFTL: The data at LBA=4 are updated, so NFTL check whether the corresponding page in primary block is empty. If not, NFTL writes data to the page on replacement block sequentially.

The flash translation layer is designed not only to implement the translation between the Logical Block Addresses (LBA) and the Physical Block Address (PBA), but also to implement garbage collection in flash memory. Because of the asymmetric performance characteristics of flash memory between writing/reading and erasing mentioned above, researchers have proposed different strategies to minimize the garbage collection overhead. For example, the authors in [7] propose the *Ef-greedy* policy, which focuses on reducing garbage collection time as well as wear-leveling. To address the problem that the original greedy policy does not consider wear-leveling, the *Ef-greedy* policy selects blocks based on a combination of page update frequency and block-erase time. Alternatively, the authors in [6] regard the least recently swapped-out page from main memory as a *hot* page, which is likely to be swapped-in main memory next as a result of the round-robin operating system scheduler. The garbage collector adopts the extended greedy policy by considering the swapped-out time of the pages. In this thesis, we use a simple greedy garbage collector as described in [16]: The garbage collector always chooses the block containing the most of invalid pages to recycle. When garbage collection is required, the garbage collector always recycles the block containing the maximum number of invalid pages.

The performance of flash memory is also affected by the access patterns of user applications. By exploiting the locality of flash memory accesses, the garbage collector can group hot pages and cold pages respectively. This can significantly reduce garbage collection overhead because blocks with hot pages have fewer valid pages that must be copied before erasing the block [15]. However, when flash memory storage systems are adopted for general-purpose applications or for systems with multiple independent tasks, memory accesses come from all of the applications running on the system. As a result, the memory locality of user applications becomes ambiguous and hard to predict. For example, the authors in [17] describe how worst-case response times can happen when pages are accessed randomly. In this thesis, we ignore memory locality, which is appropriate when we model aggregated memory access from multiple independent tasks and FTL-level permutation of pages. By assuming that write operations occur for random pages, we can evaluate the system performance to capture worst-case write response times.

Since NAND flash memory can only stand for limited erase times (usually 100000 erase times per block in SLC and 10000 erase times per block in MLC,) researchers proposed different strategies to achieve *wear leveling* i.e., to distribute block erase times evenly to improve the lifetime and reliability of flash memory. For example, Chang *et al.* [18] used static wear leveling, which moves data proactively, so that cold data will not hold block for a long time. Besides, Wang *et al.* [19] proposed dynamic wear leveling, depending on dynamic logical mapping table to decide which block should be erased, as the wear leveling policy. In this thesis, we won't concern the wear leveling problem so that we could maximize the schedulable utilization of real-time systems.

C. File Systems for Flash Memory

Modern file systems are designed for hard drives and tend to access metadata, such as file attributes, frequently in small chunks. This kind of metadata access strategy causes problems when adopted to flash memory storage systems. In hard drives, *in-site* overwrite of data is possible and has no vice effects. Comparing with hard drives, one of the distinct properties of flash memory is out-place updates: When an update request comes, the FTL allocator allocates a new page for writing the new data and labels the old data page as invalid. The out-place requirement for flash memory might quickly deteriorate the memory capacity and system performance when small-size updates for metadata are requested frequently. To conquer this problem, two native flash file systems have been proposed: Journaling Flash File System (JFFS) [20] and Yet Another Flash File System (YAFFS) [21]. JFFS is a log-structure file system [22] for NOR flash memory, and JFFS2, the second version of JFFS, would work for NAND flash memory. On the other hand, YAFFS is designed for NAND flash memory, and YAFFS2, the second version of YAFFS, would work for MLC flash memory, which requires special constraints on write operations.

D. Garbage Collection in Real-Time Systems

Due to its notorious unpredictability, garbage collection has been extensively studied in the context of real-time systems. For example, Nilsen proposed real-time garbage collection for linked data structures and string regions [23]. Schoeberl introduced a real-time garbage collector that can be executed like a normal real-time thread [24]. Pizlo *et al.* proposed a concurrent and real-time garbage collector that can guarantee time-and-space worst-case bounds [25].

Because of the required out-place updates and the significant overhead for garbage collection mechanism, garbage collection affects the real-time system performance much more in flash memory than in RAM. Chang *et al.* [5] proposed a real-time garbage collection policy that can guarantee performance for hard real-time time systems under reactive garbage collection. In this thesis, we formulate a performance model for flash memory garbage collection and use it to form the garbage collection policies to maximize the schedulable utilization of real-time tasks.

CHAPTER III

FLASH MEMORY MODEL

We model the flash memory in a system as a collection of N *blocks*, each consisting of B *pages*. Pages can be in one of the following states: *empty*, *valid*, *invalid*. Page that are marked as *empty* contain no data and can therefore be written to. A *valid* page can be read, but it must be *erased* before being written to. An *invalid* page is the result of an *update* operation. Since the update can not happen in-place, the memory controller writes the new data into an empty page and marks the old page as *invalid*.

The memory controller supports the following three types of operations:

- *Read* operation: This operation reads data from a memory page. Let t_r be the latency of a read operation.
- *Write* operation: This operation writes data into an empty page. We let t_w denote the latency of write operation.
- *Update* operation: This operation overwrites the current data in a page. Since data cannot be overwritten directly at page level, the memory controller reads the current content of the page, performs the update, and writes the updated content to an empty page. The FTL [12] remaps the user-level identifier of the page to the new page, thus rendering the migration of the page content invisible to the user. The cost t_u of an update is thus $t_r + t_w$ if an empty page is available. If no such page is available, the update cost must include the cost

to recover at least one empty page by recycling invalid pages.

A. Reclaiming Empty Pages

Empty pages to be used in write operations are reclaimed by *recycling* invalid pages as follows:

1. Identify a block, say Block b_i, that contains invalid pages.
2. Copy the valid pages in Block b_i to empty pages.
3. Erase Block b_i. Append the B newly generated empty pages to the list of empty pages.

Assuming that Block b_i contains I_i invalid pages and E_i empty pages, the cost to reclaim the I_i pages is $(B - I_i - E_i) \times (t_r + t_w) + t_e$, i.e. the cost to copy the $B - I_i - E_i$ valid pages plus the time to erase the block. For the specific I_i, the worst-case garbage collection delay happens when $E_i = 0$, which means that all of the pages in B_i except the invalid pages are valid.

In practice, the cost to reclaim pages depends greatly on the distribution of invalid pages per block. In the following, we assume that the FTL permutes pages sufficiently to eliminate task-level locality. After a start-up period we assume that write operations happen to random pages in the flash memory. It has been shown that random accesses cause the worst-case delay for update operations [17]. We assume that block information is stored in RAM and operations such as finding the block with the maximum number of invalid pages can be made with negligible overhead. Therefore, the garbage collector will pick that block with maximum number I_i of invalid pages in order to minimize the cost of the garbage-collection round.

