eNVy: A Non-Volatile, Main Memory Storage System

To appear in ASPLOS VI

Michael Wu
Dept. of Electrical and Computer Engineering
Rice University
mikewu@rice.edu

Willy Zwaenepoel
Department of Computer Science
Rice University
willy@rice.edu

Abstract

This paper describes the architecture of eNVy, a large non-volatile main memory storage system built primarily with Flash memory. eNVy presents its storage space as a linear, memory mapped array rather than as an emulated disk in order to provide an efficient and easy to use software interface.

Flash memories provide persistent storage with solid-state memory access times at a lower cost than other solid-state technologies. However, they have a number of drawbacks. Flash chips are write-once, bulk-erase devices whose contents cannot be updated in-place. They also suffer from slow program times and a limit on the number of program/erase cycles. eNVy uses a copy-on-write scheme, page remapping, a small amount of battery backed SRAM, and high bandwidth parallel data transfers to provide low latency, in-place update semantics. A cleaning algorithm optimized for large Flash arrays is used to reclaim space. The algorithm is designed to evenly wear the array, thereby extending its lifetime.

Software simulations of a 2 gigabyte eNVy system show that it can support I/O rates corresponding to approximately 30,000 transactions per second on the TPC-A database benchmark. Despite the added work done to overcome the deficiencies associated with Flash memories, average latencies to the storage system are as low as 180ns for reads and 200ns for writes. The estimated lifetime of this type of storage system is in the 10 year range when exposed to a workload of 10,000 transactions per second.

This research was supported in part by the National Science Foundation under Grant CCR-9116343 and by the Texas Advanced Technology Program under Grant 003604012.

1 Introduction

Traditional random access storage systems use magnetic disks for stable storage. Magnetic disks possess several properties that have enabled them to dominate the storage market, namely low media cost and reasonable access times. However, the mechanical nature of disks makes it difficult to significantly reduce access times, creating a discrepancy between computation time and I/O time that is growing as processors get faster. Methods such as caches, write buffers, and RAID arrays have been developed to alleviate disk bottlenecks. Caches are commonly used to hide the read latency of disks [15] while non-volatile write buffers have been developed to hide write latency [1, 3, 12]. RAID arrays improve available I/O rates by transferring data in parallel [7, 10, 11].

We are exploring an alternative approach, building a persistent storage system using solid-state memories, thereby avoiding the mechanical latencies associated with disks. Solid-state memories provide a factor of 100,000 improvement in access times compared to disks, from about 100ns for memory to 10ms for disks. Rapid decreases in memory prices due to increasing densities have made it feasible to build memory arrays on the order of a few gigabytes in size, although at a cost an order of magnitude or more greater than that of a disk. It is our expectation that for applications whose performance is currently bound by disk random access rates and whose data requirements stay within a few gigabytes, the performance of a solid-state storage system is well worth the extra cost. Examples of such applications include small to medium sized high performance databases.

We argue that access to this permanent storage system should be provided by means of word-sized reads and writes, just as with conventional memory. This interface simplifies data access routines because there is no need to be concerned with disk block boundaries, optimizations for sequential access, or specialized disk "save" formats. Substantial reductions in code size and in instruction pathlengths can result. For backwards compatibility, a simple RAM disk program can make a

Feature	Disk	DRAM	Low Power SRAM	Flash
Read Access	8.3ms	60ns	85ns	85ns
Write Access	8.3ms	60ns	85ns	4 – 10 microsec.
Cost/MByte	$1.00	$35.00	$120	$30.00
Data Retention Current/GByte	0A	1A	2mA	0A

Figure 1: Feature Comparison of Storage Technologies

memory array usable by a standard file system.

Figure 1 compares various characteristics of disks and three memory technologies: DRAM, Flash, and battery backed SRAM. Flash seems attractive for a persistent storage system since it costs less than other memories and needs no power to maintain its contents. In contrast, low power SRAM is about four times as expensive and requires battery backup for persistent data storage. DRAM, the memory used in most computers, can already be found in large arrays but requires more power for data retention than batteries can provide for extended periods of time. It is possible to dump the contents of DRAM to a disk after a power failure, but this can add complexity to the storage system if the subsystem that performs this function must be highly reliable. In addition, the availability of such a system can suffer since data must be recovered from the disk before peak performance can be restored.

Although cheap and inherently non-volatile, Flash memory is not without its drawbacks. Flash memory cannot be updated in-place. Instead, a whole "block" of memory must be erased before it can be reprogrammed. This presents an image much more like a WORM (Write-Once Read-Many) drive than a memory. Furthermore, updates to Flash memory are much slower than updates to conventional memory, and the number of program-erase cycles is limited.

eNVy is a large Flash based storage system that uses a variety of techniques to overcome these deficiencies and to present its contents as a conventional linear array of non-volatile memory with in-place updates. eNVy uses copy-on-write and memory remapping to provide normal in-place update semantics. A relatively small amount of battery-backed SRAM functions as a non-volatile write buffer, hiding the latency of Flash program operations. Space in the Flash array invalidated during the copy-on-write needs to be reclaimed and erased so that it can be made available for new data. A *cleaning* algorithm is used for this purpose. Our cleaning algorithm is similar in function to the Sprite LFS cleaning algorithm [13], but optimized for use with a large Flash array.

Simulation results of a 2 gigabyte eNVy system running under a TPC-A database benchmark workload show that the system can support I/O rates corresponding to 30,000 TPS (transactions per second). Sustained average latencies to the storage system are as low as 180ns for reads and 200ns for writes. We stress the fact that our performance claims relate to the I/O rates only, and not to overall transaction throughput which is dependent on many other factors.

This paper presents the architecture of eNVy, focusing in particular on the techniques used to overcome the deficiencies of Flash memory. The outline of the rest of this paper is as follows. Section 2 provides basic information on the characteristics of Flash memories and the problems encountered when trying to modify data in a Flash array. Section 3 describes the methods that eNVy uses to overcome Flash memory's limitations, while section 4 elaborates on the cleaning algorithm. Simulation results are presented in Section 5. The remaining sections discuss extensions and conclusions.

