Non-Preemptive Demand Paging Technique for NAND Flash-based Real-Time Embedded Systems

Wangyu Kim and Dongkun Shin, *Member*, IEEE

Abstract —*NAND flash memory is utilized as code storage as well as for file system storage in consumer electronics. The demand paging technique for NAND flash code storage can reduce the required main memory space. However, in real-time systems, demand paging may invoke several problems, one such example is unpredictable timing behavior. Moreover, when NAND flash memory is used for both code storage and file system storage, the resource conflict issue needs to be resolved to allow for simultaneous accesses of demand paging and file system requests. This paper addresses several practical problems of NAND flash-based demand paging in real-time embedded systems and proposes a non-preemptive demand paging technique to resolve the resource conflict within non-preemptive critical sections. Experiments on a real mobile phone platform show that the proposed demand paging requires no significant overhead*[1].

Index Terms —NAND flash memory, demand paging, resource synchronization, embedded storage, real-time systems.

I. INTRODUCTION

In mobile consumer electronics, flash memory is an indispensable component due to its non-volatility and low power consumption. Flash memories are subdivided into two main classes: NOR type and NAND type. While NOR flash memory is well suited for code storage because of its eXecute-In-Place (XIP) feature, NAND flash memory is more suitable for data storage because of its fast write performance and its lower cost per bit in comparison to those of NOR flash memory [1].

Therefore, mobile systems which require data storage as well as code storage incorporate both NOR and NAND flash memories. In order to reduce the component cost, it is better to use a single-type storage device to service both code and data. For example, NAND flash shadowing is a widely-used technique for NAND flash-only storage architecture, in which both code and data are stored in NAND flash memory and all the program codes are loaded into the main memory during start-up. We can eliminate NOR memory in the system by using NAND flash shadowing. However, the shadowing technique requires sufficient main memory to retain all program codes, which increases the product cost and power consumption.

Demand paging, in which the main memory contains only part of the program code, is a promising technique to reduce the size of the main memory. If an attempt is made to access a page not contained within the main memory, a page fault occurs, and the page fault handler loads the required page into the main memory from the NAND flash storage. Therefore, programs would no longer be constrained by the amount of available physical memory.

Although the demand paging technique is widely used in general-purpose operating systems such as Linux, a real-time operating system generally does not support demand paging due to its unpredictability. This unpredictable behavior first arises due to unpredictable timing. The page faults which occur during program execution make it difficult to guarantee the real-time constraints. Demand paging may cause unexpected long-term delays in the execution of a process while the missing pages are loaded into the memory.

The second issue with unpredictability concerns the scheduling behavior. In NAND flash-based demand paging, the NAND flash storage is a shared resource since it services both the data request from the file system and the code request from the demand paging. Though NOR flash memory supports a simultaneous read operation while writing or erasing another flash memory partition, NAND flash memory does not provide such a feature. This limitation results in a resource conflict when more than two requests attempt to simultaneously access the NAND flash storage. Therefore, a synchronization mechanism is required to deal with these conflicting requests. However, the use of synchronization can result in changes in the task schedule, contrary to the user's expectations. The unpredictable scheduling behavior results in unpredictable timing behavior.

This paper addresses several problems involved in the use of demand paging in real-time consumer electronics and then provides practical solutions. We focus in particular on the non-preemptive critical section (NPCS) that is widely used in embedded systems but which makes it difficult to use demand paging. For the NPCS, we propose the page pinning technique and the non-preemptive demand paging (NPDP) technique, in which there is no need for application developers to consider the effects of demand paging. In addition, legacy codes developed without consideration for demand paging can be reused in NAND flash-based demand paging systems. We applied the proposed NAND flash-based demand paging technique to an actual mobile phone, a typical real-time system that uses flash memory, and ascertained that the demand paging did not disturb the time-critical mobile phone

[1] This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0010387).

W. Kim is with Samsung Electronics, Ltd., Suwon, Korea (e-mail: wangyu.kim@samsung.com).

D. Shin (corresponding author) is with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea (e-mail: dongkun@skku.edu).

Contributed Paper
Manuscript received 07/08/10
Current version published 09/23/10
Electronic version published 09/30/10.

operations. Experimental results showed that there was no significant run-time overhead due to the use of demand paging.

The rest of the paper is organized as follows: Section 2 briefly reviews the related works of NAND flash-based demand paging, Section 3 introduces the architecture of NAND flash memory, Section 4 describes the proposed NAND demand paging technique, Section 5 presents the experimental results, and Section 6 concludes the paper with a summary as well as considerations for future work.