B. Effectiveness of Flash Memory Garbage Collection

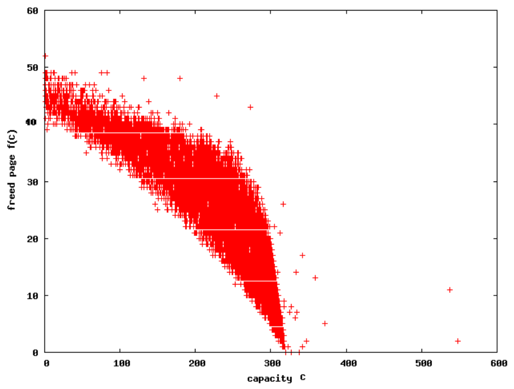


Fig. 3.: Effectiveness of garbage collection

Figure 3 shows the result of a simulation that monitors the garbage collection activity of a flash memory controller. We use *flashsim* [26], a high-fidelity flash memory simulator. The workload writes to random pages (as visible to the user, the flash translation layer maps these pages internally). In order to capture the effectiveness of the garbage collection under different conditions, we triggered garbage collection at random intervals. We ran the simulation on a simple flash memory with 10 blocks of 64 pages each. The user sees 5 blocks, with the rest of the memory used internally to support garbage collection and other activities. (This is a very small memory, but the results hold with increasing memory sizes as well.) In the following discussion we call the number of empty pages the *capacity* of the memory and we denote it by C. The number of valid pages is denoted by V, and the number of invalid pages by I. Therefore, the following relation holds:

$$C = N \times B - V - I \ . \tag{3.1}$$

Figure 3 displays the effectiveness $f(C)$ of the garbage collector in terms of number of pages ”freed” in one garbage collection round. This representative figure illustrates the following points: First, the value of $f(C)$ diminishes as the available capacity C increases. This is to be expected, as with increasing capacity the number of invalid pages in blocks diminishes, and the garbage collector has fewer invalid pages to reclaim in a single round. Second, $f(C)$ is upper-bounded by the block size and by the total number of pages that are made available to the user. Finally, $f(C)$ is lower-bounded as well. In the following, we will characterize this lower bound in terms of the flash memory parameters.

C. Lower Bound on Garbage Collection Effectiveness

Recall that we denote by C the memory capacity (in number of empty pages) at the beginning of the garbage collection. Furthermore, we denote by $f(C)$ the number of freed pages during a garbage collection round as a function of the memory capacity. We claim that $f(C)$ is lower-bounded as follows:

$$f(C) \geq -a \times C + b \ , \tag{3.2}$$

with the following discussion defining parameters a and b.

Lemma 1.

$$f(C) \geq \frac{I}{N} = \frac{(N \times B - V - C)}{N} \ .$$

Proof. A page can either be valid, invalid, or empty. By pigeon-hole principle, the minimum number of invalid pages in one block is lower-bounded by the overall minimum invalid pages divided by the number of blocks, in this case $(N \times B - V - C)/N$. □

We make the following two observations:

Observation 1. *When no empty pages are available, the minimum number of reclaimable pages is lower-bounded by* $\frac{(N \times B - V)}{N}$, *i.e.*

$$f(0) \geq \frac{(N \times B - V)}{N} \ .$$

Observation 2. *The effectiveness of the garbage collector is null if no invalid pages are available, i.e.* $f(C = N \times B - V) = 0$.

By Lemma 1 and the above two observations we can derive that even in the worst scenario, the garbage collection effectiveness is lower-bounded as follows:

$$f(C) \geq -\frac{C}{N} + \frac{N \times B - V}{N} \ . \tag{3.3}$$

Thus, the effectiveness can be lower-bounded by the function $f(C) = -aC + b$, where $a = \frac{1}{N}$ and $b = \frac{N \times B - V}{N}$.

D. Garbage Collection in the Time Domain

The time for the garbage collector to recycle one block is the sum of (1) the time to read and write the valid pages in the block and (2) the time for erasing the blocks. For convenience, we let t_{wr} denote $t_w + t_r$. The variables B, N, t_{wr} and t_e are properties of the flash memory, hence we can treat them as constants. The worst-case garbage collection delay happens when all pages except the invalid ones are valid pages in the freed block. In this case, we need to move the most of valid pages to other block. We conservatively assume that the block to be freed contains only invalid and valid pages in the following discussion. We can formulate the time t_B for recycling one block as follows:

$$t_B = (B - f(C)) \times t_{wr} + t_e \ . \tag{3.4}$$

From Equation (3.2) and (3.4), we can formulate the time t_P for recycling one page as follows:

$$\begin{aligned} t_P &= \frac{t_B}{f(C)} = \frac{(B - f(C))t_{wr} + t_e}{f(C)} \qquad (3.5)\\ &= \frac{Bt_{wr} + t_e}{f(C)} - t_{wr} = \frac{Bt_{wr} + t_e}{-aC + b} - t_{wr} \ . \qquad (3.6) \end{aligned}$$

Therefore, the total time, denoted by ΔT for garbage collection from capacity C_1 to C_2 is:

$$\Delta T = \int_{C_1}^{C_2} t_P dC = \int_{C_1}^{C_2} (\frac{Bt_{wr} + t_e}{-aC + b} - t_{wr}) dC \ . \tag{3.7}$$

This resolves to:

$$\Delta T = \frac{Bt_{wr} + t_e}{-a} \times \ln \left| \frac{b - aC_2}{b - aC_1} \right| - t_{wr} \times (C_2 - C_1) \ . \tag{3.8}$$

If we define $G = C_2 - C_1$ as the total capacity recycled by garbage collection, we can rewrite the relationship between time and recycled capacity as follows:

$$\Delta T = \frac{Bt_{wr} + t_e}{-a} \times \ln \left| \frac{b - aG - aC_1}{b - aC_1} \right| - t_{wr} \times G \ . \tag{3.9}$$

In Equation (3.9), the total recycled capacity G appears twice on the right side, which prevents us from representing G by total time ΔT. On the other hand, we know that G cannot be less than the number of pages recycled in the first erased block. The number of recycled pages in the first block is in turn lower-bounded by $-aC_1 + b$. So we can derive the following equation:

$$\Delta T \leq \frac{Bt_{wr} + t_e}{-a} \times \ln \left| \frac{b - aG - aC_1}{b - aC_1} \right| - t_{wr} \times (-aC_1 + b) \ . \tag{3.10}$$

By Equation (3.10), we can represent the total recycled capacity G during time ΔT as follows:

$$G \geq \frac{b}{a} - C_1 - (\frac{b}{a} - C_1) \times e^{\frac{a^2 C_1 t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \ . \tag{3.11}$$

In the following we will use the notation

$$G = G(C, \Delta T)$$

to emphasize that G is a function of the global capacity C and the running time ΔT of the garbage collector.

