Cost-Efficient Memory Architecture Design of NAND Flash Memory Embedded Systems

Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim and Bumsoo Kim
Software Center, SAMSUNG Electronics, Co., Ltd.
{ci.park, pelican,dy76.seo, shinhank, bumsoo}@samsung.com

Abstract

NAND flash memory has become an indispensable component in embedded systems because of its versatile features such as non-volatility, solid-state reliability, low cos,t and high density. Even though NAND flash memory gains popularity as data storage, it also can be exploited as code memory for XIP (execute-in-place). In this paper, we present a cost-efficient memory architecture which incorporates NAND flash memory into an existing memory hierarchy for code execution. The usefulness of the proposed approach is demonstrated with real embedded workloads on a real hardware prototyping board.

1. Introduction

A memory architecture design is a main concern to embedded system engineers since it dominates the cost, power, and performance of embedded systems. The typical memory architecture of embedded systems consists of ROM for initial bootstrapping and code execution, RAM for working memory, and flash memory for permanent data storage. In particular, emerging memory technology, the flash memory, is becoming an indispensable component in embedded systems due to its versatile features: non-volatility, solid-state reliability, low power consumption, and so on. The most popular flash types are NOR and NAND. NOR flash is particularly well suited for code storage and execute-in-place (XIP) [1] applications, which require high-speed random access. While NAND flash provides high density and low-cost data storage, it does not lend itself to XIP applications due to the sequential access architecture and long access latency.

Table 1 shows different characteristics of various memory devices. Mobile SDRAM has strong points in performance but requires high power consumption over the other memories. Fast SRAM or low power SRAM can be selected according to the trade-off between power consumption and performance with a high cost. In non-volatile memories, NOR flash provides fast random access speed and low power consumption, but has high cost compared with NAND flash. Even though NAND flash shows long random read latency, it has advantages in low power consumption, storage capacity, and fast erase/write performance in contrast to NOR flash.

Table 1. Characteristics of various memory devices. The values in the table were calculated based on SAMSUNG 2003 memory data sheets [1-2].

		Current (mA)		Random Access (16bit)		
Memory	$/Gb	idle	active	read	write	erase
Mobile SDRAM	48	0.5	75	90ns	90ns	N.A
Low power SRAM	320	0.005	3	55ns	55ns	N.A
Fast SRAM	614	5	65	10ns	10ns	N.A
NOR	96	0.03	32	200ns	210.5us	1.2 sec
NAND	21	0.01	10	10.1us	200.5us	2 ms

Even though NAND flash memory is widely used as data storage in embedded systems, research on NAND flash memory as code storage are hardly found in industry or academia.

In this paper, we present a new memory architecture to enable NAND flash memory to provide XIP functionality. With XIP functionality in NAND flash, the cost of the memory system can be reduced since the NAND flash can be used as not only as data storage but also as code storage for execution. As a result, we can obtain cost-efficient memory systems with reasonable performance and power consumption.

The basic idea of our approach is to exploit the locality of code access pattern and devise a cache controller for repeatedly accessed codes. The prefetching cache is used to hide memory latency resulting from NAND memory access. In this paper we concentrate on code execution even though data memory is also an important aspect of memory architecture. There are two major contributions in this paper. First, we demonstrate the NAND XIP is feasible in real-life systems through a real hardware and

[1] XIP is the execution of an application directly from the Flash instead of having to download the code into the systems' RAM before executing it.

Proceedings of the 21st International Conference on Computer Design (ICCD'03)
1063-6404/03 $ 17.00 © 2003 IEEE

commercial OS environment. Second, we apply highly optimized caching techniques geared toward the specific features of NAND Flash.

The rest of this paper is organized as follows. In the next section, we describe the trend of memory architecture for embedded systems. Section 3 reviews related work in academia and industry. In Sections 4 and 5, we present our new memory architecture based on NAND XIP. In Section 6, we demonstrate the proposed architecture with real workloads on a hardware prototyping board and evaluate cost, performance, and power consumption over existing memory architectures. Finally, our conclusions and future work are drawn in Section 7.

