INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
The architecture of eNVy, a non-volatile, main memory storage system

Wu, Michael, M.S.
Rice University, 1994
RICE UNIVERSITY

The Architecture of eNVy, A Non-Volatile, Main Memory Storage System

by

Michael Wu

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Master of Science

APPROVED, Thesis Committee:

Willy Zwaenepoel
Dr. Willy Zwaenepoel, Chairman
Professor
Computer Science

John K. Bennett
Dr. John K. Bennett
Assistant Professor
Electrical and Computer Engineering

Peter J. Várman
Dr. Peter J. Várman
Associate Professor
Electrical and Computer Engineering

Houston, Texas
April, 1994
Abstract

The Architecture of eNVy, A Non-Volatile, Main Memory Storage System

Michael Wu

This paper describes the architecture of eNVy, a large non-volatile main memory storage system built primarily with Flash memory. Flash provides persistent storage with solid-state memory access times at a lower cost than other solid-state technologies. eNVy presents its storage space as a linear, memory mapped array rather than as a disk emulator in order to provide an efficient and easy to use software interface.

Flash chips are write-once, bulk-erase devices whose contents cannot be updated in-place. They suffer from slow write times and limited program/erase cycles. eNVy uses a copy-on-write scheme, some battery-backed SRAM, and parallel operation to overcome these problems and provide low latency in-place update semantics. A specialized cleaning algorithm maximizes the lifetime of the Flash array. Simulations show that eNVy can handle I/O rates corresponding to approximately 30,000 TPS on the TPC-A benchmark with average latencies as low as 180ns for reads and 200ns for writes.
Contents

List of Illustrations iv
Abstract v

1 Overview 1

2 The eNVy Architecture 3
2.1 Introduction 3
2.2 Flash Characteristics 3
2.3 A Persistent Store Interface 5
2.4 Implementation of a Transparent In-Place Update 6
2.5 Battery Backed SRAM as a Write Buffer 8
2.6 Reading and Programming Flash 9
2.7 Hardware Overview 10

3 Cleaning Policy 12
3.1 Introduction 12
3.2 The Basic Cleaning Function 13
3.3 Greedy Method 16
3.4 Locality Gathering 18
  3.4.1 Algorithm Description 18
  3.4.2 Example 23
3.5 Comparison to Optimal Hand Calculations 28
3.6 Hybrid Approach 30
3.7 Estimated eNVy Lifetime 32

4 Simulation Results 33
4.1 Introduction 33
4.2 Simulation Overview 33
4.3 Throughput and Latency 35
4.4 Performance of the Cleaning Algorithm ................. 38

5 Future Work 41
  5.1 Hardware Extensions ........................................ 41
  5.2 Software Issues ........................................... 42

6 Related Work 45

7 Conclusion 48

Bibliography 50
Illustrations

2.1 Steps in the Copy on Write Function ........................................ 8
2.2 Diagram of eNVy in a Host Workstation ................................. 10
2.3 Block Diagram of eNVy Architecture ................................. 11

3.1 Steps in the Cleaning Process ........................................ 13
3.2 Cleaning Costs for Varying Flash Utilization ......................... 15
3.3 Distribution of Space for Various Cleaning Methods .................. 16
3.4 Comparison of Cleaning Algorithms .................................. 17
3.5 Segment Utilization over Time ......................................... 22
3.6 Utilization and Cleaning Indices for Uniform Accesses .............. 24
3.7 Utilization and Cleaning Indices at Time 1 .......................... 26
3.8 Utilization and Cleaning Indices at Time 10 .......................... 27
3.9 Utilization and Cleaning Indices at Time 1000 ........................ 29
3.10 Cleaning Costs for a 10/90 Reference Pattern ......................... 31

4.1 eNVy Simulation Parameters ........................................ 35
4.2 Measured Throughput for Increasing Request Rates .................. 36
4.3 I/O Latency for Increasing Request Rates ........................... 36
4.4 Measured Throughput for Various Levels of Flash Utilization ....... 37
4.5 Effect of Buffer Size on Write Latency .............................. 39
4.6 Cleaning Costs vs. Partition Size .................................... 39
4.7 Cleaning Costs vs. Partition Size .................................... 40
Chapter 1

Overview

Traditional I/O systems use primarily disks for stable storage. Access to these devices require physical motion, resulting in a large and increasing performance discrepancy between stable storage and the solid-state processors and memories that make up the rest of a computer system. Caches are commonly used to hide the read latency of disks [15] and non-volatile write caches have been developed to hide the write latency [1, 3, 12]. RAID arrays improve available I/O rates by transferring data in parallel [10, 5, 11]. These techniques are described in more detail in chapter 6. In certain situations, each can provide substantial benefits, but only to a certain extent.

We are exploring an alternative approach, building a persistent storage system using solid-state technology, thereby avoiding the mechanical latencies associated with disks. There are three solid-state memory technologies that can be considered for non-volatile storage: battery-backed SRAM, disk-backed DRAM, and Flash memory. Of the three, Flash memory seems to provide the best choice for a mass storage system. Flash is much cheaper than battery-backed SRAM ($25 per MBytes, $100 per MByte), and it is inherently non-volatile. Unfortunately, Flash memories have some drawbacks that need to be considered when attempting to modify data in an array.
These drawbacks include the inability to update data in-place, slow program times, and limited cycling.