If we denote by $c(t)$ the capacity at time t, we can lower-bound the capacity freed by the garbage collector starting at time t_0 and running for ΔT time as follows:

$$\begin{aligned} c(t_0 + \Delta T) &= c(t_0) + G(c(t_0), \Delta T) \\ &\geq \frac{b}{a} - (\frac{b}{a} - c(t_0)) \times e^{\frac{a^2 c(t_0) t_{wr} - ab t_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \quad . \end{aligned} \tag{3.12}$$

Figure 4 shows the shape of the curve of Equation (3.11).

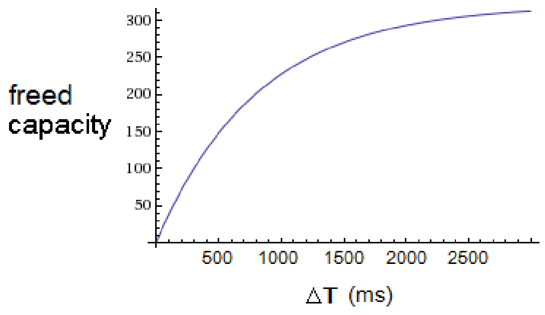


Fig. 4.: Relationship between time and freed memory capacity during garbage collection

Equation (3.12) provides a model for the effectiveness of a flash memory garbage collector in the time domain. We will make use of this model to derive the effect of flash memory garbage collection on the schedulability of real-time flash systems. We will follow a scheme originally defined to analyze the schedulability of thermally constraint systems [11]: We first define the characteristics for the critical instant, i.e. the worst-case workload arrival that causes maximum response time for the task being analyzed. We then proceed to determine the periodic steady state that guarantees all deadlines without exceeding the available memory capacity. This in turn allows us to compute the schedulable utilization bounds for the given garbage collection scheme.

E. Garbage Collection in the Presence of Other Workload

The model described in this chapter is valid for a garbage collector that runs without correlation by other workload. It can be applied easily to cases where the garbage collector has to run in the presence of other workload in the system. In such cases it is important to define how the system resources are partitioned between the garbage collector and the other workload. In this thesis we will describe three approaches on how to do just that, namely the *reactive* garbage collector, the *proactive* garbage collector, and finally the constant-bandwidth garbage collector. The three approaches differentiate from each other as follows:

- The *reactive* garbage collector is inactive as long as there is memory available in the system. Once the capacity hits a low watermark C_{min}, the garbage collector reclaims just sufficient memory to have the capacity exceed C_{min}. In this way the reactive garbage collector maximizes the CPU bandwidth allocated to other workload as long as there is memory capacity available.

- The *proactive* garbage collector extends the reactive one by allowing the garbage collector to run at maximum speed while the CPU is idle. This is particularly beneficial for lightly loaded systems with high memory consumption rates.
- The *constant-bandwidth* garbage collector has a minimum system bandwidth allocated at all times. In this way, the net memory consumption by the workload can be slowed down at the cost of reduced system bandwidth to the workload.

In the following chapters we will compare the effectiveness of these three approaches in terms of worst-case schedulability of real-time workload.

CHAPTER IV

FLASH MEMORY GARBAGE COLLECTION AND REAL-TIME TASKS

In the previous chapters we developed a performance model that describes the relation between time cost and number of freed memory for the flash memory garbage collector. In this chapter we will apply this model and develop a schedulability analysis for real-time tasks in the presence of a flash memory garbage collector. We will first describe the task model and then proceed to formulate the worst-case task release pattern, the so-called *critical instant*.

A. Periodic Real-Time Tasks

We consider a workload that consists of a set of identical-period[1] tasks $\Gamma_i : i = 1, 2, ..., n$, where each task $\Gamma_i = (P, w_i, \lambda_i)$ consists of a sequence of *jobs*, and any two jobs arrivals are separated at least by a minimum job interarrival time (called the *period*) P. Each job requires w_i processor cycles to complete in the worst case. In addition, task Γ_i updates flash memory pages at a memory consumption rate λ_i; that is, when executing for w cycles, task Γ_i updates (and thus renders invalid) $\lambda_i \times w$ pages.

B. Critical Instant

In order to compute the worst-case response time for jobs of a task Γ_i we need to determine the worst-case arrival pattern of jobs in Γ_i and other tasks. We call this the

[1]We limit ourselves to identical-period tasks. The critical instant for arbitrary-period tasks is an open problem.

critical instant. It is a well known result that the critical instant for preemptively scheduled independent periodic tasks without resource access is for a job of Γ_i to arrive together with jobs of all higher-priority tasks. We will describe in this section how to determine the critical instant for tasks in the presence of a flash memory garbage collector.

We define $c(t)$ and ΔT as the initial memory capacity and the length of time for garbage collection. Then we have the following lemma:

$$G(c(t), \Delta T) = \frac{b}{a} - c(t) - (\frac{b}{a} - c(t)) \times e^{\frac{a^2 c(t) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \ . \tag{4.1}$$

Lemma 2. *If $c(t)$ is a constant and ΔT increases, G increases, as well, that is, the amount of reclaimed memory increases with increasing running time of the garbage collector.*

Proof. First, we have:

$$f(c(t)) \geq -a \times c(t) + b \geq 0 \ , \tag{4.2}$$

$$\frac{b}{a} - c(t) \geq 0 \ , \tag{4.3}$$

$$e^{\frac{a^2 c(t) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \geq 0 \ . \tag{4.4}$$

As a result, whenever ΔT increases, $e^{\frac{a^2 c(t) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}}$ decreases. Therefore, $(\frac{b}{a} - c(t)) \times e^{\frac{a^2 c(t) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}}$ decreases, which leads the amount G of reclaimed memory to increase.

□

Lemma 3. *If ΔT is a constant and $c(t)$ increases, G decreases, that is, the amount of reclaimed memory decreased with decreasing number of invalid pages.*

Proof. First of all, we define k as a positive integer, and we set two variables, K_1 and K_2, as follows:

$$K_1 = e^{\frac{a^2 c(t) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \quad , \tag{4.5}$$

$$K_2 = e^{\frac{a^2 (c(t)+k) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \quad . \tag{4.6}$$

Since $-a(c(t)+k)+b \geq 0$ from Equation (4.2), we know that $K_2 \geq K_1$. Furthermore,

$$\begin{aligned} K_2 &= e^{\frac{a^2 (c(t)+k) t_{wr} - abt_{wr} - a\Delta T}{B \times t_{wr} + t_e}} \\ &= e^{\frac{-at_{wr}(-a(c(t)+k)+b) - a\Delta T}{B \times t_{wr} + t_e}} \leq 1 \ . \end{aligned} \tag{4.7}$$

This allows us to derive the following inequality:

$$\begin{aligned} & G(c(t), \Delta T) - G(c(t)+k, \Delta T) \\ &= [\frac{b}{a} - c(t) - (\frac{b}{a} - c(t)) \times K_1] \\ &\quad - [\frac{b}{a} - c(t) - k - (\frac{b}{a} - c(t) - k) \times K_2] \\ &= -(\frac{b}{a} - c(t)) \times K_1 + k + (\frac{b}{a} - c(t) - k) \times K_2 \\ &= k(1 - K_2) + (\frac{b}{a} - c(t)) \times (K_2 - K_1) \geq 0 \ , \end{aligned}$$

which proves the lemma. □

While the effectiveness of the garbage collector increases with decreasing available memory capacity, this does not make up for the difference in capacity, as the following lemma shows:

Lemma 4. *If* $c(t_0) \leq c(t_1)$*, then we have* $c(t_0)+G(c(t_0), \Delta T) \leq c(t_1)+G(c(t_1), \Delta T)$.

