Exposing and exploiting internal parallelism in MEMS-based storage

Steven W. Schlosser Jiri Schindler Anastassia Ailamaki
Gregory R. Ganger

March 2003
CMU-CS-03-125

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

MEMS-based storage has interesting access parallelism features. Specifically, subsets of a MEMStore's thousands of tips can be used in parallel, and the particular subset can be dynamically chosen. This paper describes how such access parallelism can be exposed to system software, with minimal changes to system interfaces, and utilized cleanly for two classes of applications. First, background tasks can utilize unused parallelism to access media locations with no impact on foreground activity. Second, two-dimensional data structures, such as dense matrices and relational database tables, can be accessed in both row order and column order with maximum efficiency. With proper table layout, unwanted portions of a table can be skipped while scanning at full speed. Using simulation, we explore performance features of using this device parallelism for an example application from each class.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This work is funded in part by NSF grant CCR-0113660 and the MARCO/DARPA Center for Circuits, Systems and Software (C2S2).

Keywords: MEMS-based storage, MEMS, parallelism, storage interface, database

1 Introduction

MEMS-based storage is an exciting new technology [3, 32]. Using thousands of MEMS read/write heads, data bits can be persistently stored on and retrieved from media coated on a rectangular surface [3, 11, 31]. Because of their size and operation, MEMS-based storage devices (MEMStores) are expected to have sub-millisecond random access times, low power dissipation, and disk-like volumetric density. These characteristics represent compelling advantages relative to disk drives.

Just like when disk arrays entered the marketplace, practicality dictates that the initial command set for MEMStores be SCSI-like, with READs and WRITEs to ranges of a linear logical block number (LBN) space. When organized for this institutionalized interface, a MEMStore looks much like a (very fast) disk in terms of performance characteristics: it takes a substantial amount of time to position before accessing data with high bandwidth. Sequential transfers are much more efficient than localized accesses which are more efficient than random accesses. Thus, standard system software principles for accessing storage devices apply. In fact, previous studies have shown that MEMStore-specific optimizations to existing scheduling and data placement algorithms yield only small benefits on the order of 10% [12]. This means that MEMStores can be quickly and easily integrated into current systems.

There is, however, one MEMStore performance characteristic that would not be exploited by the standard model of storage: its interesting form of parallelism. Specifically, a subset of the 1000s of read/write tips can be used in parallel to provide high bandwidth media access, and the particular subset does not have to be statically chosen. In contrast to the disk arms in a disk array, which can each seek to independent locations concurrently, all tips are constrained to access the same relative location in their respective regions. For certain access patterns, however, dynamically selecting which subsets of tips should access data can provide great benefits to applications.

This paper describes the available degrees of freedom MEMStores can employ in parallel access to data and how they can be used for different classes of applications. We describe minor extensions to a SCSI-like interface that minimize system changes while exposing device parallelism. Specifically, exposing a few parameters allows external software functions to compute, for any given LBN, an *equivalence class* which is the set of LBNs that can be accessed in parallel with that LBN, and a *conflict relation*, which are those LBNs from which only one can be selected because of a shared physical resource. Given these functions, external software can exploit the parallelism to achieve up to a $100\times$ increase in effective bandwidth for particular cases.

We illustrate the value of exploiting this parallelism for two classes of usage: one based on special-case scheduling and one based on special-case data layout. First, background applications can access equivalent LBNs during foreground accesses to enjoy a form of free bandwidth [18]; that is, they can access the media with zero impact on foreground requests. We quantify how well this works for full-device read scans (e.g., data integrity checking or backup). Specifically, a random workload of 4 KB requests utilizes only 40% to 80% of the available parallelism. By exploiting the remaining available parallelism for background transfers, we are able to utilize the rest for useful tasks.

Second, matrix computations and relational databases serialize storage representations on the most likely dimension (e.g., row order) but sometimes access data on the other (e.g., column order). Such an access is very inefficient, requiring a full scan of the matrix or table to access a fraction of the data. With proper layout and parallel READ and WRITE commands, such selective access can be much more efficient. For example, with tables stored on a MEMStore, we show that scanning

Device capacity	3.46 GB
Average random seek	0.56 ms
Streaming bandwidth	38 MB/s

Table 1: **Basic MEMS-based storage device parameters.** MEMStores will have a capacity of several GB, sub-millisecond random seek times, and streaming bandwidth on par with disk drives.

one column in a table with 9 columns and 10 million records takes one ninth of the full table scan time.

The remainder of this paper is organized as follows. Section 2 describes MEMS-based storage and its interesting form of parallelism. Section 3 discusses extensions to a SCSI-like interface for exposing and using this parallelism. Section 4 describes our experimental setup. Section 5 evaluates example uses of the parallelism interfaces. Section 6 discusses related work. Section 7 summarizes and concludes the paper.