2 Flash Characteristics

A Flash chip is structurally and functionally very similar to an EPROM, except for the fact that it is electrically erasable [4]. This simple structure allows it to be manufactured in high densities at low cost. Each chip consists of a byte wide array of non-volatile memory cells. An arbitrary read can be performed with DRAM-like access times (under 100ns). Individual bytes can be programmed in 4 to 10μs but cannot be arbitrarily rewritten until the entire device is erased, which takes about 50ms. Newer Flash chips allow some flexibility by dividing the memory array into large independently erasable blocks around 64 Kbytes in size.

Current Flash technology uses a programming method that slightly degrades program and erase times each time these operations are executed. Each chip is guaranteed to program and erase within specific time frames for a minimum number of cycles ranging from 10,000 to 1 million, depending on the manufacturer. A failure of the chip is defined as when a given write or erase operation takes more time than allowed in the specification. The operation might still succeed if more time is allowed. Also, existing data will remain readable. This failure mode is different from that of a disk in that no data is lost.

Manufacturer's specifications appear to be extremely conservative for chips that we have tested. For example, one chip rated for 10,000 cycles programmed in 4μs and erased in 40ms after 2 million cycles, far below the corresponding guaranteed limits of 250μs and 10 seconds. This exceptional performance shows that as the technology matures, Flash has the potential to become very durable.

A Flash chip normally operates in an EPROM-like read only mode. All other functions are initiated by writing commands to an internal Command User Interface (CUI). Commands exist for programming and verifying bytes, erasing blocks, checking status, and suspending long operations. Depending on the chip, several commands are required for a single logical operation. For example, a single byte programming operation usually involves a series of program and verify steps until the desired data is actually changed on the chip.

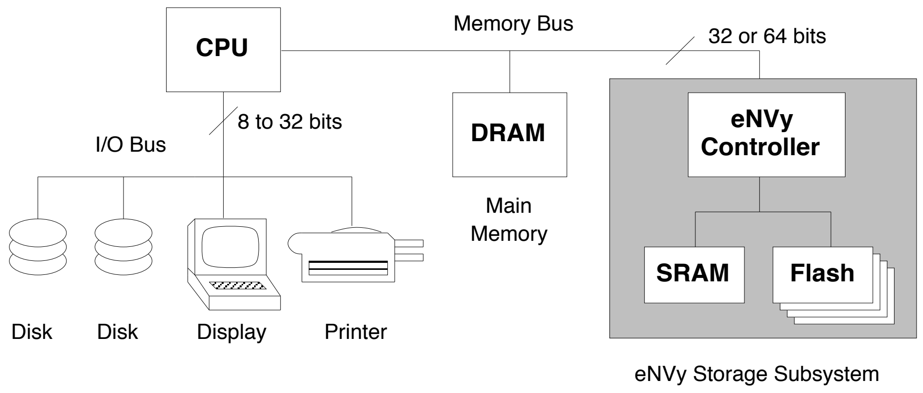

Figure 2: Diagram of eNVy in a Host Workstation

When used to provide a general-purpose non-volatile memory system, Flash technology has three basic deficiencies. First, its bulk erase nature prevents the use of normal update in-place semantics. Second, program and erase operations take much longer than reads. Finally, the number of guaranteed cycles limits the lifetime of the array. The next section describes how eNVy addresses these deficiencies.

3 The eNVy Architecture

eNVy is a large persistent storage system built on top of Flash memory. Its primary goal is to present the Flash memory to a host computer as a simple linear array of non-volatile memory. An additional goal is to achieve access times as close to that of battery backed SRAM as possible (about 100ns). eNVy is designed to reside on a memory bus and to be accessible with the same load and store instructions as main memory. Figure 2 presents a high level diagram of the eNVy storage system and shows where it is located in relation to other computer system components.

3.1 Implementation of a Transparent In-Place Update

The most obvious problem with Flash memory is that it does not allow in-place updates and therefore does not look like normal memory. eNVy uses a copy-on-write scheme to make changes appear to have been done in-place to the host processor.

The Flash array is divided into pages since too much overhead is required to manage data on a word level. A page table maintains a mapping between the linear logical address space presented to the host and the physical address space of the Flash array.

When a write to a particular address in the Flash array is requested by the host, a new copy of the corresponding page is made, including the newly written value. The page table is then updated to point to the modified copy so that further requests for data in that page are directed to the new version.

3.2 Using SRAM as a Write Buffer

The copy-on-write operation described above provides the appearance of in-place updates but is relatively slow since it involves a write to Flash memory. It is much faster to create the new version of the page in another type of memory that is better suited to updates. eNVy includes a relatively small array of battery-backed SRAM for this purpose.

When performing a copy-on-write operation, the original Flash page is copied to an unused page in SRAM. The write request is then executed on the SRAM copy, and the page table is updated to point to the modified page in SRAM. Figure 3 demonstrates the steps in the copy-on-write process. Since changes do not become visible until the page table is updated, the entire copy-on-write appears to be done as a single atomic operation.

After the Flash page is invalidated, the only valid copy of the page is the one in SRAM. For this reason, the SRAM must be battery backed to prevent data loss in the event of a power failure.

The SRAM is managed as a FIFO write buffer. New pages are inserted at the head and pages are flushed from the tail. Pages are flushed from the buffer when their number exceeds a certain threshold. More complex management schemes were discarded because it would be much more difficult to handle them in hardware. The ability to retain pages in SRAM for some time helps to reduce traffic to the Flash array since multiple writes to the same page do not require additional copy-on-write operations. Changes can be made directly in SRAM.