Proof. We assume the garbage collector takes $\Delta T'$ to increase memory capacity from $c(t_0)$ to $c(t_1)$. Because $c(t_0) \leq c(t_1)$ and $\Delta T' \geq 0$, we have $c(t_0) + G(c(t_0), \Delta T) = c(t_1)+G(c(t_1), \Delta T - \Delta T')$. By Lemma 2, the above value must be less than or equal to $c(t_1) + G(c(t_1), \Delta T)$, since $\Delta T - \Delta T' \leq \Delta T$. □

Lemma 5. *If the execution of a task is delayed, this reduces the memory capacity after the completion of the execution interval.*

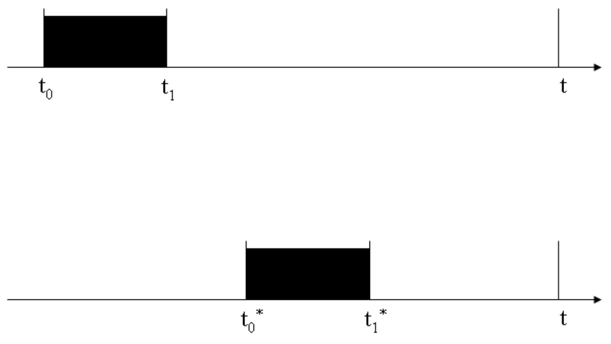


Fig. 5.: An example of a shifted successive execution part of job

Proof. We assume the fixed memory capacity $c(t_0)$ at t_0 like Figure 5 . There are four cases:

Case 1: The memory capacity neither hits C_{min} during $[t_0^*, t_1^*]$ for the job-shifted scenario nor during $[t_0, t_1]$ for the original scenario. Because both of scenarios don't hit C_{min}, they take the same time and consume the same amount of memory capacity. We define t_2 as the time in original scenario such that $t - t_2 = t - t_1^*$ and $t_2 - t_0 = t_1^* - t_0$. It is obvious that we have $c(t_1) \leq c(t_0)$. By Lemma 3, we have $G(t_1, t_2 - t_1) \geq G(t_0, t_1^* - t_0)$, and it's equal to $c(t_2) \geq c(t_1^*)$. By Lemma 4, we have $c(t_2) + G(t_2, t - t_2) \geq c(t_1^*) + G(t_1^*, t - t_1^*)$, that's equal to $c(t) \geq c(t^*)$.

Case 2: The memory capacity hits C_{min} during $[t_0^*, t_1^*]$ for the job-shifted scenario and also hits C_{min} in $[t_0, t_1]$ for the original scenario. Because both of scenario hit C_{min}, we have $c(t_1) = c(t_1^*) = C_{min}$. In addition, $t - t_1 \geq t - t_1^*$. By Lemma 2, we can make sure $c(t) \geq c(t^*)$.

Case 3: The memory capacity does not hit C_{min} during $[t_0^*, t_1^*]$ for the job-shifted scenario but hits C_{min} in $[t_0, t_1]$ for the original scenario. In this case, we introduce a transition scenario that the job is executed during $[t_0^{**}, t_1^{**}]$, where $t_0 \leq t_0^{**} \leq t_0^*$, $t_1 \leq t_1^{**} \leq t_1^*$, and the memory capacity hits C_{min} just at t_1^{**} in this scenario. The existence of this scenario is obvious. Define $c(t^{**})$ as the memory capacity at t in this scenario. Since the memory capacity hits C_{min} at the boundary t_1^{**} in the transition scenario, we can treat it as the scenario that the memory capacity hits C_{min} either during the execution time or after the execution time. Therefore, we can apply either of the analysis in Cases 1 and 2 to this scenario. First, we compare the original scenario with the transition scenario. We apply the analysis in Case 2 to both scenarios. Following the result of Case 2, we have:

$$c(t) \geq c(t^{**})$$

Second, we compare the transition scenario with the job-shifted scenario. We apply the analysis in Case 1 to both scenarios. Following the result of Case 1, we have:

$$c(t^{**}) \geq c(t^{*})$$

Therefore, we have $c(t) \geq c(t^{*})$.

Case 4: The memory capacity hits C_{min} during $[t_0^*, t_1^*]$ for the job-shifted scenario but doesn't hit C_{min} during $[t_0, t_1]$ for the original scenario. Since $t_0 \leq t_0^*$, we have $c(t_0) \leq c(t_0^*)$. It is impossible to hit C_{min} during $[t_0^*, t_1^*]$ and not hit C_{min} during $[t_0, t_1]$. The case will never happen. □

Theorem 1. *Assume that the tasks in a task system are phased so that the last job of each task during the busy interval Λ is completed a sufficiently-small time interval ϵ before the completion time of the last job of the next lower-priority task during the busy interval Λ. The release time of the first job of each task during Λ will be a critical instant when the memory capacity at the beginning of Λ is minimized.*

Proof. If we shift a task such that its last job during the busy interval Λ is completed ϵ time unit before the completion time of the last job of its next higher-priority during the busy interval Λ, then we will not change the worst-case preemption by the high-priority tasks. However, with the shifted task, more jobs should be pushed forward before the critical time instance of each task. Then, by Lemma 5, the initial memory capacity will be decreased. The release time of the first job of each task during Λ will be a critical instant when the memory capacity at the beginning of Λ is minimized. $\square$

CHAPTER V

SCHEDULABILITY ANALYSIS FOR IDENTICAL-PERIOD TASKS

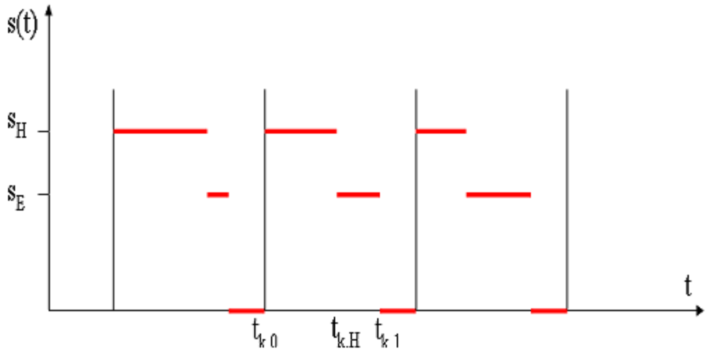


Fig. 6.: Illustration of a periodic task under garbage collection

Once the critical instant is identified, one can proceed to develop a schedulability analysis, which in turn allows to determine whether a set of tasks is schedulable in the presence of garbage collection. Unfortunately, even with the crisp definition of the critical instant as formulated in Theorem 1 earlier, it is very difficult to derive results that hold for arbitrary periodic task sets. Even for the mathematically much simpler case of scheduling in the presence of thermal constraints, arbitrary periodic task sets are an open problem. We therefore limit ourselves to the case of identical-period task sets, i.e., task sets where all tasks have the same period length.