2. Motivational Systems: Mobile Embedded Systems

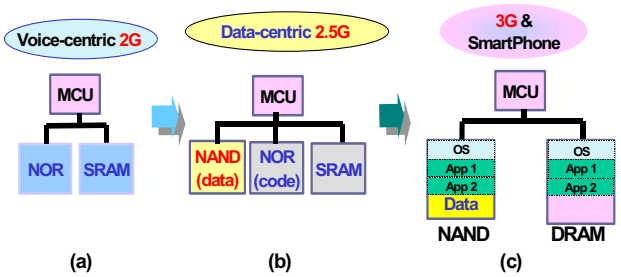


Figure 1. Mobile System Trend

Figure 1 shows mobile system trend in terms of memory hierarchy. The first approach is to use NOR and SRAM for code storage and working memory, respectively, as shown in Figure 1(a). It is appropriate for low-end phones, which require medium performance and cost. However, as mobile systems evolve into data centric and multimedia-oriented applications, high performance and huge capacity for permanent storage have become necessary. The second architecture (Figure 1(b)) seems to meet the requirements in terms of storage capacity through NAND flash memory, but its performance is not enough to accommodate 3G applications which consist of real-time multimedia applications. In addition, the increased number of components increases system cost. The third architecture (Figure 1(c)) eliminates NOR flash memory and uses NAND flash memory for using shadowing [2] technique. Copying all code into RAM offers the best performance possible, but contributes to the slow boot process. A large amount of SDRAM is necessary to hold both the OS and the applications. The higher power consumption from power hungry SDRAM memory is another problem for battery-operated systems.

[2] During system booting time, entire code image is copied from permanent storage into systems' RAM for execution.

As an improved solution of the third architecture in Figure 1(c), demand paging can be used with the assistance of operating system and it may reduce the size of SDRAM. However, this approach is not applicable to low or mid-end mobile system since it requires heavy virtual memory management code and MMU.

Thus, it is important to investigate an efficient memory system in terms of cost, performance and power consumption.

3. Related Work

In the past, researchers have exploited NOR Flash memory as caches for magnetic disks due to its low power consumption and high-speed characteristics. eNvy focused on developing a persistent storage architecture without magnetic disks [7]. Fred et al showed that flash memory can reduce energy consumption by an order of magnitude, compared to magnetic disk, while providing good read performance and acceptable write performance [9]. B. Marsh et al examined the impact of using flash memory as a second-level file system buffer cache to reduce power consumption and file access latency on a mobile computer [8].

Li-Pin et al investigated the performance issue of NAND flash memory storage subsystems with a striping architecture, which uses I/O parallelism [10]. In industry [5], NAND XIP is implemented using small size of buffer and I/O interface conversion, but the XIP area is limited to boot code, thus OS and application codes should be copied to system memory.

In summary, even though several researches have been made to obtain the maximum performance and low power consumption from data storage, few efforts to support XIP in NAND flash are found in academia or industry.

4. NAND XIP Architecture

In this section, we describe NAND XIP architecture. First, we look into the structure of NAND flash and illustrate basic implementation of NAND XIP based on caching mechanism.

4.1. Background

A NAND flash memory consists of a fixed number of blocks, where each block has 32 pages and each page consists of 512 bytes main data and 16 bytes spare data as shown in Figure 2. Spare data can be used to store auxiliary information such as bad block identification and error correction code (ECC) for associated main data. NAND flash memories are subject to a condition called "bad block", in which a block cannot be completely erased or cannot be written due to partial or 2-bit errors. Bad blocks

may exist in NAND flash memory when shipped or may occur during operation.