II. RELATED WORKS

The critical issue in demand paging architecture is how to support simultaneous accesses to both code and data. For example, it takes 2 msec to erase a block of NAND flash memory. Moreover, if the multi-block erase operation is used, the erase time is 4 msec. Without a simultaneous access mechanism, a read request has to wait a maximum of 2 msec (or 4 msec) while the flash memory serves the erase operation.

NOR flash memory provides both the suspend and resume functions for program or erase operations in order to support a read operation while writing or erasing the flash memory. Brown *et al*. [2] proposed a NOR flash memory which allows the program and erase operations in one partition to be suspended by a software command in order to read the program code in another partition. It takes 5~20 μsec to suspend the program or erase operation. A partition is a group of blocks that share common program and erase circuitry and a command status register. Because each partition has its own status register, the flash can continue from the point at which the suspend command was issued by using the resume command. Using the suspend/resume feature, a true background erase can thus be achieved. Gefen *et al*. [3] proposed a hardware mechanism which can support automatic suspend and resume operations to enable dual functionality of the flash memory while Brown *et al*. used the software command.

For NAND flash memory, one simple technique is to use a dual-bank architecture. The device can begin programming or erasing in one bank, and then simultaneously read from the other bank, with zero latency. This can prevent the read request from having to wait for the completion of program or erase operations. However, this solution is not flexible because it is difficult to fit data and code completely within the fixed banks.

Several NAND storage architectures have been studied for the use of NAND flash memory as code storage. Park *et al*. [4] proposed a NAND XIP controller to provide XIP functionality. The controller uses a small amount of SRAM for cache, improving the average access time of the NAND flash memory. Park *et al*. [5] also studied a NAND flash-based demand paging technique which exploited the main memory as a page cache. Specially, they were interested in energy consumption and, therefore, proposed an energy-aware page replacement algorithm which assigns higher priorities to clean pages rather than dirty pages when selecting a victim page because the energy consumption of writing is much higher than that of reading in NAND flash memory.

Since a memory management unit (MMU) is required to implement demand paging, the compiler-assisted code overlay technique [6] is useful for low-end embedded systems without an MMU. In the overlay technique, the compiler transforms a program using a post-pass analysis of an executable image. The transformed program loads the code of called function into the main memory on demand at run time without the need for hardware intervention.

Joo *et al*. [7] proposed a demand paging technique for fusion NAND flash memories [8], consisting of a NAND flash array and SRAM buffers. The technique utilizes the internal SRAM buffer as a page cache in order to avoid loading data onto the external memory.

Two kinds of optimization techniques can be considered to improve the demand paging performance: reducing the frequency of page faults and reducing the page fault latency. Prefetching is one method for reducing the number of page faults, and there have been several studies on improving the accuracy of prefetching [1, 9].

In *et al*. [10] proposed a Search-While-Load (SWL) demand paging scheme which reduces the long latency of page faults by overlapping the page load operation with the victim page search operation. While the requested page is being loaded from the NAND flash memory, the page fault handler searches the victim page that is to be evicted in order to make room for the requested page. Hyun *et al*. [11] proposed the vectored read scheme, which reduces the page fault latencies at the fusion NAND flash memories.

Those researches have focused only on enabling the NAND flash for use as code storage but did not address the concurrent access of code and data requests. To the best of our knowledge, our work is the first attempt to attend to the practical issues of NAND flash-based demand paging in real-time embedded systems.

III. NAND FLASH ARCHITECTURE

As shown in Fig. 1, a NAND flash device includes control logic, an address register, a status register, a command register, a NAND flash array, and an I/O buffer.

In order to read a page from the NAND device, the following four steps must be performed: 1) the target address and command are sent, 2) the device driver waits until the status register is changed into the *ready* status, indicating that the target page has been loaded into the I/O buffer, 3) the error collection code (ECC) is verified, and 4) the required data is copied from the I/O buffer to the host. The program operation is composed of the following three steps: 1) the target address, data and command are sent, 2) the status register of the NAND device is changed into the *busy* status, and the device driver waits until the status is updated to the *ready* status (during this step, the task of accessing the NAND device can be preempted by higher priority tasks), and 3) the device driver does any required error handling and notifies the command completion

to the host. Fig. 2 shows the access steps for NAND flash reading and programming.

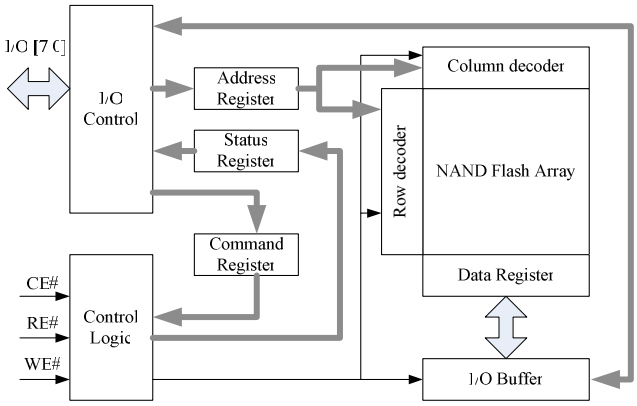

Fig. 1 The architecture of NAND flash memory.

