SWL: A Search-While-Load Demand Paging Scheme with NAND Flash Memory

Jihyun In Ilhoon Shin Hyojun Kim

Samsung Electronics Co., Ltd.
{jh_in, ilhoon.shin, zartoven}@samsung.com

Abstract

As mobile phones become increasingly multifunctional, the number and size of applications installed in phones are rapidly increasing. Consequently, mobile phones require more hardware resources such as NOR/NAND flash memory and DRAM, and their production cost is accordingly becoming higher. One candidate solution to reduce production cost is demand paging using MMU. However, demand paging causes unpredictably long page fault latency, and as such mobile phone manufacturers are reluctant to deploy this scheme. In this paper, we present a method that reduces the long latency of page faults by performing page fault handling in a parallelized manner, considering the characteristics of NAND-Type flash memory. We also discuss how to modify the existing page cache replacement policies so that they can exploit the benefits of the parallelized page fault handler. Experimental results show that the parallelized page fault handler improves the worst case latency of page faults significantly, by up to roughly 20%, and that the modified page cache replacement policies improve both the average and worst instruction fetch time.

Categories and Subject Descriptors B.3.2 [**Memory Structures**]: Design Styles – Virtual memory; D.4.2 [**Operating Systems**]: Storage Management – Secondary storage, Virtual memory

General Terms Algorithms, Measurement, Performance, Design

Keywords Demand paging, Parallelization, Page fault handler, Page replacement, NAND flash memory

1. Introduction

Since first appearing in the early nineties, mobile phones have become dramatically smaller, lighter, and multi-functional. At present, it is common for users to be able to take pictures, play games, and listen to mp3 music with their mobile phones. As mobile phones become increasingly multi-functional, however, the number and size of applications installed in the phones are rapidly increasing. Consequently, they require more hardware resources such as NOR flash memory and DRAM, and their production cost is rising accordingly.

There are generally two methods to execute applications in mobile phones. The first is called eXecute-In-Place on NOR flash memory (NOR-XIP) (Figure 1(a)). In NOR-XIP, program code is stored in NOR flash memory and executed in place without loading the code to DRAM. As NOR flash memory stores all the application code, the capacity of the memory should increase in proportion to the size of applications, thus inducing higher production cost. We predict that NOR-XIP will become less popular as mobile phones become more multi-functional, because the byte-price of NOR flash memory is much higher than that of NAND flash memory or DRAM (Table 1).

The second method, known as Shadowing, does not use NOR flash memory for code storage. Instead, it stores the application code to NAND-Type flash memory such as OneNAND and loads it to DRAM at booting time (Figure 1 (b)). By eliminating the expensive NOR flash memory, Shadowing allows for lower production cost than NOR-XIP. However, the production cost is expected to increase with application size. Long booting time is another concern of Shadowing.

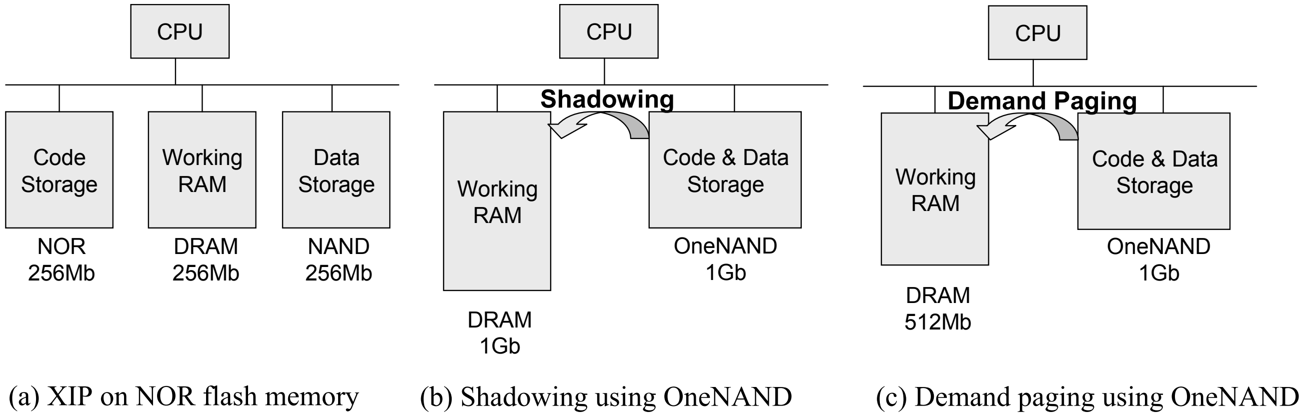
As both NOR-XIP and Shadowing have the common problem of high production cost, mobile phone manufacturers are searching for cost-effective solutions [1], [2]. One candidate is demand paging using MMU (Memory Management Unit). The demand paging scheme stores the application code to a cheap secondary storage and loads the required pages to the main memory on demand. For example, in a mobile phone environment, the application code is stored to OneNAND and loaded on demand to DRAM (Figure 1(c)). Demand paging makes it possible to execute large software with limited DRAM, and contributes to reduced production cost. The capacity of OneNAND should additionally increase to store the applications, but the cost is not severe because the byte-price of NAND flash memory is much cheaper than that of NOR flash memory and DRAM, respectively, as seen in Table 1. For example, the price of NAND flash memory is about 38% that of NOR flash memory for 512 Mb (64Mbytes) memory, and about 27% that of DRAM for 1Gb (128Mbytes) memory.