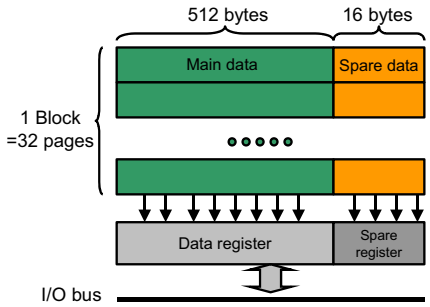


Figure 2. Structure of NAND flash memory

In order to implement the NAND XIP, we should consider the following points.

- Average memory access time
- Worst case handling
- Bad block management

The performance of memory system is measured by average access time [3]. In order to implement XIP functionality, the average access time of NAND flash should be comparable to that of other memories such as NOR, SRAM and SDRAM. Though average memory access time is a good metric for performance evaluation, worst-case handling, or cache miss handling is another problem in practical view since most mobile systems such as cellular phones include time-critical interrupt handling such as call processing. For instance, if the time-critical interrupt occurs during cache miss handling, the system may not satisfy given deadline and to make it worse, it may lose data or connection. The third aspect to be considered in NAND XIP is to manage bad blocks, which are inherent in NAND flash memory because bad blocks cause discontinuous memory space, which is intolerable for code execution.

4.2. Basic Implementation

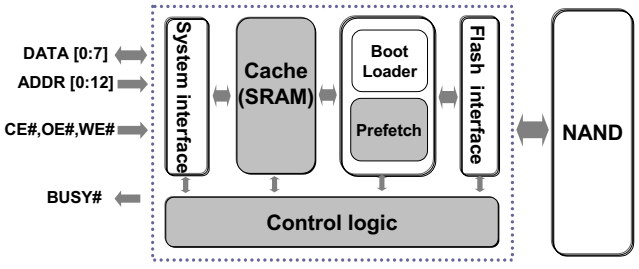


Figure 3. NAND XIP controller

The proposed architecture consists of a small amount of SRAM for cache, interface conversion logic, the control logic and NAND Flash as shown in Figure 3. Interface conversion is necessary to connect the I/O interface of NAND flash to memory bus. For cache mechanism, direct map cache with victim cache is adopted based on Jouppi's work in [4] with optimization for NAND flash. In [4], the victim cache is accessed on a main cache miss; if the address hits the victim cache, the data returned to the CPU and at the same time it is promoted to the main cache; the replaced block in the main cache is moved to the victim cache, therefore performing a "swap". If the victim cache also misses, NAND flash access is performed; the incoming data fills the main cache, and the replaced block will be moved to the victim cache. In next section, we modify the above "swap" algorithm using system memory and *page address translation table (PAT)*. The prefetching cache is used to hide memory latency resulting from NAND memory access. Several hardware prefetching techniques can be found in literature [12]. In our case, prefetching information is analyzed through profiling process and the prefetching information is stored in spare data at code image building time.

5. Intelligent Caching: Priority-based Caching

Though the basic implementation is suitable for application code which shows its spatial and temporal localities, it may be less effective in systems code which has a complex functionality, a large size, and interrupt-driven control transfers among its procedures [13]. Torrellas et al. presented that the operating system has the characteristics that large sections of its code are rarely accessed and suffers considerable interference within popular execution paths [13]. For example, periodic timer interrupt, rarely-executed special-case code, and plenty of loop-less code disrupt the localities. On the other hand, real-time applications should be retained as long as possible to satisfy the timing constraints[3]. In this paper, we distinguish the different cache behavior between system and application codes, and adapt it to the page-based NAND architecture. We apply profile-guided static analysis of code access pattern.