The goal of eNVy to overcome Flash memory's storage limitations while providing an efficient persistent store interface. eNVy uses a copy-on-write function, a small amount of battery-backed SRAM, and memory remapping to provide in-place update semantics while hiding write latency with a small SRAM write buffer. The basic hardware is supported by software that manages movement of data within the Flash array to minimize the amount of data written in a process known as cleaning. The cleaning function maximizes the lifetime of the Flash array to a point where Flash chips, even with their limited cycling restriction, can provide usable lifetimes on the order of 10 years.

The outline of the rest of this paper is as follows. Chapter 2 describes the eNVy architecture in detail while Chapter 3 elaborates on the cleaning algorithm used to manage the Flash array. Chapter 4 presents the results of our simulation of the basic architecture. Subsequent chapters discuss extensions and conclusions.
Chapter 2

The eNVy Architecture

2.1 Introduction

The goal of eNVy hardware is to overcome the limitations of Flash memories for use as fast mass storage devices. The basic architecture is responsible for handling two of Flash’s major drawbacks, specifically, the inability to update data in-place and the relatively long latency of programming operations. This section discusses Flash chips in more detail to explain where the Flash drawbacks come from and the persistent store interface eNVy tries to supply, then describes the architecture itself.

2.2 Flash Characteristics

A Flash chip is very similar to an EEPROM. Reads occur at speeds comparable to DRAM (100ns). Writes involve programming non-volatile cells and take under 10 microseconds, much faster than a disk, but relatively slow when compared to main memory. Once a byte has been programmed, it must be erased before it can be rewritten. Flash is different from other memories in that data can only be erased in large blocks. To provide some flexibility in erasure, each chip is divided into some small number of blocks, typically about 16. For a 1 Mbyte chip, this yields block sizes
of 64K. Each block can be independently erased in a few tens of milliseconds. The inability to erase data on the byte or word level prevents standard Flash memories from being accessed with normal update-in-place memory semantics.

Furthermore, the number of write/erase cycles is limited. Chips currently in volume production are rated for 10,000 to 1 million cycles. It is important to note that manufacturers calculate the lifetime of their chips by defining a failure as the inability to fully program a byte in a given amount of time (about 10µs). A failure does not signify the loss of any data, as in a disk drive. In the case of failure, the controller can simply reattempt to program the byte until it succeeds. A “real” failure occurs when the amount of time it takes to write a byte exceeds the maximum time defined by the system. The byte can be then marked bad, but again, without loss of data.

The program and erase times increase slowly as the device is used, but experiments we have done suggest that the decay in modern devices is minimal and the lifetime specifications given by the manufacturers are very conservative. In one experiment, we ran a chip rated for 10,000 cycles through 2 million cycles. After the test, the chip still exhibited program times of 4µs and erase times of 40ms, only about twice as long as it took when it was new and well within its specifications.
2.3 A Persistent Store Interface

eNVy attempts to provide a conventional in-place update memory interface to its storage system unlike current commercial solutions that package solid-state memory arrays as non-volatile disk emulators. While such a disk emulator provides much faster stable storage access than a disk using a backward-compatible interface, the disk emulation overhead limits maximum performance. Since memory access times are relatively close to processor speeds, it becomes harder to justify the infinitely fast processor assumption typically used in I/O performance modeling that ignores software costs. A storage access can be dominated by the time it takes to execute file system, disk emulation, and cache code rather than by the actual time it takes to access a memory array. A memory interface eliminates these problems as well as others, such as the need to move large blocks of data to satisfy requests for just a few bytes and the use of high cost interrupts to signal block transfers.

This type of interface allows the use of standard in-memory data structures for all storage requirements without any need for disk data placement or linear, file-oriented "save" formats, allowing for faster and more efficient access. Programming concerns such as these are apparent for emerging applications such as object oriented databases that can provide gains in performance and functionality by removing constraints imposed by fixed record sizes. The simplicity of the interface also allows a great deal of flexibility. For example, if desired, a RAM disk program can easily be used to provide compatibility with existing programs.
Researchers working on the Starburst project at IBM [7] have demonstrated the potential benefits of a memory storage interface. They were able to replace the lower levels of a database with routines that operate on data directly in memory. When compared to the disk version of the same database with all data cached in memory but using the normal file system interface, the direct access method reduced the time spent in the storage component of database transactions by over 600%. This translated into typical overall performance gains of around 200% to 300%.

2.4 Implementation of a Transparent In-Place Update

The presentation of the Flash array as an in-place update, non-volatile memory is achieved through a copy-on-write mechanism using a relatively small amount of battery backed SRAM. The Flash array is internally divided into pages, similar to a virtual memory system but completely managed by the eNVy hardware. When a write to Flash address space is requested, the page corresponding to the write is copied to SRAM where the actual write is performed. Figure 2.1 illustrates this copy-on-write process. At some later point, e.g., when the Flash array is idle, pages from SRAM are flushed back to the Flash array. The data path between the SRAM and Flash is very wide, allowing an entire page to be transferred in just one memory cycle. The page table entries can be updated in parallel with the transfer from Flash to SRAM to help keep minimize copy-on-write costs. Additional selection hardware
lets the host processor access 32 or 64 bits of the array at a time according to its native bus width.

A level of indirection, provided by a page table, allows the eNVy controller to move pages between the Flash and SRAM arrays transparently to the host processor. The address provided by the host is treated as a logical address and is translated into a physical address using the page table. The MMU acts as a cache of recently used mappings to make this translation faster.