Table 1 : Memory components price in 2006

	NAND	NOR	DRAM
256Mb	$2.67	$5.42	$2.72
512Mb	$3.67	$9.75	$3.62
1Gb	$3.87	N/A	$14.09
2Gb	$5.92	N/A	$83.88
8Gb	$9.64	N/A	N/A

IDC Forecast: Memory, Worldwide, 2000-2010 (3Q06 Update)

LCTES'07 June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006…$5.00.



(a) XIP on NOR flash memory (b) Shadowing using OneNAND (c) Demand paging using OneNAND

Figure 1: The methods to execute application in mobile phones

The drawback of demand paging is a long page fault latency that is caused by page cache misses. Page cache miss refers to the case where the main memory does not hold a requested page. In the case of a page cache miss, the requested page should be loaded from a slow secondary storage. As this page loading time is much longer than the main memory access time, it is critical to the system performance to reduce both the number of page cache misses and the latency of page cache misses. Most previous studies have focused on reducing the number of page cache misses by deploying an efficient page replacement policy such as Clock[3][4], FIFO with second chance [5], and EELRU [6], etc. However, methods to reduce the latency of page cache misses have seen relatively little attention. In this paper, we present a method that reduces the latency of page cache misses by handling a page fault in a parallelized manner considering the characteristics of NAND-Type flash memory.

NAND-Type flash memory generally has a stopover buffer between the host interface and the NAND cell array. Data are transferred via a stopover buffer such as a data register between the host interface and the cell array with distinct commands. For example, a read operation from NAND proceeds in two steps: from the NAND cell array to the stopover buffer and from the stopover buffer to the host interface [7]. Interestingly, the address of the host memory is not required when loading data from the NAND cell array to the stopover buffer. Thus, in the event of a page cache miss, free page allocation and page loading to the stopover buffer can be processed simultaneously. This is one of the major differences between NAND-Type flash memory and HDD. Note that the host interface of HDD does not provide two step operations. The operations in HDD are processed in a single step: from/to HDD to/from host memory[1]. Parallelization using a two step operation can result in reduced page fault latency.

The existing page cache replacement policies were designed with single step operations of HDD. Thus, mobile phones that can use two step operations need to be modified to exploit the parallelization benefits. In the paper, we discuss how to modify the existing page cache replacement policies with the examples of Clock and FIFO with Second Chance.

The remainder of this paper is organized as follows. Section 2 explains the architecture and the operations of OneNAND, which is a kind of NAND-Type flash memory. Section 3 describes the SWL (Search-While-Load) demand paging scheme, which consists of a parallelized page fault handler and modified page cache replacement polices. Performance evaluation results based on an implementation and measurement are presented in Section 4. Finally, we conclude in Section 5.

2. OneNAND Flash Memory

Figure 2 describes the architecture of OneNAND. As seen in the figure, OneNAND mainly consists of NAND flash cell array, an internal buffer including DataRAM, error correction logic, and a host interface. The cell array stores data in a non-volatile manner, DataRAM is a stopover buffer between the NAND cell array and the host interface, and the host interface transfers data from/to DataRAM to/from the host memory. As described in Section 1, I/O operations are processed in two steps. We explain the READ operation in more detail through the following example.

The READ operation is initiated by a load command with the NAND cell address, which notifies the start of READ to OneNAND. Immediately after the load command is issued to OneNAND, OneNAND starts to load the data from the NAND cell to DataRAM. The status register of OneNAND is marked busy while the loading is processing. After the loading finishes, the status bit is cleared and the data then resides in DataRAM. This data loading usually takes about 30 us for a 2 Kbytes page [8]. In order to transfer data from DataRAM to the host memory, memory copy should be performed with the destination address of the host memory. The transfer time between DataRAM and the host memory is dependent on the clock frequency of OneNAND and the host processor. When the clock frequency of OneNAND is 54 MHz and the host processor supports 54 MHz, 2 Kbytes transference takes about 18.96 us in sync burst mode, theoretically [8].

[1] HDD has cache memory inside it that can be used as a stopover buffer. Thus, if the host interface supports two step operations, the parallelized demand paging scheme can also be implemented in HDD.

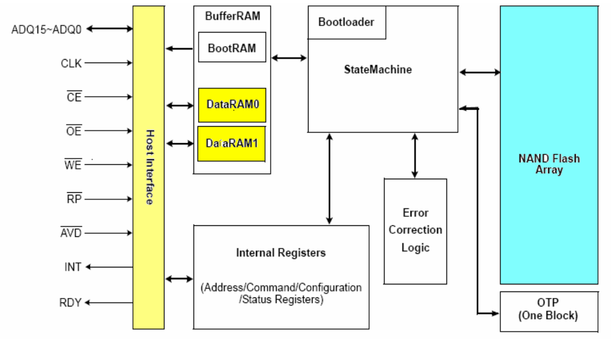


Figure 2: The architecture of OneNAND

An interesting feature of OneNAND I/O operations is that the destination of the host memory is not needed when loading data from the NAND cell array to DataRAM. This makes it possible to parallelize some page fault handler jobs such as free page allocation when loading data from the NAND array. By exploiting this parallelization, the new demand paging scheme can reduce the page fault latency.