3.3 Page Remapping

The mapping of logical to physical addresses is critical to the integrity of the system, so 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 trade-off must be made when choosing the page size. On one hand, larger pages lead to a smaller page table and lower

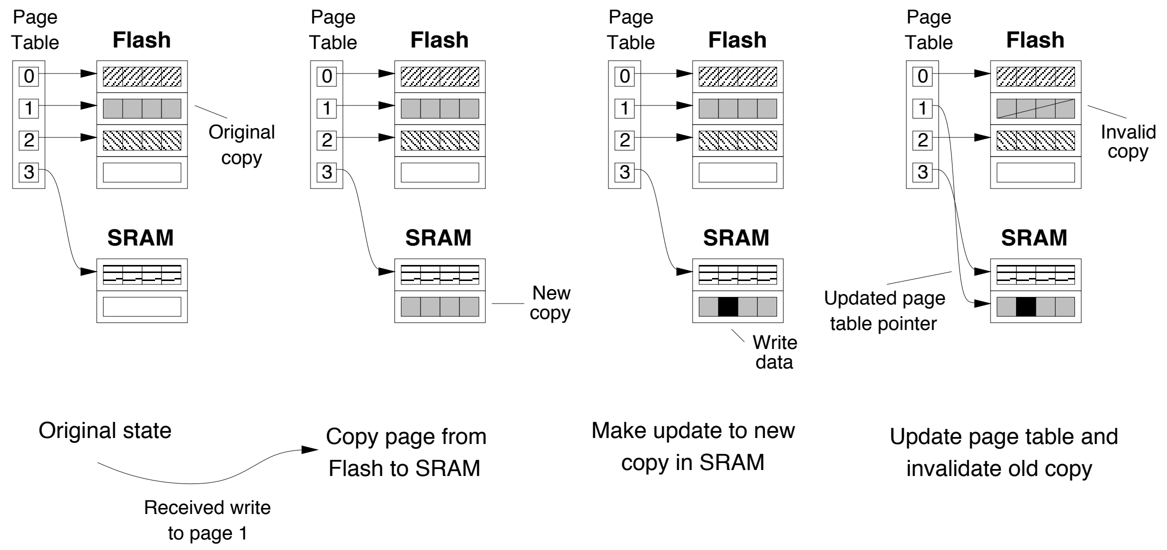
Page Table
Flash
0
1
2
3
Original copy
SRAM
Original state
Received write to page 1
Copy page from Flash to SRAM
New copy
Make update to new copy in SRAM
Write data
Update page table and invalidate old copy
Invalid copy
Updated page table pointer

Figure 3: Steps in the Copy on Write Function (for a write to page 1)

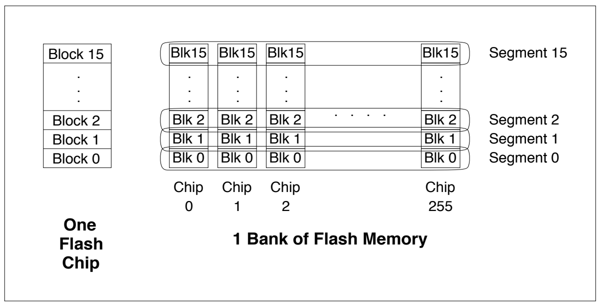

Figure 4: Flash Bank Organization

SRAM requirements. On the other hand, since an entire page has to be written to Flash with every flush, larger pages cause more unmodified data to be written for every word changed. In the interests of maximizing the Flash array's lifetime, the page size should thus be as small as possible.

A tradeoff 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. For every gigabyte of Flash ($30,000), 24 MBytes of SRAM ($2,880) is required for the page table, only about a 10% increase in overall cost (using the costs in Figure 1).

In order to make the copy-on-write operation as fast as possible, we use a 256 byte wide data path between the SRAM and Flash. The Flash array is also organized in banks of 256 (byte wide) chips. This organization allows an entire page to be transferred in just one memory cycle.

3.4 Cleaning

When the SRAM write buffer has been filled to some threshold, eNVy attempts to flush 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 eNVy cleaning system has the same basic responsibility as the Sprite LFS segment cleaner [13], although the reasons for cleaning differ. eNVy recovers space invalidated by the copy-on-write which cannot be overwritten with new data because of the bulk-erase nature of Flash memory. In LFS, data is invalidated as files are modified. The space occupied by this data is not reused because doing so would require many random accesses, which are costly when using a disk.

Flash chips normally allow erasure in large blocks. However, the organization of the Flash array in wide memory banks, as described in Section 3.3, imposes a further restriction on erasure. When combining 256 Flash chips in a bank, the smallest independently erasable unit consists of 256 physical Flash erase blocks (see Figure 4). We refer to this unit as a Flash *segment*. Since a Flash segment is much larger than a typical Sprite LFS segment, an eNVy system has far fewer segments than a Sprite system of comparable size. For example, with current technology, eNVy has 64 segments per gigabyte of storage space while Sprite LFS has 2,000. The importance of this distinction will become clear when we discuss various cleaning algorithms in Section 4.

When eNVy cleans a segment, all of its live data is copied to an empty segment so the original one can be erased and reused. The new segment contains a cluster of live data at its head, with the remaining space being unused and ready to accept data being flushed from the write buffer. The steps involved in the cleaning process are shown in Figure 5. eNVy must always keep one

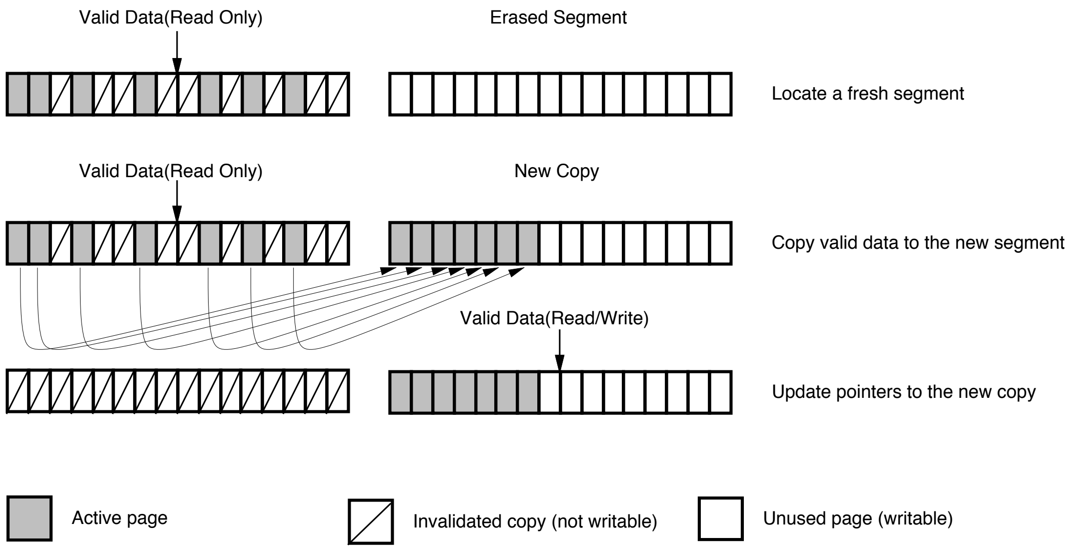

Figure 5: Steps in the Cleaning Process

segment completely erased between cleaning operations so that a free segment is available for the next cleaning operation. The state of the cleaning process is kept in persistent memory so the controller can recover quickly after a failure.

