Energy-Efficient Flash-Memory Storage Systems with an Interrupt-Emulation Mechanism[*]

Chin-Hsien Wu, Tei-Wei Kuo, and Chia-Lin Yang
{d90003, ktw, yangc}@csie.ntu.edu.tw
Department of Computer Science and Information Engineering
National Taiwan University Taipei, Taiwan, 106

ABSTRACT

One of the emerging critical issues for flash-memory storage systems, especially on the implementations of many embedded systems, is on its programmed I/O nature for data transfers. Programmed-I/O-based data transfers might not only result in the wasting of valuable CPU cycles of microprocessors but also unnecessarily consume much more energy from batteries. This paper presents an interrupt-emulation mechanism for flash-memory storage systems with an energy-efficient management strategy. We propose to revise the waiting function in the Memory-Technology-Device (MTD) layer to relieve the microprocessor from busy waiting and to reduce the energy consumption of the system. We show that energy consumption could be significantly reduced with good saving on CPU cycles and minor delay on the average response time in the experiments.

Categories and Subject Descriptors

C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems; D.4.2 [**Operating Systems**]: Secondary storage; B.3.2 [**Memory Structures**]: Mass storage

General Terms

Design, Performance, Algorithm

Keywords

Flash Memory, Storage Systems, Embedded Systems, Interrupt-Emulation I/O, Energy-Efficient, Programmed I/O

1. INTRODUCTION

Flash memory has been widely adopted in various platforms for storage systems. Many of its usages are now well beyond its original designs. Application programs or operating systems on embedded systems usually use programmed I/O to access flash memory. Such a phenomenon might not only result in the wasting of valuable CPU cycles of microprocessors but also unnecessarily consume much more energy from batteries, especially for embedded systems.

In past few years, many excellent research results have been proposed for the performance enhancement of flash-memory storage systems, e.g., [1, 5, 6, 8, 9, 10]. In particular, Wu, et al. proposed to adopt SRAM as write buffers and presented several cleaning policies for garbage collection [10]. Kawaguchi, et al. proposed the *cost-benefit* policy [5], which uses a value-driven heuristic function as a block-recycling policy. Kim, et al. [6] proposed to periodically move live data among blocks so that blocks have more even life-times. To improve the overall performance, Chang and Kuo [1] proposed an adaptive striping architecture which consists of several independent banks. While energy-efficient designs have become an important issue on embedded systems, researchers have started exploiting energy-aware designs of flash-memory storage systems, e.g., [2, 3]. In particular, Douglis, et al. [3] provided a series of energy consumption measurement for flash memory under different percentages of capacity utilization.

The objective of this research is to evaluate the feasibility and benefits of energy-efficient flash-memory storage systems with interrupt-emulation. The goal is not only to relieve the microprocessor from wasting valuable CPU cycles because of the programmed-I/O-based data transfers of flash memory in many existing implementations, but also provide an energy-efficient strategy. First, we propose an interrupt-emulation mechanism for flash-memory storage systems, in which the the waiting function in the Memory-Technology-Device (MTD) layer is revised to relieve the microprocessor from busy waiting. Each I/O request of the flash-memory storage system is inserted into a queue for the storage system for scheduling, where a single task dedicated for the storage system is responsible of scheduling and dispatching requests and notifying the completion of each request. Furthermore, an energy-efficient strategy is presented for multi-bank flash-memory storage systems, especially on when to switch the power state of each flash-memory bank. We show that energy consumption could be significantly reduced, and much saving on CPU cycles could be achieved. In the experiments, it was also observed that only minor delay on the average response time of I/O requests in realistic traces.

The rest of this paper is organized as follows: Section 2 provides an overview of flash memory. Section 3 introduces

[*]Supported in part by a research grant from the National Science Council under Grant NSC 92-2213-E-002-065 and a research grant from the Academia Sinica.

the interrupt-emulation mechanism. Section 4 provides an energy-efficient strategy based on the interrupt-emulation mechanism. Section 5 shows experimental results. Section 6 is the conclusion.

2. FLASH-MEMORY CHARACTERISTICS

NAND flash memory is popular in the markets and considered for storage systems designs. NAND flash-memory might consist of multiple banks, and each bank is a flash unit that could operate independently. Each bank is partitioned into blocks, where each block is of a fixed number of pages. A block is the smallest unit of an erase operation, while read and write operations are handled by pages. The typical block size and page size are 16KB and 512B, respectively. Because of the hardware characteristics, data could not be overwritten over pages on updates. Instead, data must be written to some free space. The update strategy is called "out-place update". Pages that store the most recent versions are called "live pages", and pages that store old versions are called "dead pages".