2 MEMS-based storage devices

Microelectromechanical systems (MEMS) are mechanical structures on the order of 10–1000 μm fabricated on the surface of silicon wafers [20, 33]. These microstructures are created using photolithographic processes similar to those used to manufacture other semiconductor devices (e.g., processors and memory) [9]. MEMS structures can be made to slide, bend, and deflect in response to electrostatic or electromagnetic forces from nearby actuators or from external forces in the environment. A MEMS-based storage device uses MEMS for positioning of recording elements (e.g., magnetic read/write heads).

Practical MEMStores are the goal of major efforts at several research centers, including IBM Zurich Research Laboratory [31], Hewlett-Packard Laboratories [13], and Carnegie Mellon University [4]. While actual devices do not yet exist, Table 1 shows their predicted high-level characteristics. This section briefly describes how MEMStores work and the interesting form of internal parallelism that they exhibit.

2.1 MEMS-based storage basics

Most MEMStore designs, such as that illustrated in Figure 1, consist of a media sled and an array of several thousand probe tips. Actuators position the spring-mounted media sled in the X-Y plane, and the stationary probe tips access data as the sled is moved in the Y dimension.[1] Each read/write tip accesses its own small portion of the media, which naturally divides the media into *squares* and reduces the range of motion required by the media sled. For example, in the device shown in Figure 1, there are 100 read/write tips and, thus, 100 squares.

Data are stored in linear columns along the Y dimension. As with disks, a MEMStore must position the probe tips before media transfer can begin. This positioning is done by moving the sled in the X direction, to reach the right column, and in the Y direction, to reach the correct starting offset within the column. The X and Y seeks occur in parallel, and so the total seek time is the

[1]Some devices, such as the IBM Millipede, fix the media in place and position the probe tip array instead [30, 31]. The relative motions of the tips and media are the same.

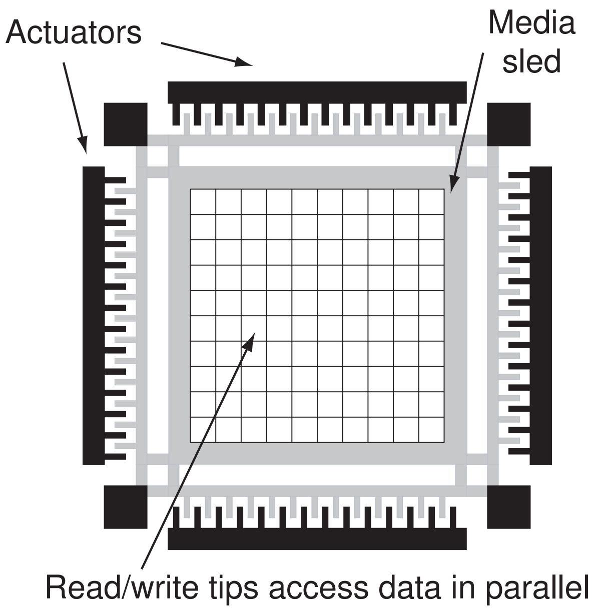

Figure 1: **High-level view of a MEMStore.** The major components of a MEMStore are the sled containing the recording media, MEMS actuators to position the media, and the read/write tips that access the media. This picture emphasizes the media organization, which consists of a two-dimensional array of *squares*, each of which is accessed by a single read/write tip (not shown). As the media is positioned, each tip accesses the same position within its square, thus providing parallel access to data.

maximum of the two independent seek times. Once the media sled is positioned, active read/write tips access data as the sled is moved at a constant rate in the Y direction.

As in disks, data are stored in multi-byte sectors, such as 512 bytes, to reduce the overhead of extensive error correction coding (ECC). These sectors generally map one-to-one with the LBNs exposed via the device interface. Unlike disks, a MEMStore stripes each sector across many tips for two reasons: performance and fault tolerance. A single tip's transfer rate is quite low, and transferring an entire sector with a single tip would require $10\times$ more time than a random seek. In addition, some of the 1000s of probe tips will be defective, and encoding each sector across tips allows the ECC to cope with tip failures. We assume throughout this paper that data are striped across multiple tips for these reasons, and that data are grouped together into 512 byte LBNs.

Once striping is assumed, it is useful to consider that the number of active tips has been reduced by the striping factor (the number of tips over which a single LBN has been striped), and that each tip accesses a single, complete 512 byte LBN. In this way, there is a virtual geometry which is imposed on the physical media, one which exposes full 512 byte LBNs. The example shown in Figure 1 has, in reality, 6400 read/write tips with each LBN striped over 64 tips. But once striping is assumed, the virtual device shown in the figure has only 100 read/write tips, each accessing a single (striped) LBN at a time.

Given the expected SCSI-like interface, MEMStores should assign LBNs to physical locations in accordance with the general expectations of host software: that sequential LBNs can be streamed with maximum efficiency and that similar LBNs involve shorter positioning delays than very different ones. As with disks, the focus is on the former, with the latter following naturally.