Flushing and cleaning are labeled "long" operations because they take over an order of magnitude longer 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 is 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.

4 Cleaning Policy

The decisions as to which segments to clean, when to clean them, and where to write new data have an important effect on overall performance. These decisions are described by our cleaning policy.

4.1 Flash Cleaning Cost

To measure the amount of work involved in cleaning, we define the term *Flash cleaning cost* to be the number of Flash program operations performed by the cleaning algorithm for every page that is flushed from the write buffer. This ratio is essentially the amount of overhead (cleaning) that has to be done for every useful write to Flash. The cleaning cost differs from the *write cost* [13] used in Sprite LFS in two ways. First, the cleaning cost does not include the cost of reads since cleaning time is dominated by the time it takes to write to Flash memory. Second, it does not include the cost of doing the initial flush from SRAM. The cleaning cost only measures the overhead due to cleaning.

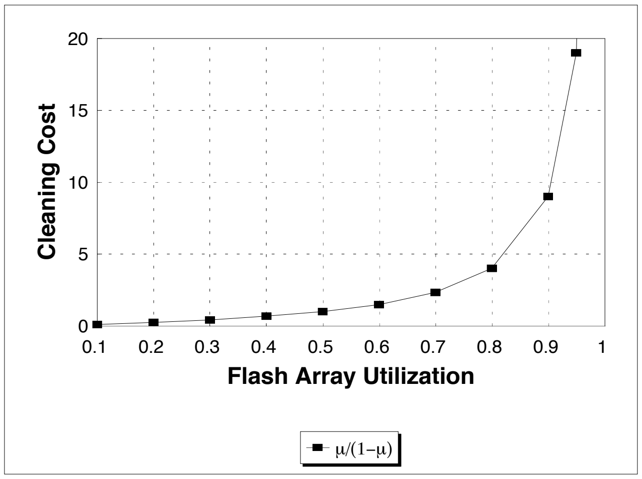

Figure 6: Cleaning Costs for Various Flash Utilizations

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 μ is the utilization of the segment, its cleaning cost is $\frac{\mu}{1-\mu}$. Figure 6 graphically illustrates this cost for different levels of utilization. After about 80% utilization, the cleaning 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 size. Based on Figure 6, a naive cleaning scheme that keeps each segment at 80% utilization would have an average cleaning cost of 4.

One of the primary goals of any cleaning policy is to reduce cleaning costs, resulting in higher performance (less bandwidth is wasted cleaning) and longer array lifetime (fewer program/erase cycles are used). Lower cleaning costs are achieved by managing segments so that the ones being cleaned have a large amount of in-

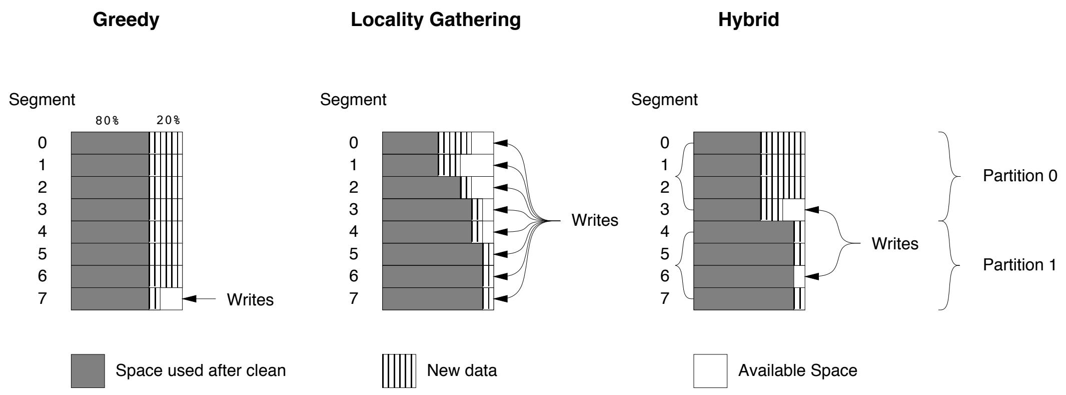

Figure 7: Distribution of Space for Various Cleaning Methods

valid data, providing large amounts of recovered space.

Sprite LFS reduces cleaning costs by separating data with different access frequencies into different segments and picking which segments to clean using a cost/benefit equation [13]. The Sprite LFS cleaner reads several segments at once, sorts the active data by age, and rewrites 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 (hot) data while others get filled with colder data.

We did not use the LFS cleaning policy for several reasons. First, it requires several segments to be cleaned at once. Unlike in LFS, eNVy's segments are defined by the hardware and tend to be both large in size and few in number. Cleaning several at once takes too much time and consumes a large portion of the available resources. Second, in eNVy we do not need to be concerned with the cost of seeks. There is therefore no need to try to force a "log" structure on the data. Instead, we can, without penalty, write to different segments in quick succession. Finally, given the small page size in eNVy, maintaining the age of each page would entail substantial storage overhead.

We present three different cleaning strategies. The first is based on the greedy method which was shown to perform poorly with Sprite LFS. The second strategy allocates varying amounts of data to different segments in the array to reduce cleaning costs. We conclude by showing that a hybrid method based on a combination of the first two algorithms performs better than either one in isolation. While analyzing these policies, it is important to note that since only writes make modifications to the Flash array, only write locality and write access patterns affect cleaning efficiency.

Throughout the remainder of this section we will refer to Figures 7 and 8. Figure 7 summarizes the data distribution and the data movement for the three cleaning methods. Figure 8 presents the cleaning cost for the three algorithms for various localities of reference on a system with 128 segments. In this and subsequent graphs, the numbers on the locality of reference axis represent parameters for a bimodal access distribution. For example, "10/90" means that 90% of all accesses go to 10% of the data, while 10% goes to the remaining 90% of data.