Because of the out-place update strategy, a dynamic address translation mechanism is needed to map a given LBA (logical block address) to the physical address where the valid data reside. To accomplish this objective, a RAM-resident translation table is adopted. The translation table is indexed by LBA, and each entry of the table contains the physical address of the corresponding LBA. Each entry in the table is a triple ($bank\_num$, $block\_num$, $page\_num$) indexed by LBA, where each triple of an LBA indicates its corresponding bank number $bank\_num$, block number $block\_num$, and page number $page\_num$.

After a certain number of page writes, the amount of free space on flash memory would be low. Activities that consist of a series of reads, writes, and erases with the intention to reclaim free spaces would then start. The activities are called "garbage collection" and considered as overheads in flash-memory management. The objective of garbage collection is to recycle dead pages scattered over blocks so that they could become free pages after erasing.

3. AN INTERRUPT-EMULATION MECHANISM FOR FLASH-MEMORY STORAGE SYSTEMS

3.1 System Architecture

Table 1: NAND-Flash Performance (Samsung K9F6408U0A 8MB NAND flash memory)

Phase	Interval (μs)
The setup/busy phase of read	27/25
The setup/busy phase of write	27/350
The setup/busy phase of erase	31/2500

Due to the hardware characteristics of flash memory, application programs or operating systems on many embedded systems must use programmed I/O to access flash memory. The operation model of NAND flash, in general, consists of two phases: setup and busy phases. For example, the first phase (called "setup" phase) of a write operation is for command setup and data transfer. The command, the address, and the data are written to proper registers of flash memory in order. The second phase (called "busy" phase) is for busy-waiting of the data being flushed into flash memory. The operation of reads is similar to that of writes, except that the sequence of data transfer and busy-waiting is inverted. The phases of an erase is as the same as those of a write, except that no data transfer is needed in the setup phase. Note that writes and erases are time consuming, and most of the time is spent in the busy phase, where busy waiting occurs. The performance of NAND flash memory is summarized in Table 1.

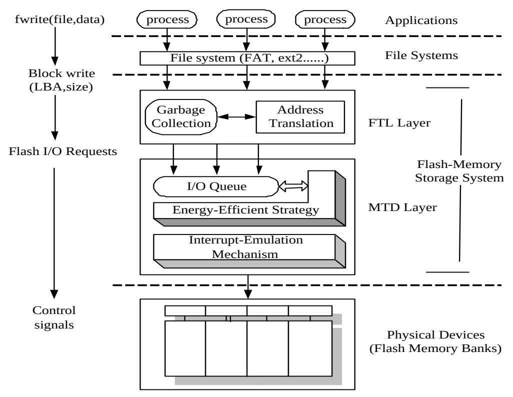


Figure 1: System Architecture

The system architecture of a flash-memory storage system consists of two layers, as shown in Figure 1. They are the Flash Translation Layer (FTL) and the Memory Technology Device (MTD) layer. The file systems layer is over the flash-memory storage system to provide a logical file interface for applications. FTL provides block-device emulation for transparent access from file systems without any modifications to existing file-system implementations [12]. Garbage collection, and address translation are handled in FTL. The MTD layer provides handling routines for read, write and erase operations, between devices (e.g., flash memory) and an upper layer (e.g., FTL) [11].

In this paper, we propose to provide an I/O request model for interrupt emulation (Please see Section 3.2.1) and to revise the waiting function in the MTD layer to relieve the microprocessor from busy waiting (Please see Section 3.2.2). In the MTD layer, we will present an energy-efficient strategy for multi-bank flash memory (Please see Section 4), especially on when to switch the power state of each flash-memory bank.