Figure 2 shows how LBN numbers are assigned in a simple device. Starting in the first square,

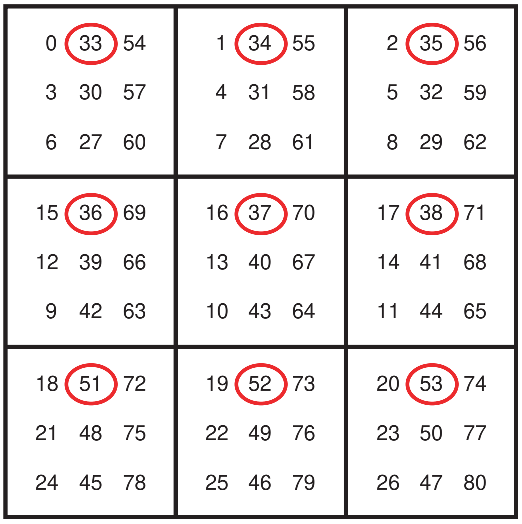

Figure 2: **Data layout with an equivalence class of LBNs highlighted.** The LBNs marked with ovals are at the same location within each square and, thus, comprise an equivalence class. That is, they can potentially be accessed in parallel.

ascending LBN numbers are assigned across as many squares as can be accessed in parallel to exploit tip parallelism. In this example, three LBNs can be accessed in parallel, so the first three LBNs are assigned to the first three squares. The next LBNs are numbered downward to provide physical sequentiality. Once the bottom of the squares is reached, numbering continues in the next set of squares, but in the upward direction until the top of the squares is reached. This *reversal* in the LBN numbering allows the sled to simply change direction to continue reading sequential data, maintaining the expectation that sequential access be fast.

It is useful to complete the analogy to disks, as illustrated in Figure 3. A MEMStore cylinder consists of all LBNs that can be accessed without repositioning the sled in the X dimension. Because of power constraints, only a subset of the read/write tips can be active at any one time, so reading an entire cylinder will require multiple Y dimension passes. Each of these passes is referred to as a track, and each cylinder can be viewed as a set of tracks. In Figures 2 and 3, each cylinder has three tracks. As in disks, sequential LBNs are first assigned to tracks within a cylinder and then across cylinders to maximize bandwidth for sequential streaming and to allow LBN locality to translate to physical locality.

2.2 Parallelism in MEMS-based storage

Although a MEMStore includes thousands of read/write tips, it is not possible to do thousands of entirely independent reads and writes. There are significant limitations on what locations can be accessed in parallel. As a result, previous research on MEMStores has treated tip parallelism only as a means to increase sequential bandwidth and to deal with tip failures. This section defines the sets of LBNs that can potentially be accessed in parallel, and the constraints that determine which subsets of them can actually be accessed in parallel.

When a seek occurs, the media is positioned to a specific offset relative to the entire read/write

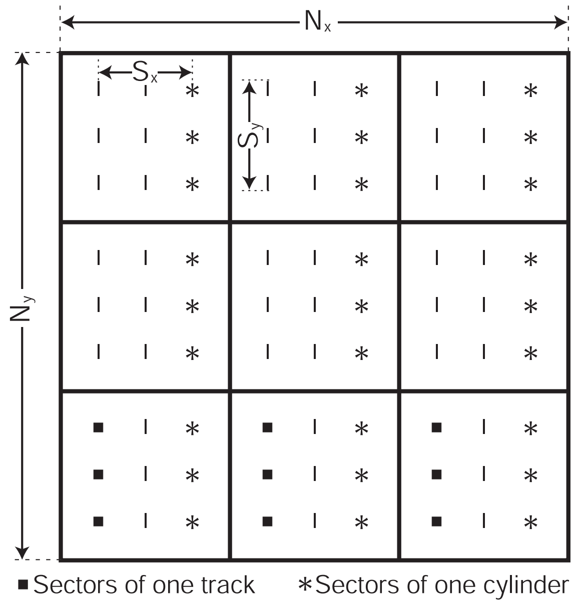

Figure 3: **Terminology.** This picture illustrates two things: the organization of LBNs into tracks and cylinders, and the geometric parameters of the MEMStore. Cylinders are the groups of all LBNs which are at the same offset in the X dimension. In this picture, all of the LBNs of a sample cylinder are marked as stars. Because the number of LBNs that can be accessed at once is limited by the power budget of the device, a cylinder is accessed sequentially in tracks. The LBNs of a sample track are marked as squares in this picture. Three tracks comprise a single cylinder, since it takes three passes to access an entire cylinder. The parameters N_x and N_y are the number of squares in the X and Y directions, and S_x and S_y are the number of LBNs in a single square in each direction.

tip array. As a result, at any point in time, all of the tips access the same locations within their squares. An example of this is shown in Figure 2 in which LBNs at the same location within each square are identified with ovals. This set of LBNs form an *equivalence class*. That is, because of their position they can potentially be accessed in parallel. It is important to note that the size of an equivalence class is very small relative to the total number of LBNs in a MEMStore. In the 3.46 GB device described in Table 1, the size of an equivalence class is 100, meaning that only 100 LBNs are potentially accessible in parallel at any point out of a total of 6,750,000 total LBNs in the device.