In this chapter, we describe how to compute worst-case memory capacities and worst-case response times in the presence of garbage collection. For this we define the concept of *memory steady state*: The maximum schedulable utilization is achieved when the capacity at the beginning of the period is equal to the capacity at the end of the period. The rationale for this straightforward: Were the capacity to decrease, garbage collection would force tasks to miss their deadlines. Were it to increase on the other hand, then the utilization could be increased as well.

If we denote the total workload $W = \sum w_i$, Figure 6 illustrates the execution of a single-period task set in the presence of a garbage collector. The busy interval starts at time $t_{k,0}$ and ends at time $t_{k,1}$. The memory capacity hits C_{min} at time $t_{k,H}$, at which point the garbage collector starts reclaiming pages. This in turn reduces the processing speed allocated to real-time tasks from S_H to some lower speed S_E, which we call *equilibrium speed.* [1] In order to describe the memory steady state, we first obtain the time-instance formulas for $t_{k,0}$, $t_{k,H}$, and $t_{k,1}$.

The time $t_{k,0}$ is the beginning of the k-th period. At that point, the tasks consume memory at the S_H until the memory level hits $C_{k,H}$ at time $t_{k,H}$.

$$t_{k,0} = kP \ , \tag{5.1}$$

$$t_{k,H} = t_{k,0} + \frac{C_{k,0} - C_{k,H}}{S_H} \ , \tag{5.2}$$

The end of the busy interval can then be calculated by having the processor run at speed S_H until $t_{k,H}$ and then complete the rest of the workload W at speed S_E by

[1]The computation of S_E is described in Chapter VI.

time $t_{k,1}$:

$$t_{k,1} = \frac{S_H}{S_E}(t_{k,0} + \frac{W}{S_H}) - (\frac{S_H}{S_E} - 1)t_{k,H} \ . \tag{5.3}$$

By setting the memory capacity during the low-speed execution to the low-watermark, i.e., $C_{k,H} = C_{min}$ and $C_{k,1} = C_{min}$, we obtain the memory capacity at the beginning of the period as follows:

$$C_{k,0} = C_{min} + G(C_{min}, t_{k,0} - t_{k-1,1}) \ . \tag{5.4}$$

Based on the above formulas, we define the length of the high-speed execution interval $\pi_{k,0H}$, the length of the low-speed execution interval $\pi_{k,H1}$, the length of the overall execution interval $\pi_{k,01}$, and the length of the idle interval $\pi_{k-1,10}$ as follows:

$$\pi_{k,0H} = t_{k,H} - t_{k,0} = \frac{C_{k,0} - C_{k,H}}{S_H} \ , \tag{5.5}$$

$$\pi_{k,H1} = t_{k,1} - t_{k,H} = \frac{S_H}{S_E}(\frac{W}{S_H} - \pi_{k,0H}) \ , \tag{5.6}$$

$$\pi_{k,01} = t_{k,1} - t_{k,0} = \frac{W}{S_E} + (1 - \frac{S_H}{S_E})(\pi_{k,0H}) \ , \tag{5.7}$$

and

$$\begin{aligned} \pi_{k-1,10} &= t_{k,0} - t_{k-1,1} \\ &= (P - \frac{W}{S_E}) + (\frac{S_H}{S_E} - 1)(\frac{C_{k-1,0} - C_{k-1,H}}{S_H}) \ . \end{aligned} \tag{5.8}$$

Now we can compute $\lim_{k\to\infty} C_{k,0}$, which is the steady state memory utilization at the beginning of the period. As $k \rightarrow \infty$, we derive the fixed point $C^* = \lim_{k\rightarrow\infty} C_{k,0}$ by the following equation:

$$C^* = \frac{b}{a} - C_{min} - (\frac{b}{a} - C_{min}) \times e^{\frac{a^2 C_{min} t_{wr} - abr_{wr} - a\pi^*}{B \times t_{wr} + t_e}} , \tag{5.9}$$

where π^* denotes the length of the idle interval in steady state:

$$\begin{aligned} \pi^* &= t_{k,0} - t_{k-1,1} \\ &= (P - \frac{W}{S_E}) + (\frac{S_H}{S_E} - 1)(\frac{C^* - C_{min}}{S_H}) . \end{aligned} \tag{5.10}$$

Figure 7 illustrates how the relationship between C^* and P can be linearly approximated. This example plots C^* against P for a system with a given set of parameters.

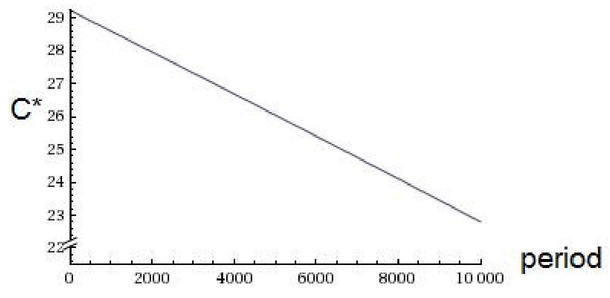


Fig. 7.: The relationship between P and C^*

Now, we go back to the original identical-periodic-task set. C^* is the minimal memory capacity at the beginning of the busy interval. Based on this, we want to obtain the memory capacity at the critical instant of each task. The following lemma allows us to formulate the latter as a function of the task's memory consumption:

Lemma 6. *Let C_i^* denote the memory capacity at the critical instant of task Γ_i. Then C_i^* can be expressed by the following formula:*

$$C_i^* = max\{C_{min}, C^* - \sum_{j>i} \lambda_j \times w_j\} \ . \tag{5.11}$$

Proof. At the critical instant $r_{i,c}$, the jobs of lower-priority tasks will be aligned back-to-back before $r_{i,c}$. If the memory capacity does not hit C_{min} at $r_{i,c}$, i.e., $C_i^* > C_{min}$, we have

$$C_i^* = C^* - \sum_{j>i} \lambda_j \times w_j \ . \tag{5.12}$$

If the memory capacity does hit C_{min} before $r_{i,c}$, then we have $C_i^* = C_{min}$, and the lemma is proved. □

Now we consider the response time $d_{i,c}$ for the instance $J_{i,c}$, that is, the c^{th} invocation of Task Γ_i. If $C_i^* = C_{min}$, we have

$$d_{i,c} = \frac{1}{S_E} \sum_{j\leq i} \lambda_j \times w_j \ . \tag{5.13}$$

Otherwise,

$$d_{i,c} = \lim_{k\to\infty} \pi_{k,01} - \frac{1}{S_H} \sum_{j>i} \lambda_j \times w_j \ . \tag{5.14}$$