The mapping of logical to physical addresses is critical to the integrity of the system, therefore it must be kept in non-volatile memory. Since mappings are updated frequently and changes must occur in-place, the page table is kept in battery-backed SRAM rather than in Flash memory. This memory is expensive so a tradeoff must be made when choosing the page size. Larger pages lead to a smaller page table and lower SRAM requirements, but since an entire page has to be written to Flash with every flush, more unmodified data is written for every word changed. In the interests of maximizing the Flash array's lifetime, we want the page size to be as small as possible without incurring too much storage overhead. A page size of 256 bytes was chosen. Fortunately, a page table mapping only requires 6 bytes, relatively little data compared to the 256 bytes of Flash memory it references, so overall system cost is not affected significantly.
Figure 2.1 Steps in the Copy on Write Function

### 2.5 Battery Backed SRAM as a Write Buffer

While a page is in SRAM, all requests are redirected to the SRAM copy. Keeping the page in SRAM rather than immediately flushing it makes the system more efficient since multiple writes to the same page do not require additional copies from Flash. Many pages can be stored in SRAM to form a buffer of recent requests. The primary purpose of this buffer is not to improve read access time since reads from the Flash array take the same amount of time. More important benefits include hiding the latency of writes, and reducing write traffic to the Flash array.
2.6 Reading and Programming Flash

When a read is issued by the host computer, a simple address translation is done by the MMU, and the desired data is fetched from the Flash or SRAM array. A write to Flash memory consists of a program initiation operation that starts writing data, then several verify operations that continue until the data is completely written. The wide bus allows an entire page to be transferred in one memory cycle. A comparator is added for each line between the Flash and SRAM, allowing an entire Flash page to be compared to a SRAM page in one cycle. The result is delivered to the controller as a bit vector.

The comparators avoid a serial comparison, which would render the wide bus all but useless. In a large memory array, data transceivers are needed to control the same bus lines that the comparators need to monitor in eNVy. By combining the data driving/isolation function with the comparator in an ASIC with a similar number of pins, we can add the comparison function without increasing the actual chip count to more than that required by a normal Flash array.

When the SRAM write buffer has been filled to some fraction of its capacity, the memory controller attempts to flush some pages from SRAM to Flash. If there is no free space where the controller wants to flush the page, a cleaning operation is initiated to reclaim space. The flushing and cleaning functions are labeled "long" operations because they take significantly more time than a memory access. If a host request arrives while a long operation is in progress, the long operation is suspended and the
host access serviced, minimizing the latency seen by the host. The memory controller waits a few microseconds before resuming the long operation to avoid spurious restarts during bursts of I/O activity.

2.7 Hardware Overview

Figure 2.2 shows how eNVy fits into a computer system. eNVy resides on the memory bus, and is accessed with exactly the same load/store primitives as main memory, although with variable rather than fixed latency. Memory accesses pass from the host’s memory bus to the eNVy controller which translates each access into Flash and SRAM operations to satisfy the host’s request.

A more detailed diagram is shown in Figure 2.3. The controller (shaded) is actually made up of several components: a hardware memory controller, a 32 bit RISC CPU, a memory management unit (MMU) and a page table. The memory controller is
Figure 2.3 Block Diagram of eNVy Architecture

responsible for initiating all internal operations. Certain operations, such as the cleaning algorithm described in Chapter 3 are too complex to implement completely in hardware, so a RISC CPU (hereafter called the cleaning processor) is included to assist the memory controller. The actual transfer of control from the memory controller to the cleaning processor is done through a shared register which the cleaning processor polls to avoid interrupt latencies. The combination of the hardware memory controller and the cleaning processor is powerful enough to handle all eNVy functions, and the host processor is never interrupted to do any work. This eliminates a potentially large source of latency since the tens of microseconds spent in an interrupt handler is very large compared to a 100ns memory access.
Chapter 3

Cleaning Policy

3.1 Introduction

The copy-on-write function invalidates one page of data each time it is invoked. Since Flash acts as a write-once device in between erasures, new data cannot be written in its place. Over time the invalidated data will occupy all of the free space in the Flash array and flushes from the write buffer will be blocked. The eNVy hardware must periodically reclaim this space in a process we call cleaning. The decision as to which blocks to clean, when to clean them, and where to write new data is described by our cleaning policy.

The goal of the cleaning policy is to move as little data as possible. A more efficient algorithm has several desirable effects such as higher performance (less bandwidth is consumed by the cleaner) and longer Flash array lifetime (less data actually has to be written). This chapter describes the basic cleaning operation and work we have done in creating an efficient cleaning policy.
3.2 The Basic Cleaning Function

The eNVy cleaning system performs the same basic function as the Sprite LFS segment cleaner [13]. To clean a segment eNVy finds a new, freshly erased segment and copies all active pages to it from the old one. The old segment will be left with no useful data and can therefore be erased and reused. Free space in the new segment will be grouped at its tail and can start accepting new data. The steps involved in the cleaning process are shown in Figure 3.1. eNVy must always keep one segment completely erased between cleaning operations so the next cleaning operation will have somewhere to copy its data to. The state of the cleaning function is kept in a piece of the battery backed SRAM in the eNVy controller (also used by the page
table) so no lengthy recovery has to be done if a system failure occurs. The controller simply restarts where it left off.

We define the Flash write cost to be the number of Flash program operations actually performed by the cleaning algorithm for every write operation to Flash the eNVy controller initiates. It provides a measure of the efficiency of the cleaning algorithm. The Flash write cost is different from Sprite LFS's write cost because it counts all disk I/Os including reads. In a Flash system, since program times are so much longer than reads, the time spent reading during a cleaning operations is small. We also don't consider the time it takes to initially flush a page since it is not actually part of the cleaner.