We can divide code pages into three categories depending on their access cost: high priority, mid priority and low priority pages. Even though the priority can be determined by various objectives, we set the priority to pages based on the number of references to pages and their criticality. For example, if a specific page is referenced more frequently or has time critical codes, it is classified as a high-priority page and should be cached or retained in cache to reduce the later access cost in case that the page is in NAND flash memory. OS-related code, system libraries and real-time applications have high-priority pages. On the

[3] In this paper, real-time applications indicate multimedia applications with soft real-time constraints.

other hand, mid-priority page is defined to be normal application code which is handled by normal caching policy. Finally, low-priority page corresponds to sequential code such as initialization code, which is rarely executed. PAT is introduced to remap pages in bad blocks to pages in good blocks and to remap requested pages to swapped pages in system memory. We illustrate the caching mechanism in detail in Figure 4. First, when page *A* with high-priority is requested, it is cached from NAND flash to main cache. Next, when page B is requested from the CPU, it should be moved to main cache or system memory. Here assuming that page B is in conflict with page A, page B is moved to system memory (SRAM/SDRAM) since page B is low priority page ("*L"* in spare area of NAND flash memory means *low-priority*). At the same time, PAT is updated so that later access to page B is referred to system memory. Again, when page C is requested and in conflict with page A, page C replaces page A and page A is discarded from main cache since C's priority is high. The evicted page A is moved to victim cache. In summary, on NAND flash's page demand, the controller discards or swaps existing cache page according to the priority information stored in spare area data. The detail algorithm is explained in Figure 5. Another usage of spare area data is to store prefetching information based on profiling information gathered during code execution. This static prefetching technique improves memory latency hiding without miss penalty from miss-prediction at run-time.

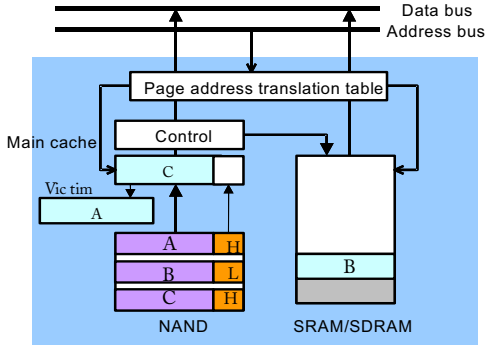


Figure 4. Intelligent Caching Architecture

```
Priority Caching (address)
{
    page = convert(address);
    if (isInPAT(page))
        main memory hit;
    else if(isInMainCache(page))
        Cache hit;
    else if(isInVictimCache(page))
        Victim hit;
    else { // miss, fetch a page from NAND flash memory

        page_priority = NAND[page].priority;
        if (cache[page%CACHE_SIZE].priority ==HIGH)
          NAND[page] → main memory;
        else if (cache[page%CACHE_SIZE].priority == MID)
        {
           cache[page%CACHE_SIZE] → victim;
           NAND[page] → cache[page%CACHE_SIZE];
        }
        else
           NAND[page] → cache[page%CACHE_SIZE];
    }
}
```

Figure 5. Intelligent Caching Architecture

6. Experimental Setup

This section presents our experiment environment. Our environment consists of a prototyping board, our in-house cache simulator with pattern analysis and a real workload, namely PocketPC [6] as shown in Figure 6. The prototyping board is composed of: main board and daughter board (a yellow rectangle in Figure 6). The main board has ARM9-based micro-controller, SDRAM, NOR flash and so on. The daughter board contains an FPGA for cache controller and victim cache, fast SRAM for tag and cache memory, and two NAND flash memories. The daughter board is used not only to implement a real cache configuration on FPGA but also to gather memory address traces from running applications. In Figure 7, one NAND is dedicated to NAND XIP and the other NAND is dedicated for collecting memory traces from host bus. Trace collection function is started and stopped by using manual switches and manual switch's on/off interval determines the time period for trace gathering. Collected address traces are stored for cache simulator.

The specification of main processor and NAND flash is shown in Table 2. The cache simulator explores various parameters such as miss rate, replacement policy, assoicativity, and cache size based on memory traces from the prototyping board. The real embedded workload, PocketPC supports XIP-enabled image based on the existing ROMFS file systems in which each application can be directly executed without being loaded into RAM.