4.2 Greedy Method

Our greedy policy is similar to the one described in the Sprite LFS [13] paper. When there is no space to flush data, the cleaner chooses to clean the segment with the most invalidated space, hoping to recover as much space as possible. After a cleaning operation, further writes are directed to the free space in the newly cleaned segment until it is full, at which time a new cleaning operation is started. Unlike Sprite LFS's policy, our greedy method does not attempt to create a bimodal data distribution using age sorting and cleaning multiple segments at a time for the reasons given in Section 4.1. As predicted by the work on Sprite LFS, the greedy method lowers cleaning costs for uniform distributions but performance suffers as the locality of reference is increased (see Figure 8).

The greedy policy tends to clean segments in a FIFO order. This is fairly intuitive for a uniform distribution. When a segment is cleaned, it is cleared of all invalid data. The uniform distribution invalidates data out of all the segments at similar rates. Therefore, the segment that was most recently cleaned will likely have to wait until every other segment has been cleaned before it is chosen again, demonstrating the FIFO-like cleaning order. For a uniform distribution, the FIFO ordering lowers the utilizations of the cleaned segments by giving the data in each segment the most time to be invalidated between cleans. This results in lower cleaning costs.

What is not so intuitive is that the FIFO behavior continues even at high localities of reference, as long as cold data is still being accessed occasionally. After a segment is cleaned, it accepts flushed data until it is full. Since most of the flushed data is hot, a cold segment that gets cleaned ends up with a mixture of hot and cold data. Cold data is also introduced into the hot segments. Over time, all of the segments end up with the same distribution of hot and cold data when they are cleaned. Data is invalidated out of each segment at the same rate and the algorithm exhibits FIFO behavior just as under a uniform distribution. In contrast to its

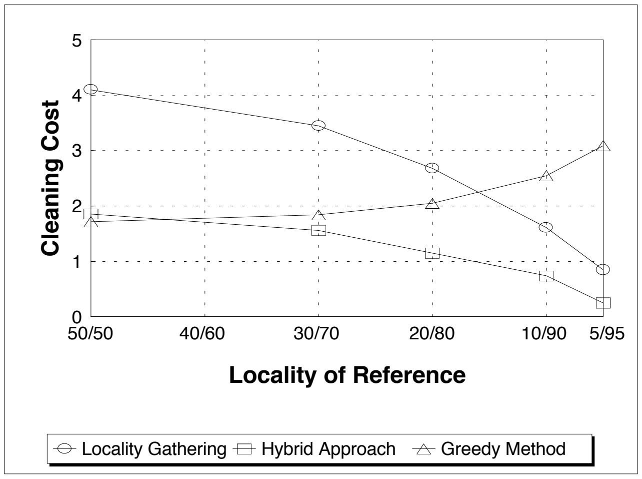

Figure 8: Comparison of Cleaning Algorithms

effect with uniform accesses, the FIFO ordering does not produce low cleaning costs for distributions with high locality. As the locality of reference goes up, the cold area not only grows in size, but also experiences fewer invalidations, which tends to increase its cleaning cost. The opposite occurs with hot data. Since there is more cold data than hot data, the former effect is dominant and the overall cleaning cost goes up.

4.3 Locality Gathering

The locality gathering algorithm attempts to take advantage of high localities of reference. It involves two parts, locality preservation and gathering and active data redistribution.

Hot segments are cleaned more often than cold ones. Therefore, we want to lower their cleaning costs. To do so, we redistribute data between segments to lower the utilizations of hot segments. Since there is a fixed amount of free space in an eNVy system, we must balance the benefits of lowering the utilization of hot segments against the cost of increasing the utilization of cold ones. We use a heuristic to decide how much free space should be present in a segment. The algorithm aims for a situation where the product of the frequency with which a segment is cleaned and its cleaning cost is the same for all segments. Intuitively, this means that a segment that is used ten times more often than another one should have one tenth its cleaning cost.

When a segment is cleaned, the cleaner computes the product of that segment's cleaning cost and the frequency with which it is being cleaned. This value is compared to the average over all segments. If the value of the product for the cleaned segment is above the average, its utilization should be lowered. Otherwise, it should be increased. Pages are transferred between the cleaned segment and its neighbors to bring their products closer to the average.

This cleaning policy moves pages between segments in a manner than encourages locality. In particular, it tries to migrate hot data towards the lower numbered segments while cold data travels in the opposite direction. To accomplish this, we take advantage of the fact that when cleaning a segment, the order of the pages is maintained, i.e., the first valid pages to be read from the old segment are the first to be written into the new segment. Since during normal operation new data is written to the tail of a segment, data near the end tends to be hotter than average. Cold data sinks to the beginning. When pages are moved to a lower numbered segment, they are taken from the end of their original segment while pages headed for a higher segment are taken from the beginning. This tends to gather hot data near segment 0. Care must be taken to prevent flushes from the SRAM write buffer from destroying locality. When a page is placed into the SRAM buffer, we record which segment it comes from. When it is flushed, it is written back to the same segment.

Over time, this process creates a multimodal distribution where hot data and cold data reside in different segments. Redistribution will take advantage of the different needs of each of these segments to find a good allocation for free space in the array, improving overall cleaning. Figure 8 shows how the locality gathering algorithm takes advantage of increasing locality of reference. Unfortunately, for uniform access distributions, the techniques used to take advantage of locality actually prevent cleaning performance from being improved. The data distribution procedure allocates the same amount of data to each segment since they are all accessed with the same frequency. Since pages are flushed back to their original segments to preserve locality, all segment always stay at 80% utilization, leading to a fixed cleaning cost of 4.

The locality gathering policy allows segments to be cleaned at different rates so care must be taken to insure that segments are cycled evenly. eNVy keeps statistics on the number of program/erase cycles each segment has been exposed to and when the oldest segment gets over 100 cycles older than the youngest, a cleaning operation is initiated that swaps the data in the two areas. This leads to an even wearing of the segments.

4.4 Hybrid Approach

The greedy and FIFO algorithms perform well for uniform access distributions, while the locality gathering algorithm gives good performance for higher localities of reference. The hybrid approach combines the two methods. Several adjoining segments are gathered into a single *partition*. The locality gathering approach is used to manage pages between partitions, while a FIFO cleaning order is used within each partition. FIFO was chosen over greedy method because it is simpler to implement and produces the same cleaning cost. Each write gets flushed back to the same partition (not segment) it was read from, where it is written sequentially into the active segment within the partition.