Only a subset of any equivalence class can actually be accessed at once. Limitations arise from two factors: the power consumption of the read/write tips, and components that are shared between read/write tips. It is estimated that each read/write tip will consume 1–3 mW when active and that continuously positioning the media sled would consume 100 mW [26]. Assuming a total power budget of 1 W, only between 300 and 900 read/write tips can be utilized in parallel which, for realistic devices, translates to 5–10% of the total number of tips. This gives the true number of LBNs that can *actually* be accessed in parallel. In our example device, perhaps only 10 of 100 LBNs in an equivalence class can actually be accessed in parallel.

In most MEMStore designs, several read/write tips will share physical components, such as read/write channel electronics, track-following servos, and power buses. Such component sharing makes it possible to fit more tips, which in turn increases volumetric density and reduces seek distances. It also constrains which subsets of tips can be active together, reducing flexibility in accessing equivalence classes of LBNs.

For each LBN and its associated equivalence class, a *conflict relation* can be defined which restricts the equivalence class to reflect shared component constraints. This relation does not actually reduce the *number* of LBNs that can be accessed in parallel, but will affect the choice of *which* LBNs can be accessed together. As real MEMStores have not yet been built, we have no real data on which components might be shared and so have not defined any realistic conflict relations. Therefore, we intend to address conflict relation handling in future work.

Figure 2 shows a simple example illustrating how LBNs are parallel-accessible. If one third of the read/write tips can be active in parallel, a system could choose up to 3 LBNs out of a given equivalence class (shown with ovals) to access together. The three LBNs chosen could be sequential (e.g., 33, 34, and 35), or could be disjoint (e.g., 33, 38, and 52). In each case, all of those LBNs would be transferred to or from the media in parallel.[2]

Some MEMStore designs may have an additional degree of freedom: the ability to microposition individual tips by several LBNs along the X dimension. This capability exists to deal with manufacturing imperfections and thermal expansion of the media due to ambient heat. Since the media sled could expand or contract, some tips may need to servo themselves slightly to address the correct columns. By allowing firmware to exploit this micropositioning, the equivalence class for a given LBN grows by allowing access to adjacent cylinders. MEMStore designers indicate that micropositioning by up to 5 columns in either direction is a reasonable expectation. Of course, each tip can access only one column at a time, introducing additional conflict relations.

For example, suppose that the device shown in Figure 2 can microposition its tips by one LBN position along the X dimension. This will expand the equivalence class shown in the figure to include the two LBNs to the immediate left and right of the current LBN. The size of the equivalence class will increase by $3\times$. Micropositioning may not always be available as predicted by a simple model. If the media has expanded or contracted so far that the tip must already position itself far away from its central point, the micropositioning options will be reduced or altered. Lastly, micropositioning does not allow tips to access data in adjacent tips' squares because of inter-square spacing.

In summary, for each LBN, there exists an equivalence class of LBNs that can be potentially accessed in parallel with it. The members of the set are determined by the LBN's position, and the size of the set is determined by the number of read/write tips in the device and any micropositioning freedom. Further, only a subset (e.g., 5–10%) of the equivalence class can actually be accessed in parallel. The size of the subset is determined by the power budget of the device. If read/write tips share components, then there will be constraints on which LBNs from the set can be accessed together. These constraints are expressed by conflict relations. Lastly, an equivalence class can be expanded significantly (e.g., $11\times$) due to micropositioning capability.

3 Interfaces for storage access

Standard block-based storage interfaces provide commonality to storage devices and systems. They allow devices with potentially very different characteristics to be utilized transparently in

[2]Although it is not important to host software, the pictures showing tracks within contiguous rows of squares are just for visual simplicity. The tips over which any sector is striped would be spread widely across the device to distribute the resulting heat load and to create independence of tip failures. Likewise, the squares of sequentially numbered LBNs would be physically spread.

p	Level of parallelism		3
N	Number of squares		9
S_x	Sectors per square in X		3
S_y	Sectors per square in Y		3
M	Degree of micropositioning		0
N_x	Number of squares in X	p	3
N_y	Number of squares in Y	N/p	3
S_T	Sectors per track	$S_y \times N_x$	9
S_C	Sectors per cylinder	$S_T \times N_y$	27

Table 2: **Device parameters.** These are the parameters required to determine equivalence classes of LBNs that can be potentially accessed in parallel. The first five parameters are determined by the physical capabilities of the device and the last four are derived from them. The values in the rightmost column are for the simple device shown in Figures 2 and 3.

systems. It is natural to access a MEMStore using such an interface, since it would allow easy integration into existing systems. But, doing so would hide the interesting parallelism of a MEMStore. This section discusses the limitations of current interfaces in this respect, what additional information needs to be exported by a MEMStore to expose its parallelism, and how such information could be used in current block-based storage interfaces.

3.1 Traditional storage interfaces

The traditional model for accessing data through SCSI or ATA is by reading or writing ranges of blocks identified by LBN. These interfaces also have some provisions for exposing information from the storage device to the host. For example, SCSI disks export both standard and vendor-specific mode pages which contain some geometry and configuration information. These methods could be used to deliver information regarding parallelism to systems.

3.2 Exposing internal parallelism

This section describes equations and associated device parameters that a system can use to enumerate LBNs in a MEMStore that can be accessed in parallel.