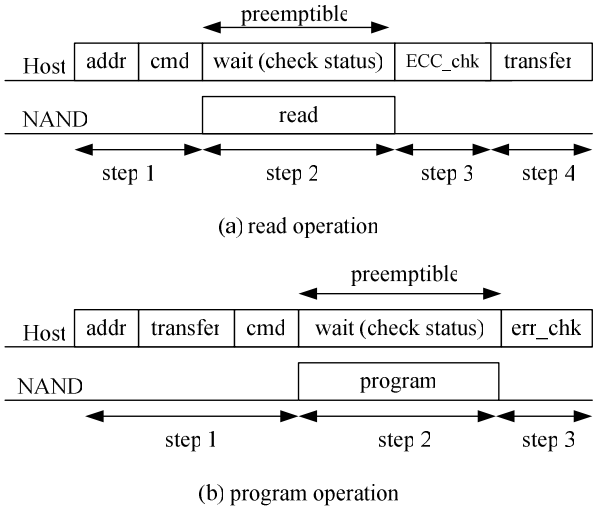

Fig. 2 Access steps for NAND flash memory.

Among the several components within a NAND device, the status register and I/O buffer are important shared resources. Unless these components are protected, the wrong data will be read or written. For example, when task τ_1 sends a read command and then waits for the change of status register, if another task τ_2 preempts task τ_1 and sends a read command, task τ_1 will read the data of τ_2 at the I/O buffer after it resumes.

IV. Flash Memory Demand Paging

A. Resource Synchronization

The device driver of NAND flash memory generally uses a semaphore (S_F) to protect the status register and I/O buffer from being altered by other operations. Only when an operation acquires the semaphore, it can send a command to the NAND device. Fig. 3 shows our NAND storage architecture, which can be simultaneously accessed by both the file system and the demand paging manager. The file system sends read/write requests through the flash translation layer (FTL) and the NAND flash device driver (FDD). The FTL is responsible for translating a logical page address into a physical page address and also performs the block allocation, block reclamation and wear-leveling. The FDD provides low-level interfaces for read, program and erase operations. The demand paging manager directly calls the flash device driver since it uses only the read operation.

The demand paging manager manages a page cache, which retains only the pages likely to be accessed within a short time. We used a clock (i.e., second chance) replacement algorithm to manage the page cache. Since there is no hardware support of reference bit or reference counter in most embedded systems, we emulated the reference bits by artificially invalidating some valid pages at every page fault. When a falsely-invalidated but cache-resident page is accessed, the page is marked as a valid page. We call this situation a false fault.

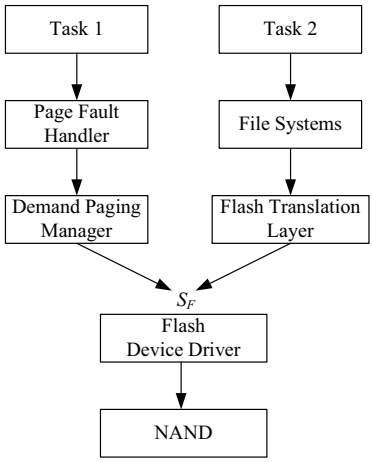

Fig. 3 Resource sharing in NAND-based demand paging.

The proposed demand paging is implemented with the help of an MMU, which contains an address translation table that provides the mapping information between the virtual address and the physical address. When there is no valid address translation information of the page to be accessed, a page fault exception occurs. Then, the demand paging manager loads the page into the page cache (unless it is a false fault) and inserts the translation information of the page into the page translation table. In order to prevent a nested page fault exception, the codes of the page fault handler and the demand paging manager are not evicted from the main memory.

In this scheme, the semaphore of the flash device driver (S_F) is exploited for resource synchronization, which is similar to the page fault handling mechanism in a general purpose OS, such as Linux. If a page fault occurs when the FTL has the semaphore S_F, the demand paging manager should wait until S_F is released by the FTL. Therefore, the task which invoked the page fault should be blocked until S_F is available, and other tasks will be scheduled. That is, the page fault handling cannot interrupt the previous operation of NAND storage. Fig. 4 shows an example of page fault handling. When the page fault exception occurs during the execution of task 2, it tries to obtain the semaphore S_F via the demand paging manager.