If we define $C_{min} = 0$, we have

$$\lim_{k\to\infty} \pi_{k,01} = P - \lim_{k\to\infty}(t_{k,0} - t_{k-1,1}) \tag{5.15}$$

$$= P - \lim_{k\to\infty}(\frac{Bt_{wr} + t_e}{-a} \times \ln\left|\frac{b - aC_{k,0}}{b - aC_{min}}\right| - t_{wr} \times (C_{k,0} - C_{min})) \tag{5.16}$$

$$= P + \frac{Bt_{wr} + t_e}{a} \times \ln\left|\frac{b - aC^*}{b}\right| + t_{wr} \times C^* \ . \tag{5.17}$$

This gives rise to the following theorem, which bounds the worst-case delay:

Theorem 2. *The worst-case delay d_i^{PGC} experienced by a job in task Γ_i under proactive garbage collection can be bounded as follows:*

If $C^* - \sum_{j>i} \lambda_j \times w_j > C_{min}$,

$$d_i^{PGC} < P + \frac{Bt_{wr} + t_e}{a} \times \ln\left|\frac{b - aC^*}{b}\right| + t_{wr} \times C^* - \frac{1}{S_H}\sum_{j>i}\lambda_j \times w_j \ ,$$

otherwise,

$$d_i^{PGC} \leq \frac{1}{S_E}\sum_{j\leq i}\lambda_j \times w_j \ . \tag{5.18}$$

In memory systems, tasks must be completed before the end of their period. One model for such task sets uses a *deadline ration* to represent the need to complete early: In such task sets the deadline $d_{i,c}$ of the c^{th} invocation of task Γ_i is ζP_i after the release time of the c^{th} invocation. we call ζ the deadline ratio of the task set. The following corollary bounds the worst-case completion time of tasks with early deadlines in systems with proactive garbage collection.

Corollary 1. *If the deadline of task $D_i = \zeta P$, where $0 < \zeta \leq 1$, then under proactive garbage collection we have the worst-case delay d_i for task Γ_i bounded as follows:*

$$d_i^{PGC} < P + \frac{Bt_{wr} + t_e}{a} \times \ln \left| \frac{b - aC^*}{b} \right| + t_{wr} \times C^* \ ,$$

when

$$\frac{1}{S_E} \sum_{i=1}^{n} w_i \leq P \ .$$

Proof. Since the task Γ_n will experience the maximum delay, we have $d^{PGC} = d_n^{PGC}$. Then by Theorem 2, we have

$$d^{PGC} \leq \begin{cases} P + \frac{Bt_{wr}+t_e}{a} \times \ln \left| \frac{b-aC^*}{b} \right| + t_{wr} \times C^* \\ when \quad C^* > C_{min} \ , \\ \frac{1}{S_E} \sum_{j \leq n} w_j \\ when \quad C^* = C_{min} \ . \end{cases}$$

$C^* > C_{min}$ means that

$$\frac{1}{S_E} \sum_{j \leq n} w_j < P \ ,$$

and $C^* = C_{min}$ means that

$$\frac{1}{S_E}\sum_{j\leq n} w_j = P \ .$$

$\square$

A. Utilization-Based Analysis

Often, it is preferrable to have a quicker schedulability test at hand, which gives a first approximation at least of the ability of the system to meet the deadlines. One popular such approach is the so-called *utilization-based* analysis [27]. A utilization-based schedulability test compares the maximum utilization caused by the task set (the so-called *utilization factor* of the task set) against the so-called *schedulable utilization* of the system. The schedulable utilization denotes the maximum utilization level at which the system is guaranteed to be schedulable. This level depends on the available resources and on the scheduling algorithm and resource access protocols being used. If we define the utilization factor as

$$U = \sum_{i=1}^{n} \frac{\frac{w_i}{S_H}}{P} \ ,$$

then we can formulate the schedulable utilization for early-deadline periodic tasks as follows:

Lemma 7. *The maximum schedulable utilization U^{PGC} for a task set with deadline ratio ζ under proactive garbage collection can be expressed as*

$$\begin{aligned} U^{PGC} =& \min\{\zeta, \frac{S_E}{S_H}\zeta + (\frac{S_H}{S_E} - 1)(\frac{S_E}{S_H S_H P}) \\ & (\frac{b}{a} - \frac{b}{a}e^{\frac{(\zeta-1)aP}{Bt_{wr}+t_e}}) + bt_{wr}(\frac{S_E}{S_H P})\} \ . \end{aligned} \tag{5.19}$$

Proof. Corollary 1 states that whenever the task set is schedulable by the end of the period with a purely reactive garbage collector, i.e., when $\frac{1}{S_E}\sum_{i=1}^{n} w_i \le P$, then it can meet an early deadline ζP for some ζ if

$$P + \frac{Bt_{wr} + t_e}{a} \times \ln\left|\frac{b - aC^*}{b}\right| + t_{wr} \times C^* \le \zeta P \ . \tag{5.20}$$

Equation (5.20) can be rewritten by a tighter bound:

$$P + \frac{Bt_{wr} + t_e}{a} \times \ln\left|\frac{b - aC^*}{b}\right| \le \zeta P \ , \tag{5.21}$$

i.e.,

$$C^* \ge \frac{b}{a} - \frac{b}{a}e^{\frac{(\zeta-1)aP}{Bt_{wr}+t_e}} \ . \tag{5.22}$$

In addition, by Equation (5.10), and using Equation (5.22) (5.9) to replace π^*, we have

$$\frac{1}{S_E}\sum_{i=1}^{n} w_i \le \zeta P + (\frac{S_H}{S_E} - 1)(\frac{C^*}{S_H}) + bt_{wr} \ . \tag{5.23}$$

Therefore,

$$U = \sum_{i=1}^{n} \frac{\frac{w_i}{S_H}}{P} = \frac{1}{S_E}\sum_{i=1}^{n} w_i(\frac{S_E}{S_H P}) \tag{5.24}$$

$$\begin{aligned} &\le \frac{S_E}{S_H}\zeta + (\frac{S_H}{S_E} - 1)(\frac{C^*}{S_H})(\frac{S_E}{S_H P}) \\ &\quad + bt_{wr}(\frac{S_E}{S_H P}) \ . \end{aligned} \tag{5.25}$$

By replacing C^* by its lower bound from Equation (5.22), we represent the schedulable utilization as follows:

$$U \le \frac{S_E}{S_H}\zeta + (\frac{S_H}{S_E} - 1)(\frac{b}{a} - \frac{b}{a}e^{\frac{(\zeta-1)aP}{Bt_{wr}+t_e}})(\frac{S_E}{S_H S_H P}) \\ + bt_{wr}(\frac{S_E}{S_H P}) \ . \tag{5.26}$$

Moreover, the workload when executed at speed S_H has to be finished before the deadline:

$$\sum_{i=1}^{n} \frac{w_i}{S_H} \le \zeta P \ , \tag{5.27}$$

which in turn bounds the schedulable utilization:

$$U = \sum_{i=1}^{n} \frac{\frac{w_i}{S_H}}{P} \le \zeta \ . \tag{5.28}$$

By Equation (5.26) and Equation (5.28), the lemma is proved.