The intuition behind the hybrid approach is as follows. The locality gathering method, in effect, sorts the

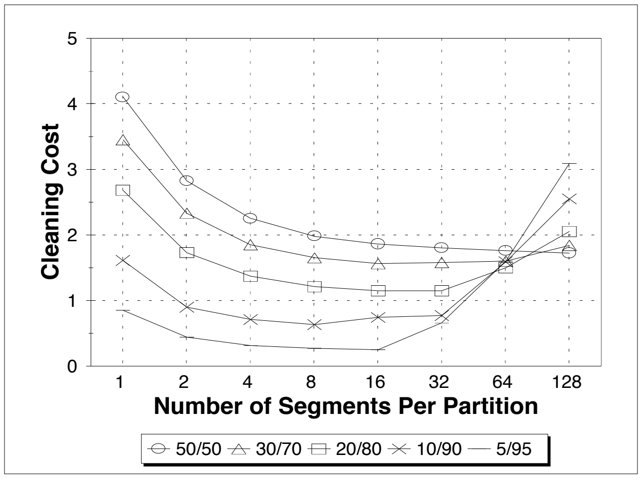

Figure 9: Cleaning Costs vs. Partition Size

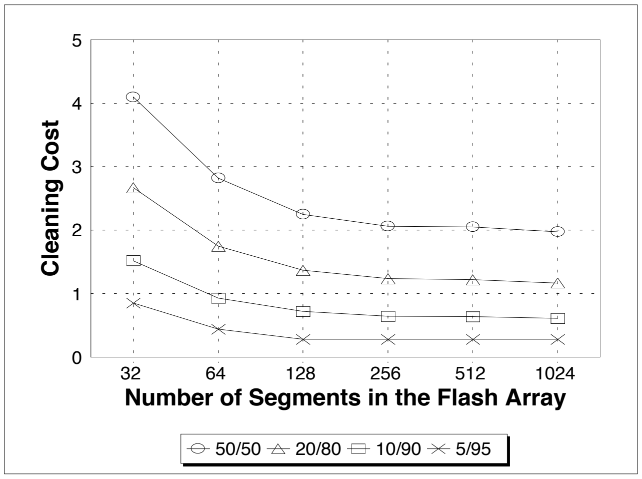

Figure 10: Cleaning Costs vs. Number of Segments

segments by their frequency of access to keep data of like frequencies of access together. Between segments that share the same frequency, accesses are essentially uniform, a case that is handled well by the FIFO algorithm. Grouping the segments into partitions allows the cleaner to break the array into uniformly accessed pieces large enough to be used by the FIFO algorithm, while at the same time small enough that the locality gathering method can efficiently separate data with varying localities of reference.

The efficiency of the resulting algorithm depends on the size of the partitions. Figure 9 shows how the cleaning cost varies as a function of the number of segments per partition for a 128 segment array. Extreme values of the partition size such as 1 or 128 segments correspond to the locality gathering and FIFO algorithms, with their advantages and disadvantages. The lowest overall cleaning cost occurs with a partition size of 16. This choice of partition size provides good performance (low cleaning costs) for uniform access distributions as well as reference patterns with high locality. As Figure 8 shows, with a partition size of 16, the hybrid approach comes close to the performance of the greedy algorithm for uniform access distributions while consistently beating pure locality gathering.

4.5 The Effect of Number of Segments

The performance of the cleaning algorithm is also related to the size of the segments. Smaller segments allow the cleaner to work with a finer granularity and achieve greater efficiency. Clearly, making the segment size smaller is desirable, but this is limited by the physical size of erase blocks as described in Section 3.4.

In Figure 10 we demonstrate how, for a fixed size array and a fixed number of partitions, the cleaning costs for the hybrid algorithm vary as a function of the number of segments the array is divided into. Cleaning efficiency does get better as the system is divided into more and more segments. However, after each segment represents less than 1% of the array, further gains are marginal. This characteristic is especially important for smaller systems that have fewer chips to work with.

5 Simulation Results

The performance of the eNVy architecture involves many factors including a wide variety of architectural design choices, Flash memory parameters, and various timing constraints. A simulation of the system was done to provide a reasonably accurate picture of how eNVy performs as a whole, as well as to allow more careful study of individual issues. This section discusses the hardware implementation we simulated, the workload we presented to it, and the results obtained from the simulation.

5.1 Simulated Hardware

A detailed diagram of the simulated eNVy storage system is shown in Figure 11. The eNVy controller is responsible for translating memory accesses from the host bus into appropriate operations in Flash or SRAM as well as performing background maintenance work on the Flash array. It 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 hardware memory controller handles all low level operations such as reads from the SRAM or Flash arrays and the copy-on-write function. In order to be able to complete a copy-on-write operation in one cycle, the controller contains logic that can check the status of the Flash chips in parallel rather than having to poll 256 individual chips. More complex tasks, such as the cleaning, are handled by the RISC CPU (hereafter called the cleaning processor). The memory controller and the cleaning processor cooperate with each other using a shared register as a communications link. This register