The number of program operations needed to clean a segment depends on its utilization, the percentage of storage space actually occupied by live data. If \( u \) is the utilization of the segment, the write cost to clean it is \( \frac{u}{1-u} \). Figure 3.2 graphically illustrates this cost for different levels of utilization. After about 80% utilization, the write cost quickly reaches unreasonable levels. For this reason, we limit the percentage of live data in the eNVy system to 80% of the total Flash array capacity.

One of the primary goals of any cleaning policy is to minimize its write cost. In an attempt to lower its write costs, LFS tries to take advantage of access patterns with some locality of reference. It generates a bimodal distribution of hot and cold data and uses a cost/benefit policy to choose which segments to clean. The bimodal distribution is generated by reading several segments at once, sorting the active data
Figure 3.2 Cleaning Costs for Varying Flash Utilization

by age, and re-writing the data to several unused segments. Since a block’s age is roughly a function of its frequency of access, some segments get filled with frequently accessed data and others get filled with colder data.

The main reason we did not implement the LFS cleaning policy is that it would consume too much of the available resources. Unlike disk based segments in LFS that can be arbitrarily small, segments in eNVy are restricted by the large size of physical erase blocks coupled with the need to use Flash chips in 256 byte wide banks rather than word sized banks. In a 2 GByte array such as the one we simulated in Chapter 4, this restriction results in a relatively small number (128) of 16 Mbyte segments. Cleaning just 10 segments at a time would tie up over 10% of the total data in the system. The need to keep an erased segment free for every segment involved in the clean uses even more space and further amplifies the problem. In addition, the short access times characteristic of the eNVy system prevents the use of some aspects
of the LFS algorithm such as sorting pages while cleaning. Since the LFS algorithm doesn’t suit eNVy’s needs, we chose to try to come up with a different policy that can still take advantage of locality, but requires less to be cleaned with each operation.

### 3.3 Greedy Method

The first policy we experimented with is very similar to the unsuccessful greedy policy in Sprite. Data flushed from the write buffer is written sequentially into a segment until it becomes filled, as if it were a log. The cleaner then chooses to clean the segment with the most invalidated space in the hopes of recovering as much space as possible. Further writes are directed to the freshly erased segment recovered by the cleaner.

As predicted by the Sprite LFS paper, performance actually gets worse as the locality of reference increases (see Figure 3.4). We found that the greedy algorithm closely approximates a FIFO ordering on the segments being cleaned. For a system

![Figure 3.3 Distribution of Space for Various Cleaning Methods](image-url)
Figure 3.4 Comparison of Cleaning Algorithms

with N segments and a uniform access pattern, once a segment is cleaned, it doesn't become the segment with the most invalidated data until N-1 clean operations later. To test this observation, we replaced the greedy algorithm with a loop that cleans the segments sequentially as if they were placed in a FIFO after they were cleaned.

The write costs are basically identical. This is interesting because FIFO ordering is very easy to implement. Our experiments indicate that this ordering provides optimal performance for a uniform access distribution. A FIFO causes a segment to wait as long as possible before being cleaned again. This gives the pages it contains the most chance to be invalidated, translating into a low write cost.
3.4 Locality Gathering

3.4.1 Algorithm Description

The greedy method works poorly with increased spatial locality of reference. Since flushed data is written to a segment arbitrarily designated for flushes rather than the segment it was read from, pages with locality become randomly dispersed in the array and lose their locality. An obvious solution is to keep track of which segment a page comes from when it is placed in SRAM after a copy-on-write, and flush it back into the same segment when flushed, cleaning the segment if necessary. The problem with this approach is that a segment will always be filled with 80% live data in it, yielding a fixed cleaning cost of 4.0.

The premise behind our locality gathering algorithm is that if spatial locality is present, it should be possible to redistribute free space in the array among the segments to minimize write costs. A segment which is infrequently used should give up free space to a frequently used segment to lower the cleaning cost of the common case. Preliminary calculations and experiments show that the array reaches its optimal state (lowest write cost) when the value of the ratio \( \frac{\text{Time since segment } u \text{ has been cleaned}}{\text{Cost to clean segment } u} \) is the same for all segments. We call this ratio the cleaning index. The cleaning algorithm attempts to modify the utilizations of all the segments to reach some final state where all cleaning indices are equal, meeting the condition given above.

When the Flash array is not in its stable state, the cleaning index can be used to determine which segments need more free space and which segments have too much
free space. The average of all the actual cleaning indices is used as a guess for the final cleaning index. Segments above the average have too much free space and need to give up free pages to raise their write cost (thereby decreasing their cleaning index) and segments below the average need to do the opposite.

Every time a segment is cleaned, its cleaning index is computed using Equation 3.2 and reflected in the average for the array by Equation 3.3.

\[
CleanIndex[n] = \frac{TurnoverTime[n]}{CleanCost[u[n]]} \quad (3.1)
\]

\[
= \frac{TurnoverTime[n]}{\frac{u[n]}{1-u[n]}} \quad (3.2)
\]

\[
AverageIndex = \frac{\sum_{n=0}^{numsegs-1} CleanCost[n]}{numsegs} \quad (3.3)
\]

The cleaner compares the segment's index with that of its neighbors and moves some pages between the two to bring them closer to the average. When individual segments get closer to the actual final state, the average index also becomes a better estimate of the cleaning index in the final state. Over time, this process will result in the desired final state where all cleaning indices are equal. Figure 3.4 shows how the locality gathering algorithm can take advantage of increasing locality of reference.
Direct computation of the final index requires an exact knowledge of the access patterns generated by the host and is therefore impractical or even impossible. An important feature of the locality gathering policy is that it is completely adaptive. The final index never has to be exactly known, the system simply evolves towards the best case state. The algorithm's adaptive nature also insures that it will work in all situations regardless of changes in overall Flash utilization and access patterns.