Since S_F belongs to task 1, task 2 enters into the sleep state. After task 1 releases S_F, task 2 acquires the semaphore S_F and then reads the required pages from the NAND device.

The page fault latency (*PFL*) is the time from the occurrence of the page fault to the completion of a page load and can be expressed as follows:

$$PFL = t_{sem} + t_{wait} + t_{read} + t_{cache} \qquad (1)$$

where t_{sem}, t_{wait}, t_{read} and t_{cache} represent the semaphore handling overhead, the waiting time for the semaphore, the page read time from the flash array to the host memory and the page cache update time, respectively. While the values of t_{sem}, t_{read} and t_{cache} are fixed for a given system, the value of t_{wait} varies depending upon how long the task should wait for the semaphore S_F. Therefore, the demand paging overhead of a task depends on t_{wait} as well as the number of page faults that occur during the task execution.

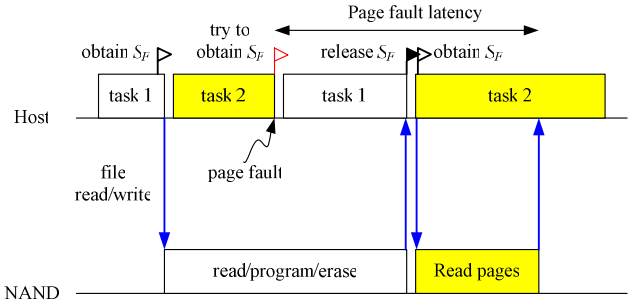

Fig. 4 Page fault handling.

B. Practical Issues in Demand Paging

The described demand paging technique introduces several problems when it is used in real-time operating systems.

1) Unpredictable Timing Behavior

The first problem is the unpredictable timing behavior of demand paging. Especially, the semaphore waiting time, t_{wait}, can be unbounded due to the priority inversion problem. For example, while a low-priority task τ_L accesses a file in NAND storage, a high-priority task τ_H can preempt task τ_L and generate a page fault, which also accesses the NAND storage. Then, task τ_H should wait until task τ_L releases S_F. Moreover, task τ_H can be delayed by all other tasks whose priorities are higher than τ_L but lower than τ_H. The priority inversion problem can be solved by using a priority inheritance or priority ceiling protocol [12] if the operating system provides the scheme. Then, t_{wait} can be bounded by the block erase time.

However, it is difficult to predict the accurate number of page faults during task execution, which must be known to estimate the demand paging overhead of a real-time task. Particularly for embedded systems using pre-built libraries, such as a mobile phone, it is impossible to analyze the worst-case behavior of an application.

2) Deadlock

The second problem of demand paging is that a deadlock situation can happen if the file system invokes a page fault while it has the semaphore S_F to access the data storage. If the file system code is shadowed at boot time, no code page fault will occur during the file system execution, however, data page faults can occur. For example, assume that a task attempts to copy data from the main memory into NAND storage via the file system. First, the file system acquires the semaphore S_F. Then, the data in the main memory is copied into the I/O buffer of the NAND device. If the data is not located in the physical memory, a data page fault can occur. When the data page fault handler attempts to acquire the semaphore S_F, a deadlock occurs since S_F is already being occupied by the file system and the file system will not release the semaphore until the page fault is resolved.

3) Debugging

When a program developer sets a software breakpoint at program code using a debugger, the debugger replaces the target program instruction with a trap instruction that causes an exception. When the trap instruction is encountered, the debugger catches the exception and stops the program. When the developer commands the debugger to continue, it restores the original instruction. Therefore, the precondition for software breakpoint is that the program code should reside within the physical memory and should be writable. However, in demand paging schemes, it is possible that the target instruction will not be located in the physical memory and, thus, a breakpoint cannot be set.

4) Non-Preemptive Critical Section

The last issue is unexpected scheduling behavior. There are two kinds of protection mechanisms for the critical section in embedded systems; one mechanism is to use a semaphore and the other is to use the non-preemptive critical section (NPCS) protocol [12]. In order to implement an NPCS in an embedded system, developers generally prevent the preemption of a task by locking the OS scheduler, disabling interrupts, or temporarily assigning the highest priority to the current task. Under some real-time operating systems (RTOSes), a non-preemptive task can also be created. The shortcoming of the NPCS is that every task can be blocked by a lower-priority task executing a non-preemptive critical section, even when there is no resource conflict between them. However, the NPCS protocol is widely used due to its simplicity. In addition, the behavior of a task using the NPCS is predictable, since it removes inter-task effects.

The problem arises when a page fault occurs within the NPCS. On a page fault, the corresponding task can be preempted by other tasks if the semaphore S_F is not available. Therefore, the task may fail to protect the shared resources within the NPCS and the system may show behaviors different with the developer's expectations. This problem results from the fact that application developers cannot know the page fault occurrences at run time. Therefore, a resource synchronization protocol is required to solve the problem without involving the developer's concern.