3. SWL Demand Paging Scheme

In this section, the SWL (Search-While-Load) demand paging scheme is described. In the first subsection, we explain the parallelized page fault handler and its benefits. The modified page cache replacement policies, which exploit the benefits of the parallelized page fault handler, are presented in section 3.2.

3.1 The Parallelized Page Fault Handler

When a page fault occurs as a result of a page cache miss, the page fault handler should perform the following jobs.

- ***PreProcess***: Page fault hander saves the current register set to the stack and finds out the source address of the faulted page.
- ***GetFreeframe***: Page fault handler searches for a free frame in the main memory to load the faulted page from the secondary storage. If there is no free frame in the main memory, it selects a victim frame with a page cache replacement policy and evicts the victim[2].
- ***LoadPage***: Page fault handler loads the faulted page from the secondary storage to the allocated frame.
- ***UpdateMMU***: Page fault handler updates page table and TLB (Translation Look-aside Buffer) to reflect the newly added page and the evicted page, if any exist.
- ***PostProcess***: Page fault handler restores the saved register set.

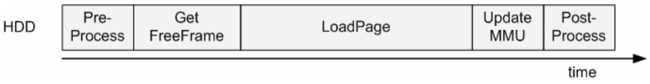


Figure 3: Sequential page fault handler with HDD

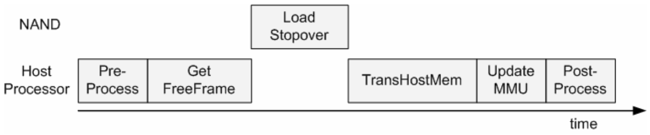


Figure 4: Sequential page fault handler with NAND

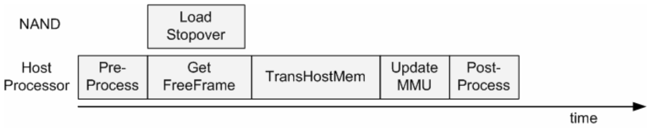


Figure 5: Parallelized page fault handler with NAND

The existing page fault handler, which is designed in an environment where HDD is used as secondary storage, performs the above jobs in a sequential manner, as illustrated in Figure 3. First, the page fault handler saves the current register set and finds out the source address of the faulted page through the page table lookup (*PreProcess*), and allocates a free frame to load the faulted page (*GetFreeframe*). It then loads the faulted page to the allocated frame using the source address and the destination address (*LoadPage*). After *LoadPage* finishes, it updates MMU to reflect the changes (*UpdateMMU*). Finally, it restores the saved register set (*PostProcess*). In an environment where *LoadPage* is processed in a single step, it is not possible to parallelize *LoadPage* with *PreProcess* or *GetFreeframe*, because *LoadPage* needs both the source address and the destination address and therefore depends on *PreProcess* and *GetFreeframe*.

However, in mobile phones where NAND-Type flash memory is used as secondary storage, the above execution sequence is different. As described in section 2, NAND-Type flash memory generally has a stopover buffer. The read operation is processed in two steps- from the NAND cell array to the stopover buffer and from the stopover buffer to the host memory. In other words, a *LoadPage* is divided into two sub-jobs, loading data from the NAND cell array to the stopover buffer (*LoadStopover*) and transferring the data from the stopover buffer to the host memory (*TransHostMem*). Thus, if the page fault handling is processed in a sequential manner, the execution sequence of the page fault handler is modified as shown in Figure 4. In the figure, note that *LoadStopover* is performed by the NAND flash memory, while the others are performed by the host processor.

- ***LoadStopover***: Page fault handler loads the faulted page from the source address to the stopover buffer.
- ***TransHostMem***: Page fault handler transfers the page from the stopover buffer to the allocated frame.

[2] In NAND-Type flash memory, a write operation is relatively slower than a read operation. Also, the write operation to update data in NAND-Type flash memory may accompany an erase operation, which takes about 2ms per 128 Kbytes block. Therefore, we do not replace dirty pages. Only code and clean data pages are paged out.

The important feature in the page fault handler with NAND-Type flash memory is that *LoadStopover* does not need the destination address, i.e., the address of the allocated frame. It only needs the source address of the faulted page. This means that *LoadStopover* does not have to follow *GetFreeframe*. As the executers of *LoadStopover* and *GetFreeframe* are different and *LoadStopover* is not dependent on *GetFreeframe*, it is possible to perform *LoadStopover* and *GetFreeframe* simultaneously; *LoadStopover* by the NAND-Type flash memory and *GetFreeframe* by the host processor [3]. The parallelized page fault handler, whereby *LoadStopover* is performed with *GetFreeframe* in parallel, is described in Figure 5. From the figures, we can see that the page fault latency is reduced by the parallelized structure.

The effect of the parallelized page fault hander is computed as follows. We define the latency of each job as *t(job)*. For example, the latency of *PreProcess* is denoted as *t(PreProcess)*.