A drawback of the standard algorithm is that it is slow to converge to the final value. The cleaner knows which segments need free space and which ones have too much free space, but only in a relative sense. There is no knowledge of the magnitude of the need, so only a few pages can be moved at a time. Page movements that overshoot the requirements of a segment can actually move the cleaning index further away from the final value, causing the system to become unstable.

If we know the final cleaning index and the time since each segment was last cleaned, we can calculate the final utilization of each segment by solving Equation 2 for \( \mu \). It is easy for the cleaner to keep track of segment cleaning times, and the cleaner can use the average index as a guess for the final one. Equation 3.1 is the actual relationship that eNVy uses to project a guess of the utilization of each segment in the final state. Estimates calculated in this manner are fairly good and allow the cleaner to distribute pages proportionally to the needs of the segments. The TurnoverTime in eNVy is the number of other segments that have been cleaned
since the segment in question was last serviced. This focuses the algorithm on write locality and eliminates any dependences on unpredictable physical time.

\[ ProjectedUtilization[n] = \frac{TurnoverTime[n]}{AverageIndex + TurnoverTime[n]} \]  

(3.1)

Finally, in order to help preserve spatial locality, data is only moved between adjacent segments. If active pages are moved from a segment to a higher numbered segment, they are taken from the beginning of the source segment. Data in this area is the oldest and theoretically least frequently used. The opposite happens when data is moved to a lower numbered segment. This helps to concentrate frequently used pages around segment 0.

When two segments are compared, their projected utilizations are used to figure out how many pages they want to have or give up. We average their needs and transfer 1/4 of the pages needed to bring them to this average. This increases the time it takes to converge to the final data distribution but prevents the cleaner from being too aggressive and sensitive to short term discrepancies. It also keeps pages from jumping back and forth between segments once they have converged to the final state.

A more intelligent algorithm could decide how many pages to move based on the incremental benefit of a transfer. As demonstrated in Figure 3.2, the cleaning cost
Figure 3.5  Segment Utilization over Time
of a segment with a high utilization will be affected much more by a 1% change in utilization than one with lower utilization.

One non-intuitive aspect of the cleaning algorithm is that data is transferred between two segments where either both need more active pages or both don’t need more active pages. A segment that needs pages can benefit from taking a page from a segment also in need, but whose requirements are less. In this case we average their needs and transfer one quarter the difference as above, bringing the two segment’s indices closer to each other. It turns out that this is also necessary to prevent segments from blocking the flow of pages just because their neighbors are in the same situation.

3.4.2 Example

Figure 3.5 shows how the locality gathering algorithm modifies the allocation of data in a 32 segment system in response to a change in the reference stream. Before time 0, a uniform distribution is applied and each segment is at 80% utilization (80% of the space is used for data, 20% is free). At time 0, a 10/90 distribution is substituted. Basically, 90% of the writes go to 10% of the segments. Segments 0 to 2 are considered the high frequency segments, while the rest are considered low frequency. As time progresses, data is moved out of segments with a high frequency of access to reduce their cleaning cost.

Figures 3.6 - 3.9 show in more detail how the cleaning indices and projected final utilizations change to adapt to the reference pattern. When exposed to a uniform
Figure 3.6 Utilization and Cleaning Indices for Uniform Accesses
distribution (Figure 3.6), all segments have a utilization of 80% and a cleaning index of 8.

Figure 3.7 shows what the cleaning index looks like at time 1, shortly after the 10/90 distribution takes effect. The cleaning index of the low frequency segments goes up since their cleaning interval becomes longer, while the opposite occurs in higher frequency areas. This creates a fairly sharp distinction between blocks that need more active data and those who need less. The projected utilization given by Equation 3.4 shows what the cleaner thinks the utilization each of the segments should be. To help reach this situation, some of the data from the high frequency segments has been moved to segments 3 to 11. The hump in this area is caused by the fact that the higher frequency segments that want to give up data are being cleaned more often than the segments that want to receive data. The hump represents a temporary bottleneck.

By time 10 (Figure 3.8), the high frequency segments are close to their final values. The 10% high frequency area is now spread out to about 5 segments since there is less data in each. The data trapped in the bottleneck is also more evenly spread out among the low frequency segments. Figure 3.9 shows what the state of the system is at time 1000, long after it has stabilized.

It is important to note that it takes a fairly long time for eNVy to adjust perfectly to radically different input streams. In a 2 GByte system, each segment contains 16 Mbytes and each must be cleaned many times to properly redistribute the data.
Figure 3.7  Utilization and Cleaning Indices at Time 1
Figure 3.8 Utilization and Cleaning Indices at Time 10
However, initial final state utilization estimates are very close to the real values (compare Figures 3.7 and 3.9) so a more aggressive data movement policy could be used if needed.