C. Solutions

In order to solve the previously mentioned problems in demand paging within real-time systems, we propose several practical solutions.

1) Hybrid Demand Paging

A practical approach can be adopted to solve the unpredictable timing behavior of demand paging. Since not all applications in high-end embedded systems such as mobile phones have real-time constraints, we divided the applications into two groups: real-time applications and non real-time applications. While the shadowing technique is applied to real-time applications, the demand paging technique is used for the others. To this end, the address space of the main memory is split into two regions: the shadowing region and the demand paging region. The codes to be shadowed are copied from the NAND flash storage to the shadowing region at boot time, and other codes are loaded into the demand paging region on demand during operation. Since all of the time critical tasks are shadowed and, therefore, invoke no page faults, we can avoid the priority inversion problem. Undoubtedly, it will be more beneficial to add the codes of real-time applications into the demand paging region if the timing constraints of the real-time applications will be satisfied, despite the demand paging overhead.

2) Nested Semaphore Acquisition

The deadlock situation can be avoided by using a counting semaphore, which is able to be locked multiple times. The deadlock situation can be identified by examining whether the task with S_F is the same as the task invoking the page fault. In such a case, the count value of the semaphore is incremented and the demand paging manager can access the NAND flash device. After the page fault handling, the count value of the semaphore is decremented.

3) Breakpoint Setting with Page Pinning

We revised the debugger to use the page pinning API function in order to solve the breakpoint problem. The debugger calls the page pinning API when the target instruction for the software breakpoint is not in the physical memory. The page containing the target instruction is loaded into and pinned at the page cache.

4) Semaphore Pre-acquisition

In order to prevent the NPCS problem, we can force a task to acquire S_F before it starts an NPCS, called semaphore pre-acquisition. However, if there is a file access within the NAND flash during an NPCS, the file operation will wait for the semaphore S_F permanently because S_F is already assigned to the task. Therefore, the semaphore should be released just after the task starts an NPCS. We can implement the semaphore pre-acquisition without user's intervention by hooking the APIs for NPCS.

However, this technique involves semaphore-related overheads whenever a task enters into NPCS even though the task generates no page fault. In order to minimize the overhead, we can exempt the shadowed code from performing the semaphore pre-acquisition. However, it is still difficult to predict whether a data page fault will occur at an NPCS.

5) Page Pinning

Another solution for the prevention of preemption in NPCS is to load and pin all of the pages to be accessed in NPCS into the page cache at boot time. The page cache replacement algorithm excludes the replacement of pinned pages and then no page fault occurs in NPCS. Shadowing and pinning are different with respect to granularity. While shadowing is performed by the unit of object file, pinning is processed by the unit of page in order to minimize the page cache waste.

Using static analysis, we are able to identify the pages to be accessed in NPCS regions at compile time. First, NPCS regions within the program code should be identified. An NPCS starts from the program code which disables interrupts, locks the scheduler or assigns the highest priority to the corresponding task, and it ends at the program code which enables interrupts, unlocks the scheduler or assigns the original priority to the task. Second, the pages which are accessed in the NPCS regions should be selected and the pinning page list is generated.

For page pinning, we added a new API which loads the specified pages into the page cache and pins the pages in order to prevent them from being replaced.

D. Non-Preemptive Demand Paging (NPDP)

The page pinning requires a static analysis technique that is able to identify the page accesses in NPCS regions. However, it is significantly difficult to implement an exact static analysis tool. An alternative solution for composing the pinning page list is to use a profile-based technique. A profiler checks the page faults during the target applications execution. If a page fault is generated with the preemption disabled, the profiler inserts the corresponding page into the pinning page list. After profiling, the pages in the pinning page list are pinned in the page cache during system start-up. However, the profile-based pinning may provide an incomplete pinning page list since the profile step does not cover all program paths.

Considering the limitations of the semaphore pre-acquisition and page pinning techniques, we devised the non-preemptive demand paging (NPDP) technique, which enables the page fault handler to preempt the ongoing NAND flash operation in order to support a read operation while the NAND flash memory is servicing the file system. Even if S_F is occupied by the file system, the page fault handler can access the NAND storage without semaphore acquisition.

The NPDP technique has the additional advantage of reducing the page fault latency. Even though we can avoid the page faults in NPCS by pinning the pages of NPCS, the normal page faults outside of the NPCS should wait a maximum of 4 msec (i.e., the multi-block erase time) before acquiring the semaphore. However, the NPDP scheme can handle the page fault without waiting for the semaphore release. In NPDP, FDD is in charge of the resource synchronization instead of the semaphore.