3.2 System Design

3.2.1 An I/O Request Model for an Interrupt-Emulation Mechanism

An I/O request could be modelled by a tuple (PID, $\{op_1, \cdots, op_n\}$), where PID denotes the ID of the process that issues the operations in the ordered list $\{op_1, \cdots, op_n\}$. Each element in the ordered list $\{op_1, \cdots, op_n\}$ that represents an operation over the flash-memory storage system, such as a read, a write, and an erase, must be executed according to its index order in the list. For example, an operation op_i could be ($READ$, ($bank_i$, $block_i$, $page_i$), $read\_data$), ($WRITE$, ($bank_i$, $block_i$, $page_i$), $write\_data$), or ($ERASE$,

```
SCHEDULING:
  Schedule I/O requests of the I/O queue according to some criteria
DISPATCHING:
  for each I/O request R in the I/O queue do
    for each operation op in R do
      Execute op  // Setup Phase
      if op is a write or an erase then
        The MTD dispatcher invokes a waiting function  // Busy Phase
      end if
    end for
    /*  NOTIFYING */
    The MTD dispatcher resumes the  corresponding I/O-requesting process of R
  end for
```

Figure 2: The MTD dispatcher

($bank_i$, $block_i$)), where $bank_i$, $block_i$, and $page_i$ represent the bank number, the block number, and the page number of the corresponding operation, respectively. When a process invokes an I/O system call, such as fwrite() or fread(), the file system might issue one or more requests to FTL to access data in the flash memory. FTL would then issue one or more I/O requests to the MTD layer.

An I/O queue is provided in the MTD layer for interrupt emulation and energy-efficient design (Please see Section 4). Each I/O request received by the MTD layer is inserted into the I/O queue and wait for the dispatching to the flash memory. I/O-requesting processes would suspend themselves when their I/O requests are inserted into the I/O queue. A system task, referred to as the *MTD dispatcher*, dispatches the first I/O request in the queue, and would invoke a waiting function to enter the busy phase if the operation of the I/O request is a write or an erase (Please see Section 3.2.2). When an I/O request completes, the MTD dispatcher would resume the corresponding process. The work of the MTD dispatcher is summarized in Figure 2.

3.2.2 The Design of the Waiting Function

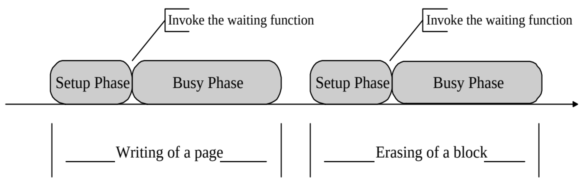


Figure 3: Two phases of a flash-memory operation

A waiting function is supposed to be invoked in the busy phase of a write or an erase, as shown in Figure 3. In the current MTD layer implementation, the waiting function is invoked to yield the CPU to other processes because the busy phase of a write or an erase is time-consuming, as shown in Function 1 in Figure 4. The invocation of yield() would virtually move the I/O-requesting process (in fact, it is the operating system running on behalf of the process) to the ready queue. If the I/O-requesting process that invokes yield() has the highest priority among ready processes in the ready queue of the operating system, then the I/O-requesting process would be dispatched such that it invokes yield() again, and the procedure repeats for the "timeout" period of time. Because the I/O-requesting process tends to be the highest priority process, such a behavior could be considered as a variation of programmed I/O. It might not only result in the wasting of valuable CPU cycles but also unnecessarily consume much more energy from batteries, especially for embedded systems. Note that no invocation of the waiting function is done for reads because their busy phase is short.

Function 1 The original waiting function

```
Input an operation op : a write or an erase
if op is a write then
  timeout = the time of the busy phase for the writing of a page
else if op is an erase then
  timeout = the time of the busy phase for the erasing of a block
end if
while timeout is valid do
  if op finishes then
    return
  else
    yield() { the flash memory is busy in the busy phase }
  end if
end while
if op fails then
  verify the status of the operation op
end if
```

Function 2 The proposed waiting function

```
Input an operation op : a write or an erase
if op is a write then
  timeout = the time of the busy phase for the writing of a page
else if op is an erase then
  timeout = the time of the busy phase for the erasing of a block
end if
sleep(timeout) { the flash memory is busy in the busy phase }
if op finishes then
  return
end if
if op fails then
  verify the status of the operation op
end if
```

Figure 4: Waiting Function

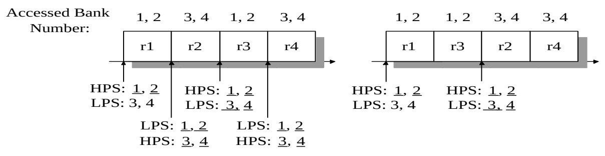
Different from the traditional implementations of the MTD layer, we propose to invoke sleep() in the waiting function to relieve the CPU from busy waiting as a part of the proposed interrupt-emulation mechanism. As shown in Function 2 in Figure 4, the busy-waiting while-loop is replaced with an invocation of sleep(). Such an invocation would put the MTD dispatcher into sleep until the expiration of the "timeout" period, and the MTD dispatcher (also referred to as the corresponding I/O-requesting process) is moved to the waiting queue of the operating system. When the timeout event occurs, the MTD dispatcher is awaken. When all operations in an I/O request complete, the MTD dispatcher resumes the corresponding I/O-requesting process.