First, we compute the latency of the sequential page fault hander. As the sequential page fault handler performs the jobs in a sequential manner, *t(SeqHandler*) is computed by simply adding the latency of each job, as seen in (1).

$$\begin{aligned} t(\,SeqHandler\,) = t(Pr\,e\,Pr\,ocess\,) + t(\,GetFreeframe\,) + \\ t(\,LoadStopover\,) + t(\,TransHostMem\,) + \\ t(UpdateMMU\,) + t(\,Post\,Pr\,ocess\,) \end{aligned} \tag{1}$$

Meanwhile, as the parallelized page fault handler can perform *GetFreeframe* in parallel with *LoadStopover*, the shorter latency between *t(GetFreeframe)* and *t(LoadStopover)* can be hidden. Thus, the latency of the page fault handler *t(ParHandler)* is computed as (2).

$$\begin{aligned} t(\,ParHandler\,) = t(Pr\,e\,Pr\,ocess\,) + t(\,TransHostMem\,) + \\ max\,(\,t(\,LoadStopover\,),\;t(\,GetFreeframe\,)\,) + \\ t(UpdateMMU\,) + t(\,Post\,Pr\,ocess\,) \end{aligned} \tag{2}$$

Therefore, the reduced latency of the parallelized page fault handler *t(ParEffect)* is computed as (3).

$$t(\,ParEffect\,) = min(\;t(\,LoadStopover\,),\;t(\,GetFreeframe\,)\,) \tag{3}$$

Equation (3) shows that the benefit of the parallelized page fault handler depends on the performance of the NAND-Type flash memory, which determines *t*(*LoadStopover*) and the page cache replacement policy, which in turn determines *t*(*GetFreeframe*).

3.2 Modifying Page Cache Replacement Policies

Equation (2) of section 3.1 shows that the time to find a free frame *t(GetFreeframe)* can be hidden almost up to the time that is required to load a page to the stopover buffer *t(LoadStopover)*, i.e., about 30 us in OneNAND. As the existing page cache replacement polices such as Clock or FIFO with second chance were designed under the assumption that *t(GetFreeframe)* can not be hidden, they can be more optimized by exploiting this spare time. In this subsection, we describe how to modify the existing page cache replacement policies so that they utilize the benefits of the parallelized page fault handler, with the example of Clock and FIFO with Second Chance.

Clock is a kind of FIFO replacement policy. Basically, the first-in page becomes the first-out page in a page cache. The difference between Clock and FIFO is that the Clock scheme gives one more chance to recently referenced pages using reference bits. When a page is referenced, the reference bit of the page is set. The reference bit is periodically reset and a page whose reference bit is cleared is selected as a victim. The Clock scheme can be implemented with a circular queue with one or two arms. If a circular queue with two arms is used, the front arm resets the reference bit and the back arm evicts a page. The Clock scheme is simple but shows better performance compared to FIFO [3], [4].

The drawback of the Clock scheme is that *t(GetFreeframe)* varies according to the size of the page cache and system circumstances. For example, if the reference bits of all pages are set, a back arm should move around the circular queue, which results in long page fault latency. To prevent this, Babaoglu and Joy allocated several free frames beforehand. If the number of allocated free frames is below the predefined threshold, additional free frames are allocated in advance [4]. This helps to shorten *t(GetFreeframe)*, but can increase the number of page cache misses by decreasing the available size of the page cache.

The above dilemma of the Clock scheme can be resolved by our Parallelization-Aware Clock Scheme (PA-Clock), which exploits the spare time created by the parallelized page fault handler. As *t(GetFreeframe)* is hidden almost up to *t(LoadStopover)*, PA-Clock performs a free frame search on-the-fly without pre-allocation. If a victim frame is found within *t(LoadStopover)*, PA-Clock operates in the same manner as the original Clock scheme. However, if the victim page has not been found within *t(LoadStopover)*, PA-Clock instantly evicts the page pointed by the front arm, regardless of its reference bit. Thus, in PA-Clock, *t(GetFreeframe)* is always less than *t(LoadStopover)*; namely, *t(GetFreeframe)* is always completely hidden. This can make the distribution of the page fault latency constant and improve the worst case page fault latency. At the same time, as PA-Clock operates in the same manner as the original Clock scheme within *t(LoadStopover)*, the average page cache miss ratio of PA-Clock is similar to that of the original Clock scheme, as will be shown in the performance evaluation section.

FIFO with Second Chance (FIFO-SC) implemented in Mach2.5 is another kind of NUR (Not Used Recently) replacement policy [5]. Instead of the circular list used in Clock, it maintains two FIFO lists of fixed size, a valid list and an invalid list. The valid list keeps recently referenced pages, and the pages of this list are marked as valid in the page table. The invalid list keeps pages that have not been referenced for a period of time, and the pages of this list are marked as invalid in the page table, which means that referencing them generates a page fault despite that they actually reside in the main memory. This kind of fault is called a false fault. When a false fault occurs, the referenced page is changed to valid in the page table and promoted to the valid list. At this time, in order to maintain a constant valid list size, the first-in page of the valid list is downgraded to the invalid list and changed to invalid in the page table. When the page cache is full and a victim must be extracted, the first-in page of the invalid list is always selected as a victim. Because the lists are ordered by FIFO, the victim search time, *t(GetFreeframe)*, is always constant and short.

[3] If it is guaranteed that context switch does not happen during page fault handling, *UpdateMMU* can also be performed with *LoadStopover* in parallel.