Fig. 5 shows the algorithm used in the NPDP, which first checks the value of semaphore and, if the semaphore is free (the semaphore count is 0), the page fault handler reads the faulted page. If the semaphore is not free (the semaphore count is larger than 0) but the semaphore is occupied by the task which invoked the page fault, the faulted page is loaded in order to avoid the deadlock (lines 4-6).

```
Non-Preemptive Demand Paging

 1: check the semaphore S_F;
 2: if S_F is occupied by the current process   /* deadlock
prevention */
 3:              or S_F is free  then
 4:      increment the semaphore count;
 5:      load the fault page;
 6:      decrement the semaphore count;
 7: else        /* S_F is occupied by the file system */
 8:          if the page fault is invoked within NPCS then
 9:              if the status of flash memory is busy then
10:                  check the ongoing flash operation;
11:                  if the flash operation is erase then
12:                       send the reset command;
13:                  endif
14:                  wait until the status is changed into ready;
15:              endif
16:              save status reg. and I/O buffer;
17:              load the fault page;
18:              restore status reg. and I/O buffer;
19:              if the suspended flash operation is erase then
20:                   send the erase command;
21:              endif
22:          else      /* page fault outside of NPCS */
23:              sleep until S_F is released;
24:              increment the semaphore count;
25:              load the fault page;
26:              decrement the semaphore count;
27:          end if
28: endif
```

Fig. 5 The NPDP algorithm

If the semaphore is occupied by the file system and the page fault is invoked outside of the NPCS, the operation should wait for the semaphore release (lines 22-27). However, when NPCS invoked the page fault, the proposed NPDP technique is used. If read or program operations are executing, NPDP waits until the status register's value is updated to *ready* since a new command cannot be sent to the device while it is busy (lines 9-14). Fortunately, it takes a short amount of time to complete these operations of NAND flash memory. The NPDP only waits for the completion of Step 2 not Steps 3 or 4 in Fig. 3. However, the erase mode is aborted before completion since it requires a maximum 4 msec. We exploit the 'reset' command of the NAND device in order to abort the erase operation. When the device is in the *busy* state during the read, program or erase mode, the reset operation aborts these operations. The reset command requires 5 μsec, 10 μsec and 500 μsec to abort read, program and erase modes, respectively. Fig. 6 shows the aborting operations of the NPDP. Task 1 issued a NAND operation and was preempted by task 2. While executing task 2, a page fault occurs and the NPDP services the page fault.

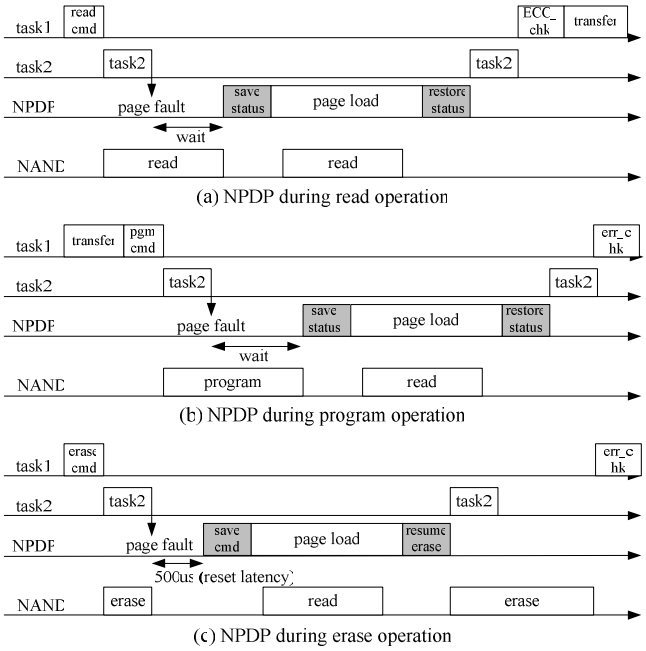

Fig. 6 Aborting NAND operations in NPDP

After the NAND device is changed to the ready status, the device is available for reading or programming data. Since a NAND device has only one status register and one I/O buffer, the read command generated by the NPDP will change the status register and the I/O buffer. Therefore, the NPDP should save the status register and the I/O buffer data before it accesses the NAND device (line 16). After the page fault handling, the NPDP restores the saved values of the status register and the I/O buffer (line 18), and the preempted task can finalize its NAND operation. When the erase mode is aborted, the NPDP should retry the preempted erase operation after page fault handling (line 20).