□

The schedulable utilization (if available) allows for the easy comparison of system designs, ranging from scheduling algorithms to resource access protocols to resource reclamation schemes. It is therefore particular by applicable when comparing proactive and reactive garbage collection schemes. It is easy to show that under reactive garbage collection we have

$$\frac{1}{S_E}\sum_{i=1}^{n} w_i \le \zeta P \ . \tag{5.29}$$

The maximum schedulable utilization U^{RGC} for the tasks under reactive garbage collection for an identical-period task sets can therefore be expressed as

$$U^{RGC} = \frac{S_E}{S_H}\zeta \ . \tag{5.30}$$

The figure on page 42 in Chapter VII compares the schedulable utilization of a proactive vs. a reactive garbage collector for varying deadline ratios for a given task set.

CHAPTER VI

CONSTANT-BANDWIDTH GARBAGE COLLECTION

While reactive garbage collection reclaims pages in a lazy fashion, and the proactive scheme described earlier reclaims pages during idle intervals as well, other schemes can be envisioned and assessed with our performance model. For example, the real-time tasks can be throttled to reduce their memory consumption. The freed CPU bandwidth could then be allocated to the garbage collector. We call this the *constant-bandwidth* garbage collector. The name of this garbage collection approach indicates that the latter could be implemented by having the system scheduler allocate a guaranteed system bandwidth to the garbage collector.

We define U_W and U_{GC} as the utilization allocated to workload and garbage collection, respectively. Based on that, we can define the workload memory consumption rate at the capacity C as follows:

$$\Lambda_W = \Sigma \lambda_i \frac{\frac{w_i}{S_H \times U_W}}{P_i} \ , \tag{6.1}$$

and reclamation rate is:

$$\Lambda_{GC} = -\frac{-aC+b}{(B+aC-b)t_{wr}+t_e} \times U_{GC} \ . \tag{6.2}$$

Hence, the overall memory consuming rate at capacity C is

$$\Lambda(C) = \Sigma \lambda_i \frac{\frac{w_i}{S_H \times U_W}}{P_i} - \frac{-aC+b}{(B+aC-b)t_{wr}+t_e} \times U_{GC} \ . \tag{6.3}$$

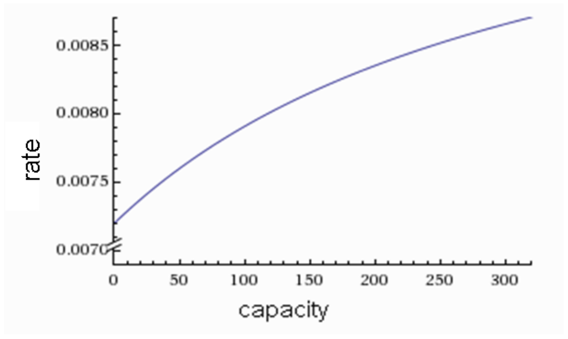


Fig. 8.: The relationship between speed and capacity

Figure 8 shows the relationship between speed and capacity for a flash memory system with parameters $a = 0.1$, $b = 32$, $t_{wr} = 1267$, $t_e = 1881$, $B = 64$. The system bandwidth is partitioned as follows: $U_w = 0.8$ and $U_{GC} = 0.2$.

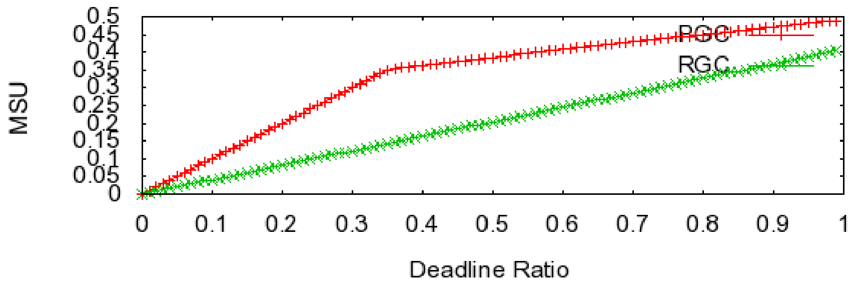
CHAPTER VII

PERFORMANCE EVALUATION

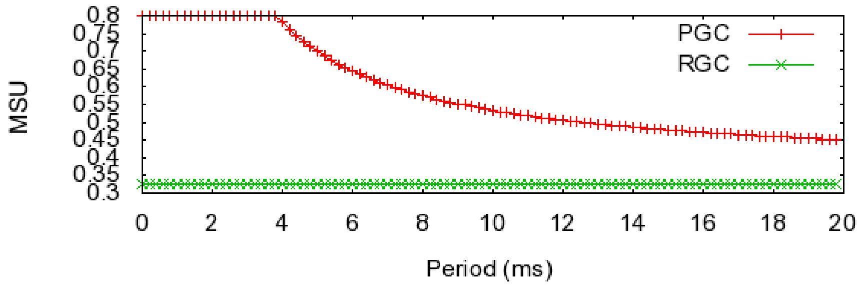
In this chapter, we compare the performance of proactive garbage collection with that of reactive garbage collection based on the theoretical results from the previous sections. We use maximum schedulable utilization (MSU) as the performance metric.

In our evaluation, we assume a flash memory with the following parameters: $t_r = 348$, $t_w = 919$, $t_e = 1881$. We also assume that the garbage collector effectiveness is characterized by $a = 0.1$ and $b = 32$. By Lemma 7, we know that the MSU depends on ζ and P. In the following, we fix one of the parameters and measure how the MSU is affected by the other parameters. In each setting, we measure MSU under proactive and reactive garbage collection. Figure 9 displays the level at which proactive garbage collection achieves a higher MSU than a reactive scheme. In the following, we explain the details of our results for each setting.

MSU vs. Deadline Ratio ζ: We measure how the deadline constraint affects MSU. We set $P = 200$ms and vary ζ from 0 to 1. When ζ is small, MSU is restricted by (5.28). This means that the flash memory capacity is always sufficient to support execution at speed S_H, and the garbage collector only executes during the idle interval. As ζ becomes larger, MSU is bounded by Equation (5.26). The proactive garbage collection take the advantage from executing real-time tasks by high speed and using the idle interval to reclaim invalid pages. In this way, the proactive garbage



(a) MSU vs. deadline ratio



(b) MSU vs. period time

Fig. 9.: Maximum schedulable utilization

collection can achieve the higher schedulable utilization than reactive schemes even when the deadline ratio is equal to one.

MSU vs. Period P: We measure how the value of period affects the MSU. We set $\zeta = 0.8$ and vary P from 0ms to 20ms. When P is small, U^{PGC} is much higher than U^{RGC}. With increasing P, U^{PGC} decreases and approaches to U^{RGC}, though the former is still higher than the latter. The reason is that with the small period P, the memory consumption in the busy interval is small, and the garbage collection is very efficient during the short idle time. The amount of freed pages is enough to keep the system run at high-speed, so MSU reaches the maximal value. When period P increases, the memory consumption increases, but the garbage collection can not maintain such efficiency. Since the memory which is recycled during idle time isn't sufficient to run system at high-speed for whole busy interval, garbage collector has to execute when memory capacity touch the lower-bound, so that MSU decreases.