Important technical questions for the proposed approach are on the predictability of the timeout period in the waiting function, the impacts on the system performance, and the potential overheads. We must point out that the access time of flash memory has a very small variance over widely different workloads. The average time of the busy phase for the writing of a page (512B) was 350 μs, and that of the erasing of a block (16KB) was 2500 μs. The variance of the access time was roughly between 10 μs and 30 μs. Therefore, the access time of flash memory is highly predictable. In the experiments, we shall provide measured results for the system performance under different timeout periods so that users could make the best decision for their selection. The impacts on the performance of the flash-memory storage system due to the interrupt-emulation mechanism would be dominated by that of the invocation of sleep() by the MTD dispatcher and the time to suspension/resume I/O-requesting processes (and the MTD dispatcher). We could show in the experiments that the overheads are negligible.

4. THE ENERGY-EFFICIENT STRATEGY

4.1 Overview

In a multi-bank flash system, each bank can switch to different power states independently. We assume a simple power management policy that switches a bank to a low power state (LPS) once it is idle and a high power state (HPS) to service a request, where state switching incurs both the performance and energy overheads. Consider a 4-bank flash system with 4 requests in the I/O queue, as shown in Figure 5. It requires 14 power state switchings, as shown in Figure 5.(a). However, it requires only 6 state switchings, as shown in Figure 5.(b). Based on the observation, a scheduling strategy is needed to minimize the number of state switchings to achieve energy saving.



(a) For I/O Requests Without Scheduling (b) For I/O Requests With Scheduling

Figure 5: Reordering of Requests

DEFINITION 1. *The Scheduling Problem of I/O Requests:* Given a collection T of I/O requests, and a cost vector $C(\tau_i, \tau_j)$ for any τ_i and $\tau_j \in T$, the scheduling problem of I/O requests is to find an execution order of T such that the total cost is minimum.

$C(\tau_i, \tau_j)$ represents the number of power state switchings required, i.e., $C(\tau_i, \tau_j) = |Bank_i| + |Bank_j| - 2 * (|Bank_i \cap Bank_j|)$, where $Bank_i$ and $Bank_j$ denote two sets of banks that τ_i and τ_j access, respectively.

The complexity in solving the scheduling problem of I/O requests depends on the number of combinations of banks, instead of the number of requests in general. Since the number of banks in many current implementations is very limited, we can either find out the best ordering of bank combinations or run approximation algorithms in the minimization of state switchings. For example, when there are 4 banks in a system, there are 15 bank combinations: $\{(1), (2), (3), (4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4), (1, 2, 3), (1, 3, 4), (2, 3, 4), (1, 2, 3, 4)\}$. We first come out a preferred order of requests in the 15 bank combinations. When request scheduling is needed, we always schedule requests in the same bank combination together and schedule requests in bank combinations according to a pre-determined preferred bank-combination order. Note that a preferred order could be found in an off-line fashion. We referred interesting readers to [13] for optimal solution using Branch and Bound.

For the rest of this section, we shall first address the dependency issues of requests to ensure the correctness of a program execution. We will then present two scheduling heuristics for request scheduling (instead of those based on a pre-determined preferred order). Their performance evaluation will be included in Section 5.2.

4.2 Scheduling Issues on Dependency Relations

There are three types of operations over the flash memory: read, write and erase. At the MTD layer, each operation in a request is associated with a physical block address (PBA). There could be access conflicts among operations on their physical block addresses. We say that two requests R_i and R_j *conflict with* each other if R_i and R_j satisfy any of the following conditions:

1. There exist one operation in R_i and one in R_j such that the two operations have the same PBA, and one of them is a write operation (or both are write operations).

2. There exist one operation in R_i and one in R_j such that one of the two operations is an erase operation, and the other is a read or write operation with a PBA inside the block to be erased by the erase operation.