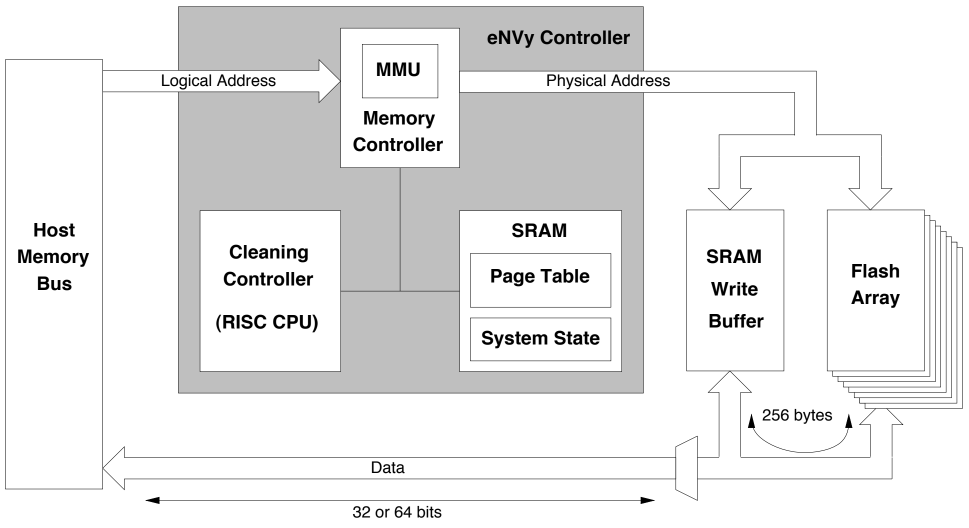

Figure 11: Block Diagram of eNVy Architecture

is polled by the cleaning processor to avoid interrupt latencies. The combination of the hardware memory controller and the cleaning processor is powerful enough to handle all internal eNVy functions, and the host processor is never interrupted to do any work. This eliminates a potentially large source of latency and removes the need to make any operating system modifications on the host side.

The address provided by the host is treated as a logical address and is translated into a SRAM or Flash address using a page table, which is kept in battery backed SRAM as described in Section 3.3. A memory-management unit (MMU) acts as a cache of recently used mappings to make this translation faster. When a copy-on-write is executed, the page table mapping is updated in parallel with the data transfer to further reduce latency. Extra space in the SRAM used by the page table can hold recovery and other system state information.

The simulation models an eNVy system which includes 2 gigabytes of Flash memory divided into 128 individually erasable segments (each 16 megabytes in size). The partition size was fixed at 16 segments based on the results of Section 4.4. A system of this size requires 48 megabytes of battery backed SRAM for the write buffer and 16 megabytes for the write buffer. The buffer size is chosen to be the size of one segment. The total cost of such a system (again, using the numbers from Figure 1) is estimated to be about $70,000. The Flash and SRAM memories are much more expensive than the other components of the system and represent the bulk of the price. An eNVy system costs about one quarter of a pure SRAM system of the same size ($250,000).

Details of the memory organization and access times are described in Figure 12. This table lists the raw characteristics of the chips themselves. While the access times of the chips themselves are 100ns, a memory access involves other factors such as propagation delays and control signal generation. To account for this, 60ns is added to each access, raising the simulated latency of a Flash read or SRAM operation to 160ns.

5.2 Simulated Workload

The simulator is driven by the I/O workload generated by the TPC-A database benchmark [5]. The benchmark consists of short database transactions with relatively small amounts of processing which makes it particularly susceptible to becoming I/O bound.

TPC-A models a banking transaction system made up of several banks, bank tellers, and individual accounts such that for every bank, there are 10 tellers, each of which is responsible for 10,000 accounts. Balance information for each bank, teller, and account is kept in the form of a 100 byte record. Each transaction involves an atomic operation consisting of changing the balance of an individual account and updating the corresponding bank and teller records to reflect the change. For each transaction, three index trees have to be searched to find the desired records, and three actual records have to be modified. The simulator implements each index tree as a B-Tree with 32 entries per node. The database can be scaled to fit any storage system using the ratios described above. Our system is large enough to manage 15.5 million account records.

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 12 includes 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 as 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.

Flash Parameters	
Flash array size	2 Gbytes
Flash chip type	1 Mbyte x 8 bits
# of Flash chips	2048
# of Flash banks	8
# of Flash chips/bank	256
Read time	100ns
Write time	100ns
Program time	4000ns
Erase time	50ms
Erase blocks/chip	16

SRAM Parameters (write buffer)	
SRAM array size	16 MBytes
SRAM chip size	64 Kbytes x 8
# of SRAM chips	256
# of SRAM banks	1
# of SRAM chips/bank	256
Read time	100ns
Write time	100ns

TPC Parameters	
BTree size	32 pointers/node
Branch records	155
# of index levels	2
Teller records	1550
# of index levels	3
Account records	15.5 million
# of index levels	5

Figure 12: eNVy Simulation Parameters

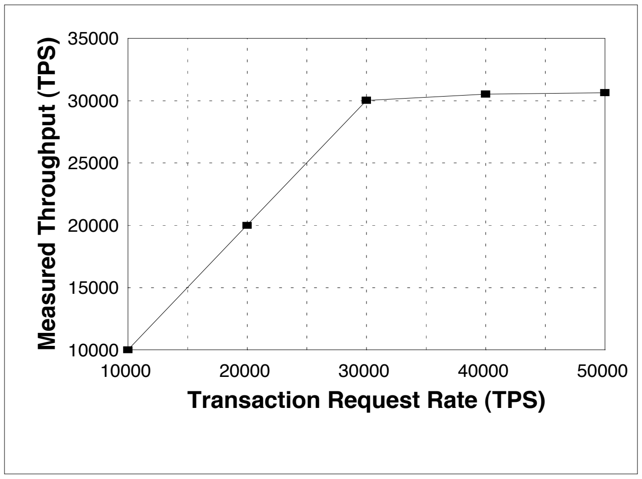

Figure 13: Throughput for Increasing Request Rates

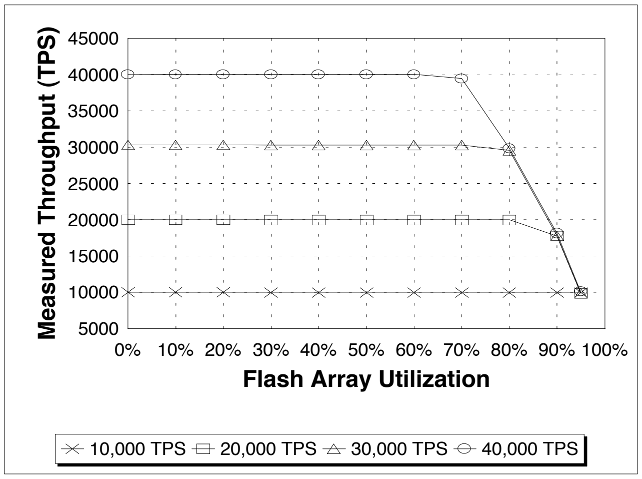

Figure 14: Throughput for Various Levels of Utilization

5.3 Throughput

Figure 13 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.