The goal is that the system be able, for a given LBN, to determine the equivalence class of LBNs that are parallel-accessible. Determining this class for a MEMStore requires four parameters that describe the virtual geometry of the device and one which describes the degree of micropositioning. Table 2 lists them with example values taken from the device shown in Figures 2 and 3. The level of parallelism, p, is set by the power budget of the device, as described in Section 2.1. The total number of squares, N, is defined by the virtual geometry of the device. Since sequential LBNs are laid out over as many parallel tips as possible to optimize for sequential access, the number of squares in the X dimension, N_x, is equal to the level of parallelism, p. The number of squares in the Y dimension is the total number of squares, N, divided by p. The sectors per square in either direction, S_x and S_y, is determined by the bit density of each square. These parameters, along with N_x and N_y, determine the number of sectors per track, S_T, and the number of sectors per cylinder, S_C.

Without micropositioning, the size of an equivalence class is simply equal to the total number of squares, N, as there is an equivalent LBN in each square. The degree of micropositioning, M, is another device parameter which gives the number of cylinders in either direction over which an individual tip can microposition. M has the effect of making the equivalence class larger by a factor of $2M + 1$. So, if M in Figure 2 were 1, then the equivalence class for each LBN would have (at most) 27 LBNs in it. Micropositioning is opportunistic since, if the media has expanded, the micropositioning range will be used just to stay on track.

Given a single LBN l, a simple two-step algorithm yields all of the other LBNs in the equivalence class E_l. The first step maps l to an x, y position within its square. The second step iterates through each of the N squares and finds the LBNs in that square that are in the equivalence class.

The first step uses the following formulae:

$$\begin{aligned} x_l &= \lfloor l/S_C \rfloor \\ y_l &= \begin{cases} (\lfloor l/N_x \rfloor \;\%\; S_y) & \text{if } \lfloor l/S_T \rfloor \text{ even} \\ (S_y - 1) - (\lfloor l/N_x \rfloor \;\%\; S_y) & \text{otherwise} \end{cases} \end{aligned}$$

The formula for x_l is simply a function of l and the sectors per cylinder. The formula for y_l takes into account the track reversals described in Section 2.1 by reversing the y position in every other track.

The second step uses the following formula, $LBN_{l,i}$, which gives the LBN that is parallel to l in square i.

$$\begin{aligned} LBN_{l,i} = {} & (x_l \times S_C) \\ & +(i \;\%\; S_x) \\ & +(\lfloor i/N_x \rfloor \times S_T) \\ & + \begin{cases} (y_l \times N_x) & \text{if } \lfloor i\,/N_x \rfloor + x_l \text{ even} \\ (((S_y - 1) - y_l) \times N_x) & \text{otherwise} \end{cases} \end{aligned}$$

Like the formula for y_l, this formula takes track reversals into account.

The second step of the algorithm is to find the LBNs in each square that comprise the equivalence class E_l. Ignoring micropositioning, the equivalence class is found by evaluating $LBN_{l,i}$ for all N squares:

$$E_l = \{LBN_{l,0}, \ldots, LBN_{l,N-1}\}$$

If the MEMStore supports micropositioning, then the size of the equivalence class increases. Rather than using just x_l, $LBN_{l,i}$ is evaluated for all of the x positions in each square that are accessible by micropositioning; i.e., for all x's in the interval $[x_l - M, x_l + M]$.

Once a system knows the equivalence class, it can then, in the absence of shared components, choose any p sectors from that class and be guaranteed that they can be accessed in parallel. If there are shared components, then the conflict relations will have to be checked when choosing sectors from the class.

3.3 Expressing parallel requests

Since LBN numbering is tuned for sequential streaming, requests that can be serviced in parallel by the MEMStore may include disjoint ranges of LBNs. How these disjoint LBN ranges are expressed influences how these requests are scheduled at the MEMStore. That is, requests for disjoint sets of LBNs may be scheduled separately unless there is some mechanism to tell the storage device that they should be handled together.

One option is for the device to delay scheduling of requests for a fixed window of time, allowing concurrent scheduling of equivalent LBN accesses. In this scheme, a host would send all of the parallel requests as quickly as possible with ordinary READ and WRITE commands. This method requires additional request-tracking work for both the host and the device, and it will suffer some loss of performance if the host cannot deliver all of the requests within this time window (e.g., the delivery is interleaved by requests from another host).

Another option is for the host to explicitly group the parallel-accessible requests into a batch, informing the device of which media transfers the host expects to occur in parallel. With explicit information about parallel-accessible LBNs from the MEMStore, the host can properly construct batches of parallel requests. This second option can be easier for a host to work with and more efficient at the device.