For any two conflicting requests, their order is, in fact, not changeable; otherwise, the contents of the flash memory could be different after the reordering of I/O requests or wrong situations could happen. As a result, I/O requests in the I/O queue are partitioned into two sets: requests with conflicting operations and requests without any conflicting operations. The MTD dispatcher services all requests with conflicting operations first and services all requests without any conflicting operations in an order determined by the following scheduling algorithms.

4.3 Approximation Algorithms

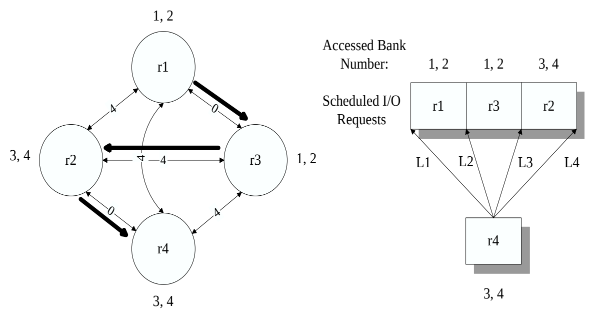
The purpose of this section is to present two algorithms for the scheduling of non-conflicting requests: A TSPP-based algorithm with an approximation bound and a greedy algorithm. The performance evaluation of the algorithms will be reported in Section 5.2. Before the presentation of the TSPP-based scheduling algorithm, we shall first prove the triangle inequality property for the cost vector:

LEMMA 1. *$C(\tau_i, \tau_j)$ satisfies triangle inequality if all I/O requests are non-conflicting requests. That is, $C(\tau_i, \tau_j) + C(\tau_j, \tau_k) \geq C(\tau_i, \tau_k)$ for any τ_i, τ_j, and $\tau_k \in$ a collection of non-conflicting requests T.*

Proof. $C(\tau_i, \tau_j) + C(\tau_j, \tau_k)$-$C(\tau_i, \tau_k)$ =($|Bank_i|$+$|Bank_j|$$-2 * (|Bank_i \cap Bank_j|)$)+($|Bank_j| + |Bank_k| - 2 * (|Bank_j \cap Bank_k|)$)- ($|Bank_i|$+$|Bank_k|$$-2$*($|Bank_i \cap Bank_k|$))=2*$|Bank_j|$-2*($|Bank_i \cap Bank_j|$+$|Bank_j \cap Bank_k|$)+ 2*$|Bank_i \cap Bank_k|$$\geq$0. As a result, $C(\tau_i, \tau_j)$ satisfies triangle inequality. □

We could apply a TSPP-based approximation algorithm on the scheduling of non-conflicting requests because the cost vector satisfies triangle inequality (Please see to Lemma 1). TSPP is defined as follows [4]: Given a complete graph with vertex V, and a nonnegative edge cost vector for any edges, the travelling salesman path problem is to find a Hamiltonian path with the minimum cost. Figure 6.(a) shows a TSPP-based solution for the example shown in Figure 5. Each request in the scheduling problem is a node in the TSPP instance. The number marked on the edge of (τ_i, τ_j) corresponds to $C(\tau_i, \tau_j)$. We adopt the well-known 2-approximation algorithm[1] in [7] because of its simplicity

[1]The 2-approximation algorithm mainly consists of the minimum spanning tree algorithm and the Eulerian tour.



(a) A TSPP-based solution (b) A greedy solution

Figure 6: Solutions based on a TSPP-based approach and a greedy approach

in implementation. Note that when a new non-conflicting I/O request arrives for scheduling, all scheduled I/O requests must be rescheduled with the new I/O request by the TSPP-based approximation algorithm. The complexity of each rescheduling is $O(|V|^2)$ for the 2-approximation algorithm.

Request scheduling could also be done by a simple and efficient heuristics, referred to as the *greedy algorithm* for the rest of this paper: Let S be a schedule of all non-conflicting pending I/O requests. When a new request arrives, the greedy algorithm simply scans over the schedule from the front to the end to find the best place to insert the new request in. The selection is based on the minimization of the final cost. For example, let r4 be a new request, and there are four potential locations (L1, L2, L3, or L4) for insertion, as shown in Figure 6.(b). According to the greedy algorithm, L3 and L4 would be better than L1 and L2 because of less cost. The time complexity of each rescheduling is $O(|V|)$.