In FIFO-SC, *t(GetFreeframe)*, which can be hidden by the parallelized page fault handler, is quite short and thus the benefits of the parallelized page fault handler is not so great. On the contrary, the latency of a false fault, which cannot be hidden by the parallelized page fault handler[4], is relatively long, because it includes various list managing operations. In our experimental configuration, the latency of a false fault in FIFO-SC was measured to be 14.3 us on average. In order to optimize FIFO-SC with the parallelized page fault handler, it is necessary reduce the latency of false faults, even at the cost of increasing *t(GetFreeframe)*.

For this, PA-FIFO-SC (Parallelization-Aware FIFO with SC), presented in this paper, delays downgrading the first-in page of the valid list to the next true page fault, i.e., the next page cache miss. In other words, when a false fault occurs, PA-FIFO-SC promotes the referenced page to the valid list, and the downgrade of the first-in page of the valid list is performed at the next page cache miss. As we have spare time up to *t(LoadStopover)* when a page cache miss occurs, the delayed downgrade is completely hidden and does not increase the latency of a true page fault. In our experiment, the latency of a false fault was reduced from 14.3 us to 11.2 us by PA-FIFO-SC, without increasing the latency of true page faults. The next section presents performance evaluation results in more detail.

4. Performance Evaluation

4.1 Performance of the parallelized page fault handler

4.1.1 Experimental Configuration

To evaluate the performance of the parallelized page fault handler, we measured the page fault latency in a sequential manner and a parallelized manner, respectively. We implemented the page fault handler on a Nucleus operating system [9] and ported it in a commercial target platform, TI OMAP 5912 OSK [10]. The target platform has a 192MHz ARM 926EJ-S™ processor [11][5], which supports MMU, and has a 16Kbytes instruction cache and an 8Kbytes data cache. The MMU of the ARM926EJ-S™ processor supports 1Kbyte, 4Kbytes, and 64Kbytes page sizes; we choose 1Kbyte as a unit of demand paging. 66MHz Samsung OneNAND KFG1G16Q2M [8] was used as a secondary storage, the block size of which is 128Kbytes and the page size is 2Kbytes.

4.1.2 Experimental Results

Figure 6 shows the page fault latency of the sequential and the parallelized page fault handler. We measured the page fault latency with both Clock and FIFO with SC. The X-axis denotes each demand paging configurations, and the Y-axis denotes the page fault latency whose time unit is us. In the graph, each stick consists of several parts, which correspond to the latency of sub-jobs of the page fault handler. In the Clock scheme, the latency of *GetFreeframe* is dependent on the page list walk count, which searches the page list in order to find an unreferenced page. In this experiment, the page list walk count was 2.

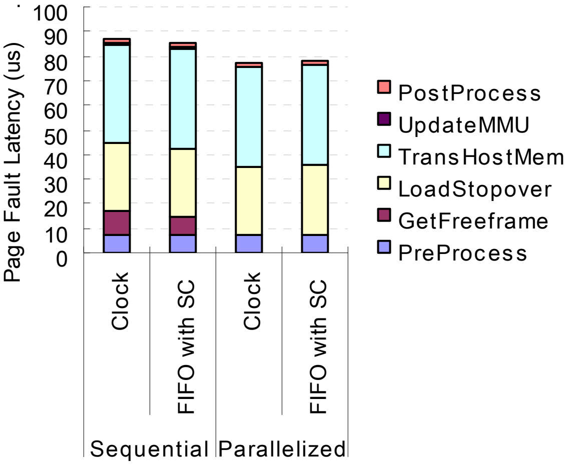


Figure 6: The page fault latency in various demand paging configurations

The figure shows that the proposed parallelized page fault handler was effective for both Clock and FIFO with SC. In the Clock scheme, the parallelized page fault handler reduced the page fault latency from 87 to 77 us, comprising an approximately 11.5 % gain. In the FIFO with SC scheme, the latency was reduced from 85 to 78 us, or roughly 8.23 % gain. The graph shows that this improvement was mainly attained by hiding *GetFreeframe* of the sequential method. In the parallelized manner, *GetFreeframe* can be completely hidden by performing *LoadStopover* in parallel.

It is worthwhile to note that in the Clock scheme, the page list walk count to find a victim page frame may considerably increase and thereby the time to obtain a free frame can be very long, whereas the page list walk count is constant in FIFO with SC. This means that the latency of *GetFreeframe* may be longer in the Clock scheme than the result of Figure 6, where the page list walk count was 2. In order to study the influence of the page list walk count, we measured the page fault latency of the Clock scheme while varying the page list walk count. The results are shown in Figure 7.

The figure shows that the page fault latency steadily increases according to the page list walk. For the sequential approach, the page fault latency is 86.1 us when the page list walk count is 1, but it is increased to 137.95 us when the walk count is 60. Using the parallelized approach, the page fault latency is constant until the walk count reaches 13, because the overhead of the page list walk is hidden up to *t(LoadStopover)*. As the page list walk count exceeds 13, however, the latency steadily increases, similar to the sequential method. The time exceeding *t(LoadStopover)* could not be hidden. Two important observations are made from Figure 7. First, the parallelized page fault handler is more effective as the time to search a victim becomes longer. The performance gain of the parallelized handler reaches roughly 20% when the page list walk count is 13. Second, the Clock scheme still caused unpredictably long page fault latency even with the parallelized page fault handler. The scheme needs to be modified so as to have bounded page fault latency by exploiting the benefits of the parallelized page fault handler.

[4] As a false fault does not include actual device I/O, the latency of a false fault is not hidden.