Table 1: The specification of the prototyping board

Parameter	Configuration
CPU clock	200 MHz
L1 Icache	64way, 32byte line, 8KB
Bus width	16bit
NAND read initial latency	10us
NAND serial read time	50ns
SRAM read time	10ns
SDRAM read time	90ns
NOR read time	200ns

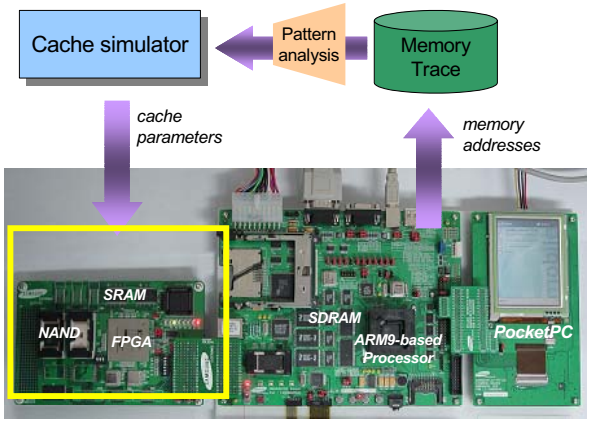


Figure 6. A prototyping board for NAND XIP

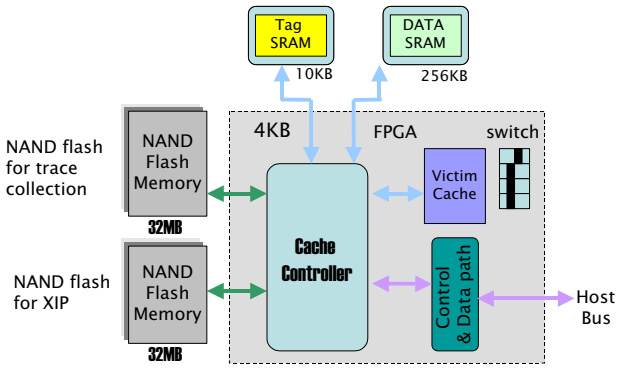


Figure 7. FPGA Prototyping for NAND XIP

6.1. Experimental Results

In Figure 8, we compare the miss ratio over various configuration parameters such as associativity, replacement policy, and cache size. We collected address traces from PocketPC while we were executing various applications such as "Media Player" and "MP3 player" since they are popular embedded multimedia applications which involve real-time requirements. Note that the cache size is the most important factor to affect miss ratio as shown in Figure 8.

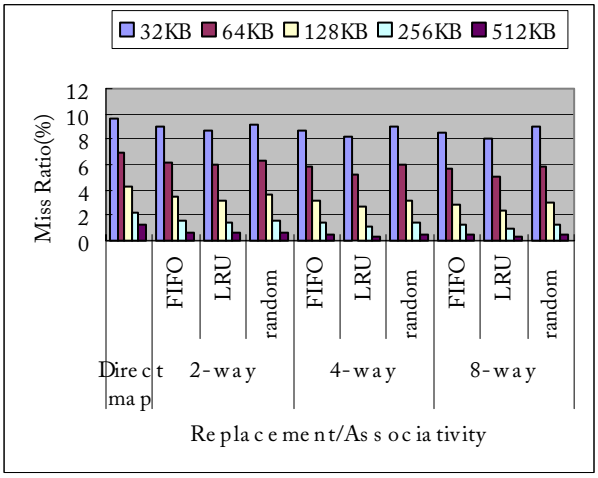
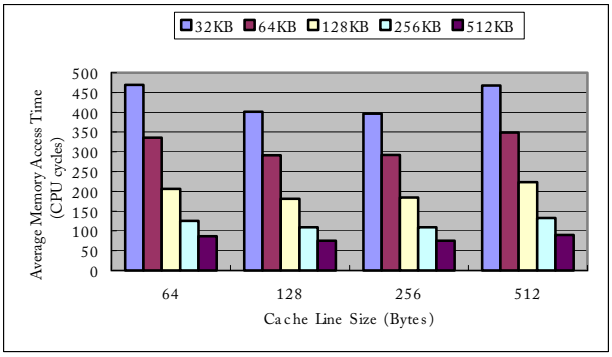
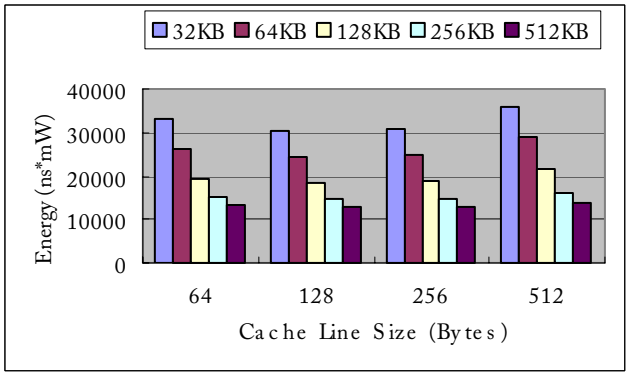