5. PERFORMANCE EVALUATION

Table 2: Characteristics of Traces

File system	FAT32
Applications	Web Applications, E-mail Clients, MP3 player, and Virtual Memory Activities
Duration	2 Hours
Total Data Written	122 MB
Total I/O Requests	33,000
Read / Write Ratio	44% / 56%
Mean Read/Write Size	31.3 / 16 Pages

A series of experiments was conducted on an AMD-Duron (750Mhz) machine running RedHat 7.3 with the proposed MTD layer. The performance evaluation was done over a 4-bank NAND type flash system. The size of each bank was 25MB, and the page size was 512B. The characteristics of the traces are summarized in Table 2. Note that the page allocation policy was based on the striping architecture [1].

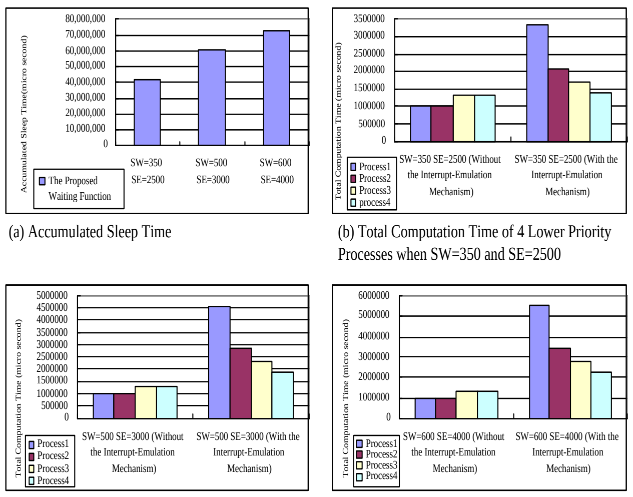
The evaluation of the proposed energy-efficient interrupt-emulation mechanism was done in two parts. First, we demonstrated the capability of the interrupt-emulation mechanism in Section 5.1. We then showed the advantages of the proposed method in energy-efficiency considerations.

5.1 Performance of the Interrupt-Emulation Mechanism

Table 3: The Overheads of the Interrupt-Emulation Mechanism

	Ave. (μs)	Deviation (μs)
Overheads of the invocation of sleep()	20	10
Overheads of the suspension and resumption time	30	10
Overheads of each I/O request	50	20

The overheads for the supporting of the interrupt-emulation mechanism mainly came from the invocation of sleep() by the MTD dispatcher and the time to suspend/resume I/O-requesting processes and the MTD dispatcher. We measured these overheads in the kernel mode for more precise results. The results are summarized in Table 3. We can see that the total overheads for each I/O request is about 50 μs on average, and the deviation for each I/O request is about 20 μs. Compared with the average service time, 2 ms and 6 ms for reads/writes, the performance overheads (that is about 50 μs on average) from the interrupt-emulation mechanism was reasonable.



(a) Accumulated Sleep Time

(b) Total Computation Time of 4 Lower Priority Processes when SW=350 and SE=2500

(c) Total Computation Time of 4 Lower Priority Processes when SW=500 and SE=3000

(d) Total Computation Time of 4 Lower Priority Processes when SW=600 and SE=4000

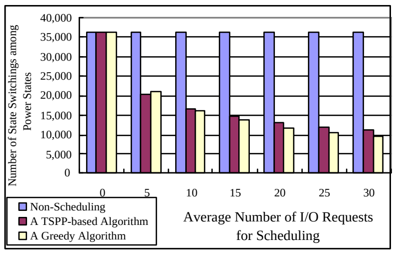
Figure 7: Experimental Results of the Interrupt-Emulation Mechanism. (SW and SE denote the timeout period of the busy phase for write and erase operations, respectively)

We measured the accumulated sleep time for 3 different timeout periods, as shown in Figure 7.(a). We can see that longer timeout periods resulted in longer sleep time as expected. During the sleep period of the MTD dispatcher, the microprocessor could be used by other processes thereby increasing the system throughput. To quantify this effect, we executed four additional CPU-bound processes with lower priorities than the MTD dispatcher. The period and computation time of the four CPU-bound processes was described in Table 4. We measured the CPU time spent on these four processes with and without the interrupt-emulation mechanism. The results are shown in Figure7.(b), (c) and (d). We can see that, with the support of interrupt emulation, these 4 processes obtained far more CPU cycles, compared to those under the original flash driver implementations, and a longer timeout period resulted in a larger discrepancy as expected. Note that process1 obtained more CPU cycles than the other 3 processes did because it had the shortest period.