[5] Because ARM926EJ-S™ does not support the reference bit in hardware, the reference bit was emulated using the valid bit in page table entry bits [1:0] [11].

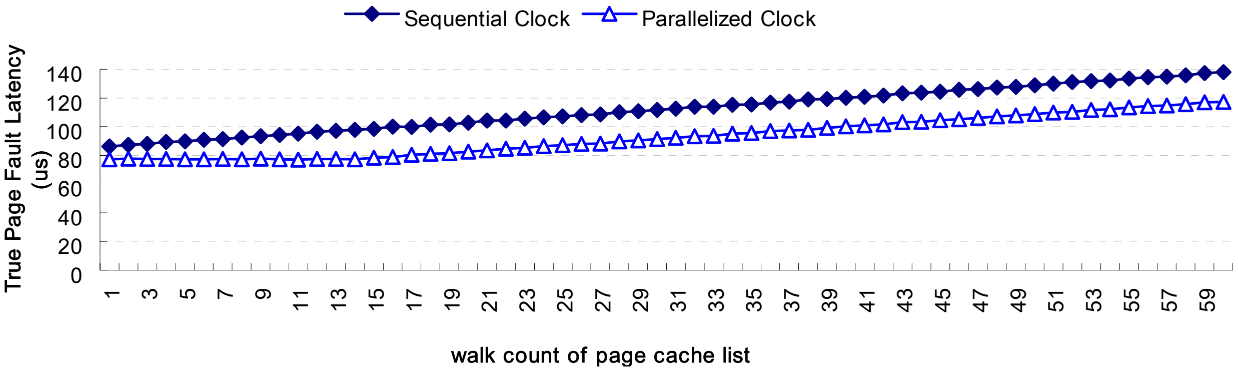


Figure 7: The page fault latency of the Clock scheme according to the page list walk count

4.2 The Performance of PA-Clock and PA-FIFO with Second Chance

4.2.1 Experimental Configuration

In this section, we describe the results of the performance evaluation of PA-Clock and PA-FIFO-SC. Unlike the experiment involving the parallelized page fault handler, which was performed by implementation and measurement, the evaluation of the page cache replacement policies was mainly done by a trace-driven simulation, because it is difficult to find realistic applications of mobile phones. Table 2 describes the detailed parameters of the simulator, which is modeled toward an ARM9 processor having 200MHz CPU clock frequency and 100MHz DRAM clock frequency. In the table, instruction cache fetch time and the instruction cache line refill time were calculated from the clock frequencies of CPU and DRAM, while the loading time from OneNAND was measured from the target of section 4.1. TLB fetch time was approximated from the CPU clock frequency.

Table 2 : The primary parameters of the simulator

Component		Input parameter
I-Cache	Set-Associativity	4-way
	Words per Line	8 words
	Replacement Policy	Round Robin
	Fetch Time	10 ns / word
	Size	8KB
TLB	Sets	16 sets
	Set-Associativity	2-way
	Replacement Policy	Round Robin
	Fetch Time	5 ns
Page Cache	Page Cache Size	128KB
	Page Size	1KB
	Instruction Cache Line Refill Time	200 ns
OneNAND	1KB loading to page cache	68.51 us

As the input trace for the simulation, three SPEC CPU 2000 traces [12], Crafty, Eon, and Bzip2, were used. A detailed description of the traces is provided in Table 3.

Table 3 : Description of the SPEC-2000 traces

	Trace Stats		Description
	References	Instructions	
Crafty	103963082	6707680 (64.5%)	Chess Game
Eon	10485760	5751240 (54.8%)	Computer Visualization
Bzip2	103962327	7013242 (67.5%)	Compression / Decompression

4.2.2 Experimental Result

As shown in section 4.1, the page fault latency of the Clock scheme varies according to the page list walk count. In order to bound the page fault latency, we developed the PA-Clock scheme. Figure 8 shows the effectiveness of PA-Clock. We compared the page fault latency of PA-Clock with Clock under varying page list walk count. The parallelized page fault handler was used in both schemes. The figure shows that, as expected, the page fault latency of the PA-Clock scheme is constant regardless of the page list walk count, whereas that of the Clock scheme gradually increases according to the page list walk count. While the latency of the Clock increases from 77.3 to 117.31 us, that of the PA-Clock is fixed at 77.5 us. The PA-Clock scheme maintains constant page fault latency by instantly evicting a victim page regardless of its reference bit, if a victim is not found within *t(LoadStopover)*.

One reasonable concern about the PA-Clock scheme is that it may increase the number of page cache misses and hurt the average performance, because it may evict a page whose reference bit is set if the victim search time exceeds *t(LoadStopover)*. In order to investigate the influence of the PA-Clock scheme on the average performance, we evaluated the page cache miss ratio of Clock and PA-Clock through a trace-driven simulation with varying the size of the page cache from 64KB to 512KB. The parallelized page fault handler was used in both schemes. Table 4 depicts the results for the traces. The results show that Clock and PA-Clock deliver a