The amount of data that has to be written to Flash puts an upper bound on eNVy's maximum throughput. The efficiency of the cleaning algorithm heavily influences this quantity. As the array utilization increases, more time is spent cleaning and less time doing useful work. Figure 14 shows how the additional cleaning time degrades 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.

At a utilization of 80% and a transaction rate of 30,000 TPS, the eNVy system is almost never idle. Under these conditions, approximately 40% of the time is servicing reads. Most of the remaining time is spent either cleaning (30%), flushing (15%), or erasing (15%). All of the functions besides reading are used solely to overcome the drawbacks of writing to Flash memory. However, even if they could be completely eliminated, as in a battery backed SRAM array, throughput would only increase by a factor of 2.5. This factor is fairly small because database workloads typically perform many more reads than writes, only the latter of which requires extra work to be done in Flash. This drop off in peak performance has to be considered in the context of the near 400% reduction in price obtained by using Flash instead of SRAM.

5.4 Latency

Figure 15 plots the read and write latencies seen by the host under the same conditions as Figure 13. Until the transaction rate gets near the system's maximum throughput, I/O latencies for both types of access are almost constant, about 180ns for reads and 200ns for writes. As the rate surpasses eNVy's ability to process them, the write latency jumps dramatically from 200ns to 7.2μs and slowly climbs to 7.6μs.

When eNVy receives write requests faster than it can handle them, the write buffer fills up. If the buffer is

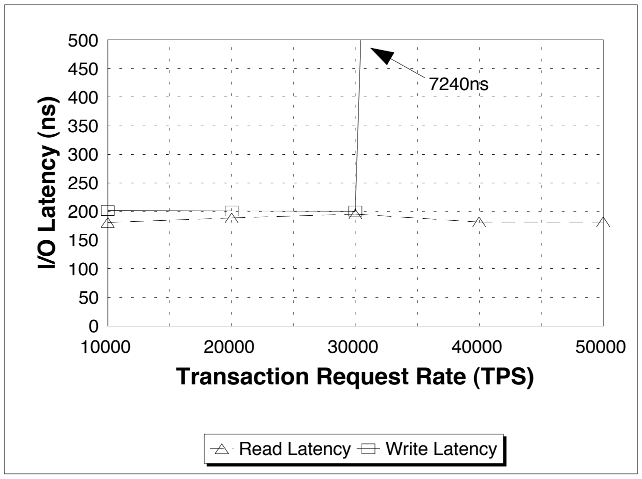

Figure 15: I/O Latency for Increasing Request Rates

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 15 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 times simulated are much closer to a Flash read time than longer operations such as a page flush or cleaning operation. Even writes exhibit close to SRAM access times.

5.5 Estimated eNVy Lifetime

For eNVy to be practical, it must have a respectable usable lifetime. We give a sample calculation using a 10,000 TPS workload as an example of Flash's durability. At this rate, the simulator reports that 10,376 pages are being flushed every second at a cleaning cost of 1.97. The lifetime of the system is equal to the number of pages than can be written to it in its lifetime (write capacity) divided by the rate that pages are actually being written. For 1 million cycle parts in a system being used continuously, this ratio is:

$$Lifetime = \frac{WriteCapacity}{PageWriteRate}$$

$$= \frac{2{,}048 Mbytes * 4{,}096 \frac{pages}{Mbyte} * 1{,}000{,}000 cycles}{10{,}376 \frac{flushes}{sec} * (1+1.97) \frac{pages}{flush} * 86400 \frac{secs}{day}}$$

$$= 3,151\ days\ of\ continuous\ use\ (8.63\ years)$$

This is well within the 3 to 5 year lifespan of most modern computer equipment, especially considering the high sustained workload it is being exposed to. While it is true that the lifetime is proportional to the size of the array (an array half the size has half the lifetime), future generations of Flash chips should provide enough cycles to make almost any size array feasible.

6 Hardware Extensions

There are several 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 affect correctness so it is easy to select pages that can be written in parallel. Parallel erasures allow multiple cleaning operations to run at the same time. With the cleaner executing 4 to 8 concurrent programming operations, the average time to flush a page can drop from 4μs (see Figure 12)to less than 1μs.

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.

7 Related Work

There has been other work done in areas relevant to eNVy. Researchers at MITL [2] 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 although with lower performance. Others have studied the problem of how to minimize write costs for data reclamation in log structured file systems [13, 14]. The benefits possible from having fast access to stable storage were demonstrated by advocates of non-volatile write caches [1, 3, 12]. Finally, 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 [6, 8, 9]. Some particularly interesting work was done at IBM as part of their Starburst project [8] in which their database was tested with two storage components, one that uses memory mapped data structures and one that uses a standard cached disk model. Even though both systems log changes to disk and had enough memory to keep the entire database memory resident, the one that uses memory data structures performed substantially better.

8 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 semiconductor manufacturers 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 can 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 persistent store, allowing for easy software access while maintaining near-SRAM 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 trade-off 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.

References

[1] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file system. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.

[2] R. Caceres, F. Douglis, K. Li, and B. Marsh. Operating system implications of solid-state mobile computers. Technical Report MITL-TR-56-93, Matsushita Information Technology Laboratory, May 1993.

[3] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 327–335, 1989.

[4] Intel Corporation. Flash memory, 1994.

[5] Transaction Processing Performance Council. TPC benchmark A standard specification rev 1.1.

[6] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, pages 509–516, December 1992.

[7] G. A. Gibson. Performance and reliability in redundant arrays of inexpensive disks. In *1989 Computer Measurement Group Annual Conference Proceedings*, pages 1–17, December 1989.

[8] T. Lehman, E. Shekita, and L. Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, pages 555–566, December 1992.

[9] L. Li and J.F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, pages 177–187, December 1988.

[10] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of SIGMOD '88*, pages 109–116, 1988.

[11] A. L. N. Reddy and P. Banerjee. An evaluation of multiple disk i/o systems. *IEEE Transactions on Computers*, 38(12):1680–1690, December 1989.

[12] C. Reummler and J. Wilkes. UNIX disk access patterns. In *Proceedings of the 1993 Winter Usenix Conference*, pages 405–420, January 1993.

[13] M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[14] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log structured file system for unix. In *Proceedings of the 1993 Winter Usenix Conference*, pages 307–326, January 1993.

[15] A. J. Smith. Disk cache – miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.