3.4 Application interface

An application writer needs a simple API that enables the use of the equivalence class construct and the explicit batching mechanism. The following functions allow applications to be built that can exploit the parallelism of a MEMStore, as demonstrated in Section 5:

get_parallelism() returns the device parallelism parameter, p, described in Table 2.

batch() marks a batch of READ and WRITE commands that are to access the media in parallel.

get_equivalent(LBN ***)*** returns the LBN's equivalence class, E_{LBN}.

check_conflicting(LBN_1, LBN_2 ***)*** returns TRUE if there is a conflict between LBN_1 and LBN_2 such that they cannot be accessed in parallel (e.g., due to a shared component).

get_boundaries(LBN ***)*** returns LBN_{min} and LBN_{max} values, where $LBN_{min} \leq LBN \leq LBN_{max}$. This denotes the size of a request (in consecutive LBNs) that yields the most efficient device access. For MEMStore, $LBN_{max} - LBN_{min} = S_T$, which is the number of blocks on a single track containing LBN.

These functions are simple extensions to current storage interfaces, such as SCSI, and can be implemented with commands already defined in the SCSI protocol. The *get_parallelism()* function can be implemented by INQUIRY SCSI command. Alternatively, it can be included in a MEMStore-specific mode page which is fetched as a result of MODE SENSE SCSI command. The *batch()* corresponds to linking individual SCSI commands with the Link bit set. SCSI linking ensures that no other commands are executed in the middle of the submitted linked batch. A MEMStore would execute all linked commands as a single batch in parallel, and return the data to the host. The *get_boundaries()* function maps to the READ CAPACITY SCSI command with the PMI bit set. According to the specification, this command returns the last LBN before a substantial delay in data transfer. For a MEMStore, this delay corresponds to changing the direction of the sled motion at the end of a single track.

p	Level of parallelism	10
N	Number of squares	100
S_x	Sectors per square in X	2500
S_y	Sectors per square in Y	27
M	Degree of micropositioning	0
N_x	Number of squares in X	10
N_y	Number of squares in Y	10
S_T	Sectors per track	270
S_C	Sectors per cylinder	2700

Table 3: **Device parameters for the G2 MEMStore.** The parameters given here take into account the fact that individual 512 byte LBNs are striped across 64 read/write tips each.

The *get_boundaries()*, *get_equivalent()*, and *check_conflicting()* functions run in either the device driver or an application's storage manager, with the necessary parameters exposed through the SCSI mode pages.

4 Experimental setup

Our experiments rely on simulation because real MEMStores are not yet available. A detailed model of MEMS-based storage devices has been integrated into the DiskSim storage subsystem simulator [7]. For the purposes of this work, the MEMStore component was augmented to service requests in batches. As a batch is serviced by DiskSim, as much of its data access as possible is done in parallel given the geometry of the device and the level of parallelism it can provide. If all of the LBNs in the batch are parallel-accessible, then all of its media transfer will take place at once. Using the five basic device parameters and the algorithm described in Section 3.2, an application can generate parallel-accessible batches and effectively utilize the MEMStore's available parallelism.

For the experiments below, the five basic device parameters are set to represent a realistic MEMStore. The parameters are based on the G2 MEMStore from [26], and are shown in Table 3. The G2 MEMStore has 6400 probe tips, and therefore 6400 total squares. However, a single LBN is always striped over 64 probe tips so N for this device is $6400/64 = 100$. We have modified the G2 model to allow only 640 tips to be active in parallel rather than 1280 to better reflect the power constraints outlined in Section 2.2, making $p = 10$. Therefore, for a single LBN, there are 100 LBNs in an equivalence class, and out of that set any 10 LBNs can be accessed in parallel.

Each physical square in the G2 device contains a 2500×2500 array of bits. Each 512 byte LBN is striped over 64 read/write tips. After striping, the virtual geometry of the device works out to a 10×10 array of virtual squares, with sectors laid out vertically along the Y dimension. After servo and ECC overheads, 27 512-byte sectors fit along the Y dimension, making $S_y = 27$. Lastly, $S_x = 2500$, the number of bits along the X dimension. The total capacity for the G2 MEMStore is 3.46 GB. It has an average random seek time of 0.56 ms, and has a sustained bandwidth of 38 MB/s.

5 Exploiting internal parallelism

This section describes two interesting ways of exploiting the explicit knowledge of internal parallelism of a MEMStore. The example applications described here are representative of two distinct application classes. The free bandwidth example demonstrates how the MEMStore parallelism can be used for background system tasks that would normally interfere with primary workload. The second example shows how exposing MEMStore parallelism provides efficient accesses to two-dimensional data structures mapped to a linear LBN space.

5.1 Free bandwidth

As a workload runs on a MEMStore, some of the media bandwidth may be available for background accesses because the workload is not utilizing the full parallelism of the device. Every time the media sled is positioned, a full equivalence class of LBNs is available out of which up to p sectors may be accessed. Some of those p sectors will be used by the foreground workload, but the rest can be used for other tasks. Given an interface that exposes the equivalence class, the system can choose which LBNs to access "for free." This is similar to freeblock scheduling for disk drives [18], but does not require low-level service time predictions; the system can simply pick available LBNs from the equivalence class as it services foreground requests.