The NPDP technique can also be used for the normal page faults in the preemptable region. However, NPDP imposes an I/O overhead because it should save and restore the status register as well as the I/O buffer. Moreover, the erase operation should be restarted after the page fault handling. The page fault latency (*PFL*) in NPDP can be expressed as follows:

$$PFL = t_{abort} + t_{save} + t_{read} + t_{cache} + t_{resume} \quad (2)$$

where t_{abort}, t_{save} and t_{resume} represent the command aborting time for the erase command (or the waiting time for read/program completion), the status saving time, and the time for resuming the aborted operation (or the status restoring time for read/program completion), respectively. Therefore, it is better to use NPDP only when a page fault occurs in NPCS.

There is another critical problem when NPDP is used within a preemptable region. During the NPDP, interrupts should be

disabled in order to prevent task scheduling. However, the system is then not able to provide timely service for a time critical task. For example, when the page fault handling reads a 4 KB page after aborting the erase command, the interrupt is disabled during at least 800 μsec (500 μsec to abort the erase operation and 300 μsec to read the page). If the relative deadline of a real-time task, triggered by an external interrupt, is smaller than 800 μsec, the task could miss its deadline.

V. EXPERIMENTS

In order to estimate the effect as well as the overhead of demand paging, we applied the proposed demand paging technique to a 140 MHz ARM926EJ-based CDMA mobile phone. The non-preemptive page fault handling technique was used. The phone is equipped with a 3 Gb NAND flash memory and 64 MB DRAM. The total program code size is 29 MB. If the full shadowing technique is used, 45% of DRAM should be allocated for code shadowing. For demand paging, we allocated 5 MB of DRAM memory to the page cache (PC). The page size is 4 KB. Only 6 MB of the program code was shadowed into DRAM, and the rest of the 23 MB code was demand-paged into the demand paging region (DPR). The shadowed region (SR) includes the call processing protocol stack and time-critical device drivers. Consequently, 11 MB of the DRAM memory is allocated for code memory, and the memory space for application code can be reduced by 62% in comparison to that of the full shadowing technique.

We compared the boot times of four different configurations of demand paging, as shown in Table I. The boot time is composed of the loading time for the shadowed program image and the total execution time of boot-related programs. While the loading time of a program image was 0.27 sec in configuration I, 1.35 sec were needed for configuration IV, which had a larger shadowing region. As a larger PC and SR are utilized, the execution time decreases due to the smaller demand paging overhead. Consequently, the total boot times under different demand paging configurations are similar.

TABLE I
DEMAND PAGING DURING BOOT TIME.

Configuration (MB)					Boot Time (sec)		
No.	SR	DPR	PC	Mem Total	Load	Exec.	Total
I	6	26	2	8	0.27	14.37	14.64
II	6	26	5	11	0.26	14.18	14.44
III	6	26	20	26	0.29	13.99	14.29
IV	22	10	5	27	1.35	13.40	14.75

We measured the page fault latency while executing real phone applications on the target platform. An MPEG4 video recording program and an MP3 audio player program were used, and both of the two applications were demand-paged and used the file system. In addition, in order to make the resource conflict situation, we executed a background program which repeats file reading, deleting and writing in a period of 700 msec.

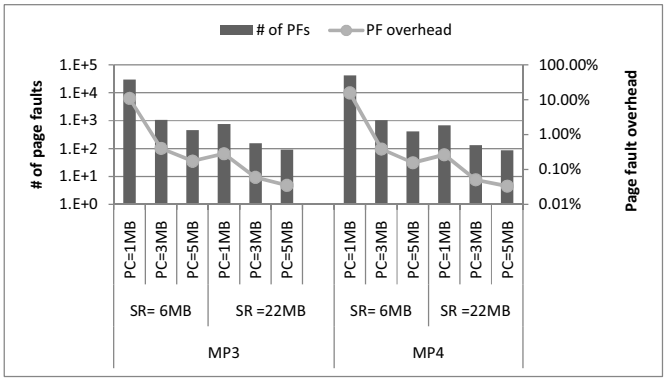

Fig. 7 Page fault counts and total page fault overheads.

Fig. 7 shows the number of page faults and the page fault overheads at various shadowing region and page cache sizes. As the sizes of the PC and SR increase, the number of page faults and the page fault overhead decrease. When the shadowing region was 6 MB and the page cache was 5 MB, the page fault overheads of the MP3 and MPEG4 applications were only 0.17% and 0.16% of the total application execution times, respectively. This means that the page fault overhead is too small to be noticeable for a reasonable page cache size.

Table II shows the number of page faults in NPCS, the number of deadlocks and the number of non-preemptive demand pagings under different SR and PC configurations. We can know from the results that there can be frequent page faults in NPCS when the page cache is small, therefore, the NPDP scheme is indispensable for embedded systems.