CHAPTER VIII

CONCLUSION

The latency induced by page reclamation in NAND-based flash memory systems makes it difficult to take advantage of this storage technology in real-time embedded systems. To make matters worse, the flash translation layer often makes it impossible to apply partitioning techniques or other schemes that would allow to reduce priority inversions. In this thesis we describe an approach to model flash memory garbage collection in a way that captures its cost for task sets for a simplified memory consumption model. Unfortunately, garbage collection for flash memory behaves in a highly non-linear fashion: The lower the available memory, the higher the effectiveness of the garbage collection. As a result, it is not clear how much benefit an eager garbage collection scheme brings. While the chance of memory under-run is reduced by early page reclamation, the reduced effectiveness in turn increases the contention cost of the garbage collector for real-time tasks. Additional design issues must be considered as well, such as the need for wear leveling. For example, aggressive garbage collection tends to reclaim pages from blocks that have fewer invalid pages. Since the valid pages must be copied before erasing the block, this scheme leads to increased write activity and earlier wear-out of the flash memory. Finally, it appears that the design of effective flash memory schemes for hard-real-time systems is greatly dependent on the development of specially targeted flash translation layers that allow for flash-layer partitioning of resources or other mechanisms to reduce priority inversions for application tasks.

REFERENCES

[1] S. Nath and A. Kansal, “Flashdb: dynamic self-tuning database for nand flash,” in *IPSN*, pp. 410–419, 2007.

[2] D. Roberts, T. Kgil, and T. Mudge, “Integrating nand flash devices onto servers,” *Commun. ACM*, vol. 52, no. 4, pp. 98–103, 2009.

[3] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, “Ultra-low power data storage for sensor networks,” *TOSN*, vol. 5, no. 4, 2009.

[4] S. E. Company, “k9f2808u0b 16mb*8 nand flash memory data sheet,” http://www.datasheetcatalog.org/datasheet/SamsungElectronic/mXxqvtt.pdf.

[5] L. Chang, T. Kuo, and S. Lo, “Real-time garbage collection for flash-memory storage systems of real-time embedded systems,” *ACM Trans. on Embedded Comput. Syst.*, vol. 3, pp. 837–863, 2004.

[6] O. Kwon and K. Koh, “Swap-aware garbage collection for nand flash memory based embedded systems,” in *CIT*, pp. 787–792, 2007.

[7] O. Kwon, J. Lee, and K. Koh, “Ef-greedy: A novel garbage collection policy for flash memory based embedded systems,” in *International Conference on Computational Science*, Beijing, China, pp. 913–920, 2007.

[8] P. Wei, L. Yue, Z. Liu, and X. Xiang, “Flash memory management based on predicted data expiry-time in embedded real-time systems,” in *SAC*, pp. 1477–1481, 2008.

[9] X. Vera, B. Lisper, and J. Xue, "Data caches in multitasking hard real-time systems," in *RTSS*, pp. 154–165, 2003.

[10] M. Schoeberl and W. Puffitsch, "Nonblocking real-time garbage collection," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 1, 2010.

[11] S. Wang and R. Bettati, "Reactive speed control in temperature-constrained real-time systems," in *ECRTS*, pp. 161–170, 2006.

[12] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song, "A survey of flash translation layer," *J. Syst. Arch. - Embedded Syst. Design*, vol. 55, no. 5-6, pp. 332–343, 2009.

[13] Z. Huo, J. Yang, S. Lim, S. Baik, J. Lee, J. Han, I. Yeo, U. Chung, J. Moon, and B. Ryu, "Band engineered charge trap layer for highly reliable mlc flash memory," *in VLSI Technology, 2007 IEEE Symp.*, pp. 138 – 139, 2007.

[14] J. Kim, J. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366 – 375, 2002.

[15] S. Lee, D. Shin, Y. Kim, and J. Kim, "Last: Locality-aware sector translation for nand flash memory-based storage systems," *Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, 2008.

[16] M. Wu and W. Zwaenepoel, "Envy: A non-volatile, main memory storage system," in *Proc. 6th Int. Conf. Arch. Support for Programming Languages and Operating Systems*, San Jose, CA, 1994.

[17] P. Huang, Y. Chang, T. Kuo, J. Hsieh, and M. Lin, "The behavior analysis of flash-memory storage systems," in *ISORC*, pp. 529–534, 2008.

[18] Y. Chang, J. Hsieh, and T. Kuo, "Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design," in *DAC*, pp. 212–217, 2007.

[19] X. Wang and J. Wang, "A wear-leveling algorithm for nandflash in embedded system," in *Proc. 2008 Fifth IEEE Int. Symp. Embedded Computing*, Washington, DC, USA, pp. 260–265, 2008.

[20] D. Woodhouse, "Jffs: The journalling flash file system," presented in the Ottawa Linux Symposium, Ottawa, Canada, July 2001 (no proceedings).

[21] C. Manning, "Yaffs: Yet another flash file system." http://www.aleph1.co.uk/yaffs, 2004.

[22] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Tran. Comput. Syst.*, vol. 10, pp. 1–15, 1992.

[23] K. D. Nilsen, "Garbage collection of strings and linked data structured in real time," *Softw., Pract. Exper.*, vol. 18, no. 7, pp. 613–640, 1988.

[24] M. Schoeberl, "Scheduling of hard real-time garbage collection," *Real-Time Syst.*, vol. 45, no. 3, pp. 176–213, 2010.

[25] F. Pizlo, L. Ziarek, P. Maj, A. Hosking, E. Blanton, and J. Vitek, "Schism: Fragmentation-tolerant real-time garbage collection," in *SIGPLAN Conference*

on Programming Language Design and Implementation, Toronto, Ontario, pp. 146–159, 2010.

[26] Y. Kim, B. Tauras, A. Gupta, D. Mihai, and N. B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *SIMUL*, pp. 125 –131, 2009.

[27] J. Liu, *Real-Time Systems.* Upper Saddle River, NJ: Prentice Hall, 2000.

VITA

Chien-An Lai received his B.S. in computer science and information enginnering from National Taiwan University in June 2008. As an undergraduate, he worked under Dr. Tei-Wei Kuo on a project related to flash memory development. During his graduate career, he received the Royce E. Wisenbaker 39 Graduate Scholarship (2009-2010) and Industrial Affiliate Program Scholarship (2010-2011). His research focuses on real-time systems in flash memory, especially studying the impact of garbage collection on the timing required jobs. He received his M.S. in Computer Science from Texas A&M University in August 2011. He may be reached at the Department of Computer Science and Engineering, c/o Dr. Riccardo Bettati, Texas A&M University, College Station, TX 77843-3112.

The typist for this thesis was Chien-An Lai.