Table 4: CPU-bound processes

	Process1	Process2	Process3	Process4
Period (μs)	300	370	710	780
Computation Time (μs)	30	30	40	40

5.2 Effectiveness of the Energy-Efficient Strategy



Number of State Switchings among Power States

Figure 8: Experimental Results of the Energy-Efficient Strategy

The proposed TSPP-based and the greedy algorithms were evaluated for energy efficiency considerations. One important parameter that affected the effectiveness of the energy-efficient strategy was the number of pending requests in the I/O queue when scheduling was performed. We measured the number of state switchings by varying the number of pending I/O requests. The experimental results were shown in Figure 8. We could see that the number of state switchings was reduced significantly, compared to that without request scheduling. As we increased the number of pending I/O requests, better improvement was observed for both of the proposed algorithms. Note that the number of pending I/O requests was mainly determined by the workload in a real system and the timeout period of the MTD dispatcher. When the number of pending requests was large, the greedy algorithm outperformed the TSPP-based algorithm although the TSPP-based algorithm could provide an approximation bound to the optimal solution.

6. CONCLUSION

The paper proposes an energy-efficient flash-memory storage systems with interrupt-emulation mechanism to relieve the microprocessor from the wasting of valuable CPU cycles in many existing embedded systems implementations. We propose to revise the waiting function in the Memory-Technology-Device layer to avoid busy waiting. An I/O queue and a request-dispatching task are proposed to schedule I/O requests and to notify the completion of each request. An energy-efficient strategy is presented for multi-bank flash-memory storage systems, especially on when to switch the power state of each flash-memory bank. The strategy is to minimize the energy consumption of flash memory without resulting in performance degradation. Issues on the execution orders of read, write, and erase operations over flash memory are also explored. We show that energy consumption could be significantly reduced with minor delay on the average response time of I/O requests in realistic traces, and much saving on CPU cycles could be achieved.

For future research, we should further explore the characteristics of flash memory, especially when application semantics is considered. With joint considerations of application designs and flash-memory characteristics, much performance improvement could be reached with even less system overheads and cost.

7. REFERENCES

[1] L. P. Chang and T. W. Kuo, “An Adaptive Stripping Architecture for Flash Memory Storage Systems of Embedded Systems,” IEEE Eighth Real-Time and Embedded Technology and Applications Symposium (RTAS), San Jose, USA, Sept 2002.

[2] L. P. Chang and T. W. Kuo, “A Dynamic-Voltage-Adjustment Mechanism in Reducing the Power Consumption of Flash Memory for Portable Devices,” IEEE Conference on Consumer Electronic (ICCE 2001), LA. USA, June 2001.

[3] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, “Storage Alternatives for Mobile Computers,” Proceedings of the USENIX Operating System Design and Implementation, 1994.

[4] M. R. Garey, and D. S. Johnson, “Computers and intractability”, 1979.

[5] A. Kawaguchi, S. Nishioka, and H. Motoda, “A Flash-Memory Based File System,” USENIX Technical Conference on Unix and Advanced Computing Systems, 1995 .

[6] H. J. Kim and S. G. Lee, “A New Flash Memory Management for Flash Storage System,” Twenty-Third Annual International Computer Software and Applications Conference October 25 - 26, 1999 Phoenix, Arizona.

[7] Vijay V. Vazirani, “Approximation Algorithm,” Springer publisher, 2001.

[8] C. H. Wu, L. P. Chang, and T. W. Kuo, “An Efficient B-Tree Layer for Flash-Memory Storage Systems,” The 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003).

[9] C. H. Wu, L. P. Chang, and T. W. Kuo, “An Efficient R-Tree Implementation over Flash-Memory Storage Systems,” The 11th International Symposium on Advances in Geographic Information Systems (ACM-GIS 2003).

[10] M. Wu, and W. Zwaenepoel, “eNVy: A Non-Volatile, Main Memory Storage System,” Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1994), 1994.

[11] http://www.linux-mtd.infradead.org/

[12] Intel Corporation, “Understanding the Flash Translation Layer(FTL) Specification”.

[13] Theo C. Ruys, “Optimal Scheduling using Branch and Bound with SPIN 4.0,” Supported by Project AMETIST, Department of Computer Science , University of Twente.