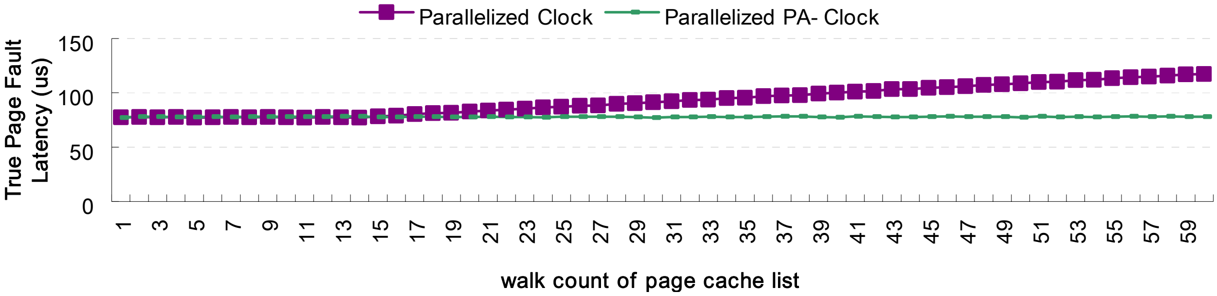


Figure 8: The page fault latency of Clock and PA-Clock

similar page miss ratio for all sizes of page cache and for all traces. For example, in the crafty trace, the page miss ratio of the Clock scheme is 0.000746 on the 128KB page cache, while that of the PA-Clock scheme is 0.000750. The other page cache sizes show similar results. The results of the experiments demonstrate the effectiveness of PA-Clock. It did not hurt the average performance by operating in the same manner as the Clock scheme within *t(LoadStopover)*, while at the same time it improves the worst case page fault latency significantly.

Meanwhile, FIFO with SC has an overhead of long false fault latency, which is not hidden by employing the parallelized structure. In order to reduce the false fault latency, we suggested using PA-FIFO with SC. Figure 9 shows the effectiveness of PA-FIFO with SC. We compared the false fault latency of PA-FIFO with SC with FIFO-SC. The parallelized scheme was used in both schemes. The figure shows that, as expected, the PA-FIFO with SC scheme reduces the false fault latency from 14.3 to 11.2 us, comprising a roughly 21.7% gain. The improvement was mainly achieved in the reference step where the page list management is performed. By delaying downgrade of the first-in page of the valid list to the next page cache miss, the latency of the reference step was reduced from 5.8 to 2.5 us.

In order to investigate the influence of the reduced false fault latency on the average performance, we compared the average instruction fetch time of PA-FIFO-SC with FIFO-SC through a trace-driven simulation. Figure 10 – Figure 12 show the results of crafty, eon, and bzip2 traces, respectively. The X-axis denotes the size of the page cache, which is varied from 64Kbytes to 512Kbytes, and the Y-axis denotes the average instruction fetch time in micro seconds. In the figures, PA-FIFO with SC reduces the average instruction fetch time by 13.2 – 23.6% in crafty, by 9.7 – 34.1% in eon, and by 1.9 – 2.0% in bzip2 trace. The improvement was more significant in crafty and eon trace, where the instruction cache hit ratios were relatively low, which were 0.944171 and 0.95876, respectively, and thereby the number of false faults was high. Bzip2 trace had the extremely high instruction cache hit ratio, which reached 0.999119, and thus most instructions were fetched from the instruction cache and the gains of reducing false fault latency was not significant.

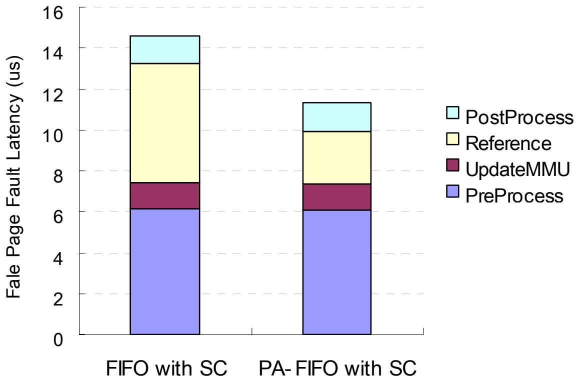


Figure 9: The false fault latency of FIFO-SC and PA-FIFO-SC

Table 4: The page miss ratio of Clock and PA-Clock in the traces

Trace & Scheme / Page Cache Size (KB)	Crafty		Eon		Bzip2	
	Clock	PA-Clock	Clock	PA-Clock	Clock	PA-Clock
64	0.001608	0.001610	0.001071	0.001080	0.000079	0.000078
128	0.000746	0.000750	0.000321	0.000324	0.000045	0.000046
256	0.000307	0.000311	0.000180	0.000184	0.000029	0.000029
512	0.000138	0.000138	0.000088	0.000088	0.000029	0.000029

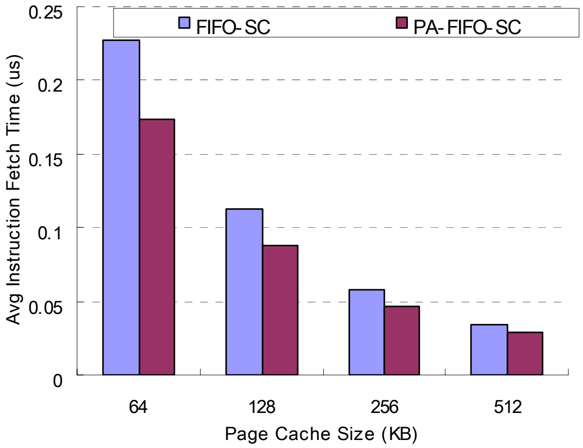


Figure 10: Average instruction fetch time of FIFO-SC and PA-FIFO-SC in the crafty trace