Figure 8. Associativity with different replacement policies and cache sizes versus miss ratio



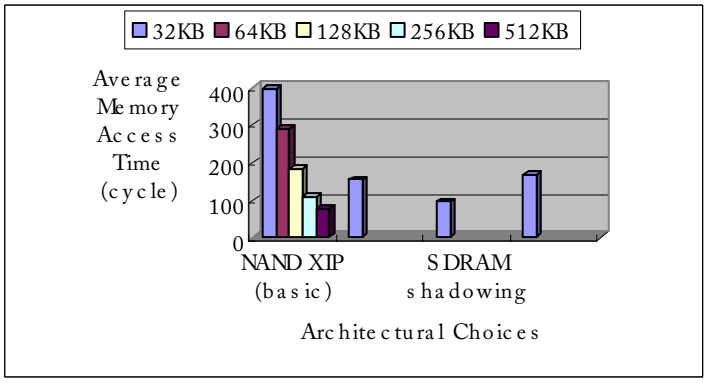
(a)



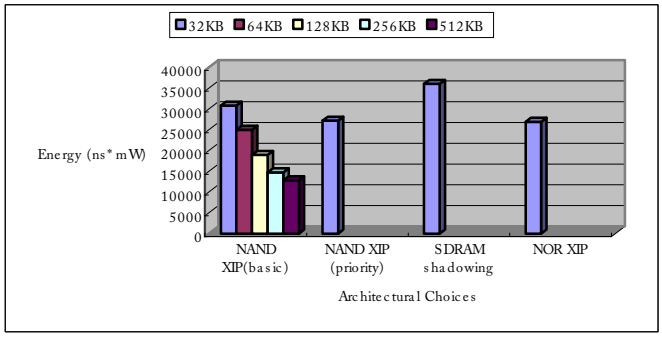
(b)

Figure 9. Cache line size versus (a) access time per 32-byte and (b) energy consumption

To analyze the optimal cache line size in NAND XIP cache, simulation has been done with the memory traces which are gathered from the prototyping board. The line size of 256-byte shows better numbers in average memory access time and in energy consumption over all other cache sizes as shown in Figure 9. Therefore, the line size of NAND XIP controller is determined to be 256-byte hereafter.



(a)



(b)

Figure 10. Overall performance comparison of different memory architectures: (a) average memory access time and (b) energy consumption.

Architectural Choices	Components	Booting Time (sec)	Cost ($)
NOR XIP	NOR(64MB)+SDRAM(32MB)	13	60
SDRAM Shadowing	NAND(64MB)+SDRAM(64MB)	15	35
NAND XIP (basic)	NAND XIP(64MB)+SRAM(256KB) Controller+SDRAM(32MB)	14	26
NAND XIP (priority)	NAND XIP(64MB)+SRAM(64KB) Controller+SDRAM(32MB)	14	24

Figure 11. Overall booting time and cost comparison of different memory architectures.