3.5 Comparison to Optimal Hand Calculations

We can check to see if the locality gathering algorithm actually converges to the correct value for a simple case like the example above. The goal is to find the optimal utilization of both parts of a bimodal distribution of data in an array which contains 80% data. Equations 3.5 - 3.7 describe the relationship between $\mu_{10}$ and $\mu_{90}$, the utilizations of the 10% high and 90% low frequency access zones respectively.

\[
\mu_{10} = Utilization \ of \ high \ frequency \ area \quad (3.5)
\]

\[
\mu_{90} = \frac{Amount \ of \ low \ frequency \ data}{1 - Fraction \ occupied \ by \ high \ frequency \ area} \quad (3.6)
\]

\[
\begin{align*}
\mu_{90} &= \frac{0.80 \times 0.90}{1 - \frac{0.80 \times 0.10}{\mu_{10}}} \\
\end{align*} \quad (3.7)
\]

and the average cost to clean a segment is given by:
Figure 3.9  Utilization and Cleaning Indices at Time 1000
Average Cleaning Cost = \%Accesses to \mu_{10} area \times Cost to clean \mu_{10} area (3.9)

+ \%Accesses to \mu_{90} area \times Cost to clean \mu_{90} area

= 0.90 \times \frac{\mu_{10}}{1 - \mu_{10}} + 0.10 \times \frac{\mu_{90}}{1 - \mu_{90}} \hspace{1cm} (3.10)

Figure 3.10 shows average cleaning cost as a function of the utilizations of the two locality zones. The lowest cleaning cost is achieved when \mu_{10} is about 0.45 and \mu_{90} is about 0.85. As shown by Figure 3.9, the cleaning algorithm does settle at this state.

3.6 Hybrid Approach

Each of the two previously described algorithms has a desirable effect. The greedy algorithm performs optimally for uniform access distributions while the locality gathering algorithm gives good performance for higher localities of reference.

A hybrid approach was tested that combined the two methods. The segments are partitioned into larger pieces that encompass 4 segments. The locality gathering approach is used to manage free space between partitions while a FIFO cleaning order was used within each partition. Each write gets flushed back to the same partition (not segment) it was read from, where it is written sequentially into the active segment. Figure 3.3 compares the data distribution and movement for the various cleaning methods. Since the hybrid policy allows segments to be cleaned at
different rates, care must be taken to insure that the Flash blocks are cycled evenly. eNVy keeps statistics on the number of write/erase cycles each segment has been exposed to and when the highest frequency segment gets over 100 cycles older than the youngest, a cleaning operation is initiated that swaps the data in the two areas.

As Figure 3.4 shows, the hybrid approach comes close to the performance of the greedy algorithm for uniform access distributions while consistently beating the original attempt at locality gathering. When compared to the Sprite LFS' write costs, using the same definition of write cost, the hybrid approach gets almost identical performance.
3.7 Estimated eNVy Lifetime

For eNVy to be practical, it must have a respectable usable lifetime. For 1 million cycle parts, a page flush rate of 10,000 per second (approximately the rate needed to sustain 10,000 TPS in the system simulated in Chapter 4), and a write cost of 1.86 (the worst case for a uniform distribution), the lifetime of the system under continuous use is given by:

\[
\text{Lifetime} = \frac{\text{WriteCapacity}}{\text{PageWriteRate}}
\]

\[
= \frac{2,048\, Mbytes \times 4,096\, \frac{\text{pages}}{Mbyte} \times 1,000,000\, \text{write/erase cycles}}{10,000\, \frac{\text{flushes}}{sec} \times (1 + 1.86)\, \frac{\text{pages}}{\text{flush}} \times 3600\, \frac{\text{sec}}{\text{hour}} \times 24\, \frac{\text{hours}}{\text{day}}}
\]

\[
= 3,734\, \text{days of continuous use (10.2 years)}
\]

This is well within reasonable limits and is especially appealing since the workload used in the calculation is probably much higher than what can be expected from actual applications over a sustained time.
Chapter 4

Simulation Results

4.1 Introduction

The performance of the eNVy storage system involves many factors including a wide variety of architectural design choices, the efficiency of the cleaning algorithm, and various Flash memory parameters. A simulation of the system was done to provide a reasonably accurate picture of how eNVy performs as a whole to validate the basic concept, as well as allow a more careful study of individual issues and design choices.

4.2 Simulation Overview

The simulation models an eNVy system which include 2 gigabytes of Flash memory, 16 megabytes of battery backed SRAM for the write buffer and 48 megabytes of battery backed SRAM for the page table.

The simulator is driven by a function that simulates the I/O workload needed for the TPC-A database benchmark [17]. The benchmark consists of short database transactions with relatively small amounts of processing which makes it particularly susceptible to becoming I/O bound. The database data itself describes a banking system made up of a number of bank branches, a number of bank tellers at each
branch, and many accounts. Each bank has 10 tellers and 100,000 accounts. Branch, teller, and account balances are kept in three tables made of 100 byte records. A transaction involves an atomic operation consisting of picking an account record at random, changing its balance, then updating the balances of the teller that handled the transaction and the bank where the account is located. In addition to the actual data modification, every update involves searching an index tree to find the records involved. The simulator implements each index tree as a B-Tree with 32 entries per node.

The ratios of bank, teller, and account records described above were scaled to fit our database size. Our 2 GByte system is large enough to manage 15.5 million records at 80% utilization. Account numbers are generated with a uniform distribution and transaction arrival times are exponentially distributed with a mean corresponding to the transaction rate being simulated. Figure 4.1 summarizes of the parameters used in the simulation. While the simulated load is derived from the TPC-A benchmark, we do not make any specific performance claims about actual TPC ratings defined by the standard. Such a rating depends on many other aspects of the database system such as database software and communication performance, not just the storage component.