TABLE II
NUMBERS OF NPCSS, DEADLOCKS AND NPDPS

App.	SR (MB)	PC (MB)	PF in NPCS	Deadlock	NPDP
MP3	6	1	585	8	11
		3	1	1	0
	22	1	2	0	0
MP4	6	1	1416	43	1
		3	5	0	0
	22	1	6	0	0

We also observed the different behaviors resulting from different page sizes. When we use a page size of 1 KB, instead of 4 KB, the number of page faults increases by 85% in the MPEG4 program. However, the increase in page fault count does not indicate an increase in page fault overhead. The total page fault overhead increases only 1% since the 1 KB page requires a small time for page transferring.

VI. CONCLUSIONS

The NAND-based demand paging can significantly reduce the required memory space for consumer electronics. However, in order to simultaneously service both the demand paging request and the file system request, a resource synchronization

technique for NAND flash memory is needed. One simple technique is the exploitation of the semaphore of NAND storage to control the simultaneous accesses to the shared resource. However, using the semaphore can result in a preemption of the non-preemptive critical section (NPCS).

We proposed the non-preemptive demand paging (NPDP) technique in order to solve the NPCS issue. With the NPDP technique, the flash device driver both saves and restores the status registers and the I/O buffer of NAND flash memory for resource synchronization. Experimental results showed that the NAND-based demand paging can reduce the memory requirement with a negligible timing overhead.

Our work can be extended in several directions. Multimedia application requires a sustained bandwidth in order to guarantee the quality-of-service (QoS). Therefore, a real-time resource reservation scheme for NAND storage, providing timely and guaranteed access to system resources, is required for multimedia applications. Hence, there is a need to study the distribution of NAND storage bandwidth between demand paging and file systems.

REFERENCES

[1] J.-H. Lin, Y.-H. Chang, J.-W. Hsieh, T.-W. Kuo, and C.-C. Yang. A NOR Emulation Strategy over NAND Flash Memory. In *Proc. the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (*RTCSA*), pages 95-102, 2007.

[2] C. Brown and R. Hasbun. Simultaneous code execution and data storage in a single flash memory chip for real time wireless communication systems. In *Proc. the 40th Midwest Symposium on Circuits and System*, pages 740–745, 1997.

[3] M. Gefen, S. Zernovizky, and A. Ban. System and method for enabling non-volatile memory to execute code while operating as a data storage/processing device. US Patent 7032081, 2006.

[4] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim. A low cost memory architecture with NAND XIP for mobile embedded systems. In *Proc. CODES+ISSS'03*, pages 138–143, 2003.

[5] C. Park, J. Kang, S. Park, and J. Kim. Energy-aware demand paging on NAND flash-based embedded storages. In *Proc. International Symposium on Low Power Electronics and Design*, pages 338–343, 2004.

[6] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler assisted demand paging for embedded systems with flash memory. In *Proc. International Conference on Embedded Software*, pages 114–124, 2004.

[7] Y. Joo, Y. Choi, C. Park, S. W. Chung, E.-Y. Chung, and N. Chang. Demand Paging for OneNAND Flash eXecute-In-Place. In *Proc. CODES+ISSS'06*, pages 229–234, 2006.

[8] Samsung Electronics. OneNAND Features and Performance. http://www.samsung.com/Products/Semiconductor/OneNAND. 2005.

[9] S. A. Belogolov, J. Park, J. Park, and S. Hong, Scheduler-Assisted Prefetching: Efficient Demand Paging for Embedded Systems. In *Proc. the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (*RTCSA*), pages 111-119, 2008.

[10] J. In, I. Shin, and H. Kim. SWL: A Search-While-Load Demand Paging Scheme with NAND Flash Memory. In *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 217–225, 2007.

[11] S. Hyun, S. Lee, H. Bahn, and K. Koh. Vectored read: Exploiting the read performance of hybrid NAND flash. In *Proc. the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 177-184, 2008.

[12] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

BIOGRAPHIES

Wangyu Kim received the B.S. degree in Information and Telecommunication engineering from Korea Aerospace University, Korea in 2004. Since 2004, he is an engineer of Samsung Electronics CO., LTD. Korea. He is also currently a Master student in the School of Information and Communication Engineering, Sungkyunkwan University. His research interests include embedded software, low-power systems, file systems and flash memory.

Dongkun Shin (M'08) received the B.S. degree in computer science and statistics, the M.S. degree in computer science, and the Ph.D. degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000 and 2004, respectively. He is currently an Assistant Professor in the School of Information and Communication Engineering, Sungkyunkwan University (SKKU). Before joining SKKU in 2007, he was a senior engineer of Samsung Electronics Co., Korea. His research interests include embedded software, low-power systems, computer architecture, and multimedia and real-time systems.