We ran DiskSim with a foreground workload of random 4 KB requests, and batched those requests with background transfers for other LBNs in the equivalence class. The goal of the background workload is to scan the entire device, until every LBN has been read at least once, either by the foreground or background workload. Requests that are scheduled in the background are only those for LBNs that have not yet been touched, while the foreground workload is random. Scanning large fractions of a device is typical for backup, decision-support, or data integrity checking operations. As some MEMStore designs may utilize recording media that must be periodically refreshed, this refresh background task could be done with free bandwidth.

In the default G2 MEMStore model, $p = 10$, meaning that 10 LBNs can be accessed in parallel. The 4 KB foreground accesses will take 8 of these LBNs. Foreground requests, however, are not always aligned on 10 LBN boundaries, since they are random. In these cases, the media transfer will take two (sequential) accesses, each of 10 LBNs. In the first case, 80% of the media bandwidth is used for data transfer, and in the second case, only 40% is used. By using the residual 2 and 12 LBNs, respectively, for background transfers, we are able to increase media bandwidth utilization to 100%.

Figure 4 shows the result of running the foreground workload until each LBN on the device has been touched either by the foreground workload or for free. As time progresses, more and more of the device has been read, with the curve tapering off as the set of untouched blocks shrinks. By the 1120^{th} minute, 95% of the device has been scanned. The tail of the curve is very long, with the last block of the device not accessed until the 3375^{th} minute. For the first 95% of the LBN space, an average of 6.3 LBNs are provided to the scan application for free with each 4 KB request.

To see the effect of allowing more parallel access, we increased p in the G2 MEMStore to be 20. In this case, more free bandwidth is available and the device is fully scanned more quickly. The first 95% of the device is scanned in 781 minutes, with the last block being accessed at 2290 minutes. For the first 95% of the LBN space, an average of 11 LBNs are provided to the scan application for free.

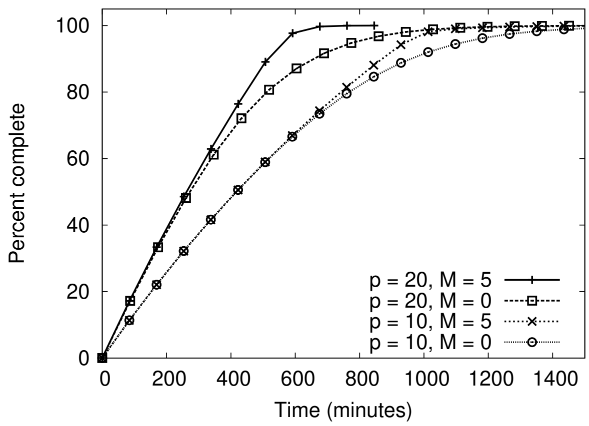

Figure 4: **Reading the entire device for free.** In this experiment, a random workload of 4KB requests is run in the foreground, with a background task that scans the entire device for free. The graph shows the percentage of the G2 MEMStore scanned as a function of time. For $p = 10, M = 0$, the scan is 95% complete at 1120 minutes and finishes at 3375 minutes. For $p = 20, M = 0$, the scan is 95% complete at 781 minutes and finishes at 2290 minutes. Allowing 5 tracks of micropositioning allows more options for the background task. At $p = 10, M = 5$, the scan is 95% complete at 940 minutes and completes at 1742 minutes. At $p = 20, M = 5$, the scan is 95% complete at 556 minutes and completes at 878 minutes.

p	M	Time to scan 95%	Time to scan 100%
20	5	556 minutes	878 minutes
20	0	781 minutes	2290 minutes
10	5	940 minutes	1742 minutes
10	0	1120 minutes	3375 minutes

Table 4: **Reading the entire device for free.** The time to read the entire device is dominated by the last few percent of the LBNs. Greater p allows the device to transfer more LBNs in parallel, and increases the set of LBNs that the background task can choose from while gathering free blocks. Increasing M increases the size of the equivalence class and, thus, the number of free blocks for the background task to choose from.

Micropositioning significantly expands the size of equivalence classes. This gives the background task many more options from which to choose, reducing the total runtime of the background scan. To quantify this, we set $M = 5$, expanding the size of the equivalence classes from 100 LBNs to 1100 LBNs. In both the $p = 10$ case and the $p = 20$ case, the device is scanned significantly faster. With $p = 10$ and $M = 5$, the device scan time is reduced to 1742 minutes; with $p = 20$ and $M = 5$, it is reduced to 878 minutes.

5.2 Efficient 2D table access

By virtue of mapping two-dimensional structures (e.g., large non-sparse matrices or database tables) into a linear LBN space, efficient accesses are only possible in either row-major or column-major order. Hence, a data layout that optimizes for the most common access method is chosen with the understanding that accesses along the other major axis are inefficient. To make accesses in both dimensions efficient, one can create two copies of the same data; one copy is then optimized for row order access and the other for column order access [23]. Unfortunately, not only does this double the required space, but updates must propagate to both replicas to ensure data integrity.

This section describes how to efficiently access data in both row- and column-major orders. It illustrates the advantages of using MEMStore and its storage interface for database table scans that access only a subset of columns.