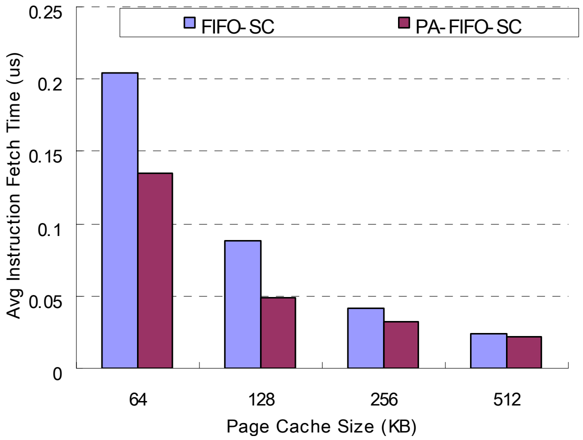


Figure 11: Average instruction fetch time of FIFO-SC and PA-FIFO-SC in the eon trace

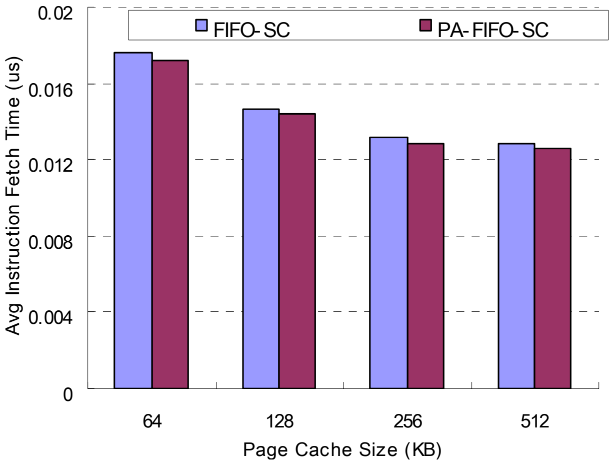


Figure 12: Average instruction fetch time of FIFO-SC and PA-FIFO-SC in the bzip2 trace

5. Conclusion

This paper addressed the question of how to reduce the page fault latency and presented a SWL demand paging scheme. The SWL demand paging scheme performs a free frame search and page loading to the stopover buffer, in parallel.

The results of the experiment revealed the following. First, the parallelized page fault handler effectively reduced the page fault latency. It was reduced by about 11.5% in the Clock scheme when the page list walk count was 2 and by 8.23% in FIFO with the SC scheme.

Second, the existing page cache replacement policies could not fully exploit the benefits of the parallelized page fault handler. For example, the Clock scheme still caused considerably long page fault latency even with the parallelized page fault handler. Also, the FIFO with SC scheme could not decrease the false fault latency that is the serious overhead of this method. The existing replacement policies need to be modified such that they can exploit the benefits of the parallelized page fault handler.

Third, PA-Clock, which we presented to overcome the drawbacks of the Clock scheme, improved the worst case latency of page fault significantly without hurting the average performance.

Fourth, PA-FIFO-SC, which we presented to overcome the drawbacks of the FIFO with SC scheme, reduced the latency of false faults by 21.7% by exploiting the spare time created by the parallelized page fault handler. Experiment results demonstrated that the reduced false fault latency contributed to reducing the average instruction fetch time.

In conclusion, the SWL demand paging scheme will contribute to proliferation of the demand paging scheme, even in embedded devices, by mitigating the unpredictably long page fault latency, which is the critical concern of demand paging.

References

[1] C. Park, J.-U. Kang, S.-Y. Park and J.-S. Kim. “Energy-aware demand paging on NAND flash-based embedded storages,” Proc. of the 2004 international symposium on Low power electronics and design (ISLPED 2004), 2004.

[2] C. Park, J. Lim, K. Kwon, J. Lee, and S. Min. “Compiler Assisted Demand Paging for Embedded Systems with Flash Memory, ” Proc of The 4th ACM International Conference on Embedded Software (EMSoft 2004), 2004.

[3] F. J. Corbato. “A Paging Experiment with the Multics System,” Project MAC Memo MAC-M-384, Mass. Inst. of Tech., 1968.

[4] Ozalp Babaoglu and William Joy. “Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits,” Proc. of the 8th ACM Symposium on Operating Systems Principles, 1981.

[5] Richard P. Draves. “Page Replacement and Reference Bit Emulation in Mach,” Proc. of the Usenix Mach Symposium, 1991.

[6] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. “EELRU: Simple and Effective Adaptive Page Replacement,” Proc. Of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 1999.

[7] Samsung NAND datasheets, http://www.samsung.com/Products/Semiconductor/NANDFlash, 2007

[8] Samsung OneNAND datasheets, http://www.samsung.com/Products/Semiconductor/OneNAND, 2007

[9] Nucleus Operating System, http://www.mentor.com/products/embedded_software/nucleus_rtos/index.cfm

[10] OMAP 5912 OSK, http://focus.ti.com/docs/toolsw/folders/print/tmdsosk5912.html

[11] ARM926EJ-S Technical Reference Manual, www.arm.com/documentation/ARMProcessor_Cores/index.html

[12] SPEC 2000 traces, http://traces.byu.edu/new/Documentation/