Unless otherwise specified, the simulations were run with an 80% Flash utilization level at a 10,000 transaction per second request rate. The 2 gigabyte Flash array is divided into 128 segments when viewed by the cleaning algorithm and the hybrid cleaning protocol is used with a partition size of 16 segments.
4.3 Throughput and Latency

The amount of data that has to be written to Flash puts an upper bound on eNVy's maximum throughput. The cleaning algorithm heavily influences this quantity. As the array utilization increases, more time is spent cleaning and less time doing useful work. Figure 4.4 shows how this additional cleaning time can degrade the throughput of the eNVy system. After about 80% utilization, performance drops off steeply, reinforcing our decision to keep at least 20% of the Flash array's storage space free at any given time.

Figure 4.2 demonstrates that throughput increases steadily with the transaction request rate until that rate exceeds the capacity of eNVy's cleaning system. The peak load the simulated system can handle is about 30,000 transactions per second. Figure 4.3 plots the read and write latencies seen by the host under the same conditions as Figure 4.2. On average, reads take about 180ns and writes take 200ns. As the
Figure 4.2  Measured Throughput for Increasing Request Rates

Figure 4.3  I/O Latency for Increasing Request Rates
Figure 4.4 Measured Throughput for Various Levels of Flash Utilization

transaction request rate surpasses eNVy's ability to process them, the write latency jumps dramatically from 200ns to 7.2μs (not shown on the graph). When eNVy receives write requests faster than it can handle them, the buffer gets filled up. If the buffer is full and a copy-on-write is initiated, the controller must flush a page to Flash before it can proceed. This flush may require a segment be cleaned before it can be written. Since the write request that triggered the copy-on-write cannot proceed until the flush is done, it is exposed to much more latency. Figure 4.3 also shows that the combination of being able to interrupt long Flash operations and a write buffer is very effective in hiding the latency of both reads and writes. The I/O latencies are almost constant across the range of workloads as long as the eNVy system can handle sustained throughput at that rate.

The size of the write buffer can have an impact on performance, but increases can have diminishing returns. Figure 4.5 demonstrates this effect. For low I/O rates,
very little time is actually spent servicing data requests, and most of the bandwidth can be used for cleaning. When the I/O rate increases, the cleaner gets less time and the latency of a cleaning operation increases. Since the write buffer is used to hide this latency, larger buffers reduce the waiting time of the host. A buffer size about equal to the size of a segment is capable of reducing write times to 200ns for rates up to 30,000 TPS, the maximum sustainable throughput. The buffer size was measured relative to the segment size since the time spent on a cleaning operation is also proportional to the segment size.

4.4 Performance of the Cleaning Algorithm

The efficiency of the cleaning algorithm depends heavily on the size of the units it chooses to clean. Figure 4.6 shows how the write cost is affected by the number of segments within a partition. Extremes of partition size such as 1 or 128 segments work well in certain situations, but poorly in others. The best overall performance seems to come with a partition size of 16, roughly midway between the extremes. This choice of partition size provides close to optimal performance (low write costs) for uniform access distributions as well as reference patterns with high locality and is used as the default in the other simulation runs.

Figure 4.7 demonstrates how cleaning becomes more efficient as the Flash array is divided into more and more segments. Having a large number of small segments rather than a small number of large segments lets the cleaner to work with a finer
Figure 4.5  Effect of Buffer Size on Write Latency

Figure 4.6  Cleaning Costs vs. Partition Size
Figure 4.7 Cleaning Costs vs. Partition Size

grain size allowing it to come closer to optimal performance. Clearly, making the segment size as small as possible is desirable, but the large size of the erase blocks in the Flash chips puts a lower bound on the size of the segments. Surprisingly, the 128 segment limit imposed by the hardware in our simulated system is right at the knee of the efficiency curve. Dividing the memory into smaller pieces has little effect if each segment already represents less than 1% of the total array. This property is critical in smaller systems.
Chapter 5

Future Work

5.1 Hardware Extensions

There are many low cost performance enhancements that can be made to the basic eNVy architecture. An obvious example is to perform multiple program and erase operations at the same time to different banks of Flash memory. The order in which pages are flushed from the write buffer does not matter, so it is easy to select pages that can be written in parallel. The ability to perform multiple erase operations allows multiple cleaning operations to run at the same time. With 4 to 8 concurrent programming operations, the average time to write a page can drop from 4\(\mu\)s to less than 1\(\mu\)s.

Another low-cost feature available to eNVy is hardware atomic transaction support. Traditional transaction processing systems use some sort of software controlled logging/checkpointing procedure to allow recovery to a consistent state after a transaction abort or a system failure. eNVy automatically copies all modified data from Flash to SRAM as part of its copy-on-write mechanism. The original data in Flash is not destroyed, and it can be used to provide a free shadow copy. An application can roll back a transaction simply by copying data back from Flash. In order to
implement this feature, the controller has to keep track of the location of the shadow copies and protect them from being cleaned. This function has been studied in the context of log file systems [14] which use similar data movement primitives.

A final hardware optimization is to use the comparators normally used to verify write data as a general purpose search engine. Since 256 bytes can be examined each cycle, this kind of search can be over an order of magnitude faster than a sequential processor search. Fast search capability allows small and medium sized data structures to be searched associatively, an operation that is normally considered to be too expensive for practical use. A more complex comparator that can handle greater-than or less-than functions can also speed access to sorted structures. Most of the hardware for this function is already present so there is very little additional cost to support this feature.