Figures 10 and 11 show overall performance comparison among different memory architectures based on our prototyping results. NOR XIP architecture (NOR+SDRAM) shows fast boot time and low power consumption at high cost. Even though SDRAM shadowing architecture (NAND + SDRAM) achieves high performance, it suffers from a relatively long booting time and inefficient memory utilization. Finally, our approach NAND XIP shows reasonable booting time, performance ,and power consumption with outstanding cost efficiency. Two implementations of NAND XIP (basic and priority) are also compared. The NAND XIP (basic) is composed of 256KB cache and 4KB victim cache. On the other hand, the NAND XIP (priority) has 64KB cache, 4KB victim and 2KB PAT. The NAND XIP (priority) has advantage of cache size reduction with the assistance of system memory and priority based caching.

6.2. Worst Case Handling

Even though NAND XIP offers an efficient memory system, it may suffer from worst-case handling, namely cache miss handling. A straightforward solution is to hold the CPU until the requested memory page arrives. This can be implemented using wait/ready signals' handshaking method. However, the miss penalty, 35us[4] is not trivial time especially in case that a fast processor is used. Besides CPU utilization problem, if the time-critical interrupt request occurs to the system, the interrupt may be lost because the processor is waiting for memory's response. In order to solve this problem, a system-wide approach is needed. First, OS should handle the cache miss as a page fault as in virtual memory management. The CPU also should supply "abort" function to restart the requested instruction after cache miss handling.

7. Conclusions

We enlarged the application of NAND flash to code execution area and demonstrated the feasibility of the proposed architecture in real-life mobile embedded environment. As future work, system-wide approach will be helpful to exploit NAND flash in embedded memory systems. A new task scheduling algorithm, considering the read, erase and program operations of NAND flash, will utilize system resources more efficiently. Code packing and replacement with compiler's assistance will be achieved in near future. In addition, data caching mechanism can be a challenging topic as flash write time is getting shortened.

[4] It is calculated as sum of NAND initial delay (10us) + page read (512 x 50ns).

References

[1] Samsung Electronics Co., "NAND Flash Memory & SmartMedia Data Book", 2002.

[2] http://www.samsung.com/Products/Semiconductor/index.htm

[3] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", 2nd edition, Morgan Kaufman Publishers, 1996.

[4] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA-17:ACM/IEEE International Symposium on Computer Architecture, pp. 364-373, May 1990.

[5] http://www.m-sys.com/files/documentation/doc/Mobile_DOC_G3_DS_Rev1.0.pdf

[6] Microsoft Co., "Pocket PC 2002", http://www.microsoft.com/mobile/pocketpc

[7] M. Wu, Willy Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System", Proc. Sixth Int'l Conf. On Architectural Support for Programming Languages and Operating Sys. (ASPLOS VI), San Jose, CA, Oct. 1994, pp. 86-97.

[8] B. Marsh, F. Douglis, and P. Krishnan, "Flash memory file caching for mobile computers," Proc. 27th Hawaii Conf. On Sys. Sciences, Wailea, HI, Jan. 1993, pp. 451-460.

[9] F. Douglis et al., "Storage Alternatives for Mobile Computers", Proc. First USENIX Symp. On Operating Sys. Design and Implementation, Monterey, CA, pp 25-37, Nov. 1994.

[10] Li-Pin Chang, Tei-Wei Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002) September, 2002. San Jose, California.

[11] J. R. Lorch and A. J. Smith, "Software Strategies for Portable Computer Energy Management", IEEE Personal Communications, June, 1998.

[12] C. Young, E. Shekita, "An Intelligent I-Cache Prefetch Mechanism," in Proceedings of the International Conferences on Computer Design, pp. 44-49, Oct 1993.

[13] Josep Torrellas, Chun Xia, and Russell Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workload", Proc. HPCA-1: 1st Intl. Symposium on High-Performance Computer Architecture, p.360, January 1995.