5.2 Software Issues

Programs can improve performance by matching their access patterns to suit the eNVy system. Just as in a disk system, there are performance gains to be had from using intelligent data placement. These gains are available but since access times for Flash are so short, the layout optimizations are done for different reasons and have much less of an impact on overall performance. For example, there is a certain penalty for random accesses in the form of TLB misses. The misses occur in both the host’s virtual memory management hardware and eNVy’s internal MMU. The best
way to reduce performance degradation due to TLB misses is to use data structures that exhibit good spatial locality. In the case of search structures, organizations that store more information at each node, such as B-Trees or T-Trees, are preferred to a binary tree.

Programs can also increase the locality of their writes, making the cleaning operation more efficient and extending the lifetime of the array. An example is the layout of a database record. In the TPC example, a 100-byte record is divided into several fields. However, only the 8-byte account balance field is actually modified. If the record is stored as an array of each field rather than an array of records, even an uniform record access pattern will only write to 8% of Flash space occupied by the account balance array. This separation of fields is only possible because of the low random access penalty of a solid-state storage system.

The high rate of I/O that eNVy can provide shifts the performance bottleneck back to the processor and communications subsystems. A multiprocessor system will be needed to fully exploit the potential of a solid-state store. In fact, the best eNVy implementation may be as a distributed shared memory (DSM) system or other parallel system utilizing a larger number of processing nodes, each with smaller amounts of Flash acting as a distributed database. It is in these areas that we look for further performance gains.

Methods for handling long transactions also have to be reexamined. The tradeoffs in a Flash system are very different than in a disk based system and the penalty
for redoing work may be less than locking up large portions of the memory array. Also, standard database query operations should be more efficient on a system where random accesses are not costly. Research in this area is necessary to insure that real applications are not hindered by guidelines that only exist because an underlying disk system is assumed.
Chapter 6

Related Work

There has been a large amount of work done to overcome the limitations of magnetic disks. Caches and disk buffers are the most mature of these enhancements. Caches use pieces of main memory to temporarily store data after it is read in the hopes that it will be used again. They work well for read data with high localities of reference, but have difficulty with uniform access distributions on large datasets. Buffers take advantage of fast sequential disks accesses to prefetch data that will likely be used in the future. Since main memory is typically volatile, neither technique is effective in speeding up persistent writes, which have to be done to a disk for safety.

Non-volatile write buffers attempt to hide the latency of persistent writes by having a synchronous write done to fast non-volatile memory, then asynchronously transferred to disk [1, 3, 12]. This hides the latency of the write, but does not improve the throughput. Studies have shown that for UNIX type file systems, just a few megabytes of battery backed SRAM are needed to almost completely hide the write latency of a disk [12].

RAID (Redundant Array of Inexpensive Disks) systems were developed to use multiple disks in an intelligent way to improve throughput and bandwidth [10, 5, 11].
The ability to store data across N disks allow either N independent accesses or one access with N times the bandwidth to occur in the same time as one I/O operation to a single disk. Unfortunately, individual access times are not improved. RAID arrays are useful in improving throughput for parallel transactions. Large, single threaded applications or transactions do not experience much speedup beyond that provided by improved bandwidth. Furthermore, writes are hindered by the need to calculate error correction information.

Extensive work on in-memory data structures and other aspects of retrieval of data in large memory arrays have been done by researchers in the main memory database field [4, 7, 9]. These systems typically assume that data is kept entirely in memory so that specialized data structures can be used to provide fast access to data. They are hindered by the fact that main memory is volatile and persistent operations still require a disk.

A close relative of eNVy is being developed at MITL [2]. Researchers there argue that Flash has interesting benefits as a memory mapped storage medium, and suggest a similar copy-on-write scheme handled by software in the operating system rather than a hardware controller. The goal of this work is to provide better operating systems support for mobile computers rather than higher performance (although that may be possible too).

Finally, the problem of cleaning and how to minimize write costs for data reclamation has been studied in the context of log structured file systems [13, 14]. eNVy’s
cleaning algorithm is closely related to their algorithms, but has more freedom in what it can do because random accesses in eNVy are much faster than the disks the file systems are built on.
Chapter 7

Conclusion

Flash memory is becoming an increasingly important storage medium. Its high-density, non-volatility, and rapidly decreasing cost have positioned it to become the first practical solid-state mass storage device. In the past year, Flash has reached several milestones, including surpassing EPROMs in sales at Intel, and dropping below DRAM in terms of cost per megabyte. Almost all semiconductors have a competing Flash product, fueled by demand from the expanding portable computing industry.

The premium for Flash products has been paid because they are more portable than standard hard disks. We believe that Flash is also attractive from a performance standpoint. Larger desktop applications such as databases have long been hindered by the disk bottleneck, and a non-volatile solid-state memory cheap enough for mass storage will help reduce the problem.

eNVy provides an efficient interface to a high speed, Flash based, storage system. It uses parallel data transfers, a copy-on-write scheme, and an efficient cleaning algorithm to overcome problems caused by Flash's WORM characteristics, long write latency, and limited cycling ability. The non-volatile memory is presented as a persis-
tent store, allowing for easy software access while maintaining solid-state access times, 180ns for reads, 200ns for writes, The ability to access stable storage this quickly makes eNVy a very good match for today’s computers, especially when running currently I/O bound applications. This performance can be maintained even when providing reliable transaction type operations, eliminating the performance/reliability tradeoff found in disk based systems.

Implementation of a 128 Mbyte prototype is planned using an SBUS interface and a SparcStation host. The system will have too few chips to transfer an entire page in a single memory cycle, so techniques will be tested that can maintain reasonable performance levels even with a lower transfer rate.
Bibliography