5.2.1 Relational database tables

Relational database systems (RDBS) use a scan operator to sequentially access data in a table. This operator scans the table and returns the desired records for a subset of attributes (table fields). Internally, the scan operator issues page-sized I/Os to the storage device, stores the pages in its buffers, and reads the data from buffered pages. A single page (typically 8 KB) contains a fixed number of complete records and some page metadata overhead.

The page layout prevalent in commercial RDBS stores a fixed number of records for all n attributes in a single page. Thus, when scanning a table to fetch records of only one attribute (i.e., column-major access), the scan operator still fetches pages with data for *all* attributes, effectively reading the entire table even though only a subset of the data is needed. To alleviate the inefficiency of a column-major access in this data layout, an alternative page layout vertically partitions data to pages with a fixed number of records of a *single* attribute [5]. However, record updates or appends require writes to n different locations, making such row-order access inefficient. Similarly, fetching full records requires n single-attribute table accesses and $n-1$ joins to reconstruct the entire record.

With proper allocation of data to a MEMStore LBN space, one or more attributes of a single record can be accessed in parallel. Given a degree of parallelism, p, accessing a single attribute yields higher bandwidth, by accessing more data in parallel. When accessing a subset of $k+1$ attributes, the desired records can exploit the internal MEMStore parallelism to fetch records in lock-step, eliminating the need for fetching the entire table.

5.2.2 Data layout for MEMStore

To exploit parallel data accesses in both row- and column-major orders, we define a *capsule* as the basic data allocation and access unit. A single capsule contains a fixed number of records for all table attributes. As all capsules have the same size, accessing a single capsule will always fetch the same number of complete records. A single capsule is laid out such that reading the whole record (i.e., row order access) results in parallel access to all of its LBNs. The capsule's individual LBNs are assigned such that they belong to the same equivalence class, offering parallel access to any number of attributes within.

Adjacent capsules are laid next to each other such that records of the same attribute in two adjacent capsules are mapped to sequential LBNs. Such layout ensures that reading sequentially across capsules results in repositioning only at the end of each track or cylinder. Furthermore, this layout ensures that sequential streaming of one attribute is realized at the MEMStore's full bandwidth by engaging all tips in parallel. Specifically, this sequential walk through the LBN space can be realized by multiple tips reading up to p sequential LBNs in parallel, resulting in a column-major access at full media bandwidth.

A simple example that lays records within a capsule and maps contiguous capsules into the LBN space is illustrated in Figure 5. It depicts a capsule layout with 12 records consisting of two attributes a_1 and a_2, which are 1 and 2 units in size, respectively. It also illustrates how adjacent capsules are mapped into the LBN space of the three-by-three MEMStore example from Figure 2.

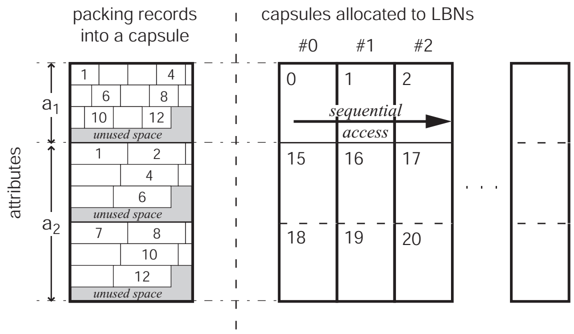

Figure 5: **Data allocation with capsules.** The capsule on the left shows packing of 12 records for attributes a_1 and a_2 into a single capsule. The numbers within denote record number. The 12-record capsules are mapped such that each attribute can be accessed in parallel and data from a single attribute can be accessed sequentially, as shown on the right. The numbers in the top left corner are the LBNs of each block comprising the capsule.

Finding the (possibly non-contiguous) LBNs to which a single capsule should be mapped, as well as the location for the logically next LBN, is done by calling the *get_equivalent()* and *get_boundaries()* functions. In practice, once a capsule has been assigned to an LBN and this mapping is recorded, the locations of the other attributes can be computed from the values returned by the interface functions.

5.2.3 Allocation

The following describes the implementation details of the capsule layout described in the previous section. This description serves as a condensed example of how the interface functions can be used in building similar applications.

Data allocation is implemented by two routines that call the functions of the MEMStore interface. These functions do not perform the calculations described in this section. They simply lookup data returned by the *get_equivalent()* and *get_boundaries()* functions. The `CapsuleResolve()` routine determines an appropriate capsule size using attribute sizes. The degree of parallelism, p, determines the offsets of individual attributes within the capsule. A second routine, called `CapsuleAlloc()`, assigns a newly allocated capsule to free LBNs and returns new LBNs for the this capsule. The LBNs of all attributes within a capsule can be found according to the pattern determined by the `CapsuleResolve()` routine.

The `CapsuleAlloc()` routine takes an LBN of the most-recently allocated capsule, l_{last}, finds enough unallocated LBNs in its equivalence class E_{last}, and assigns the new capsule to l_{new}. By definition, the LBN locations of the capsule's attributes belong to E_{new}. If there are enough unallocated LBNs in E_{last}, $E_{last} = E_{new}$. If no free LBNs in E_{last} exist, E_{new} is different from E_{last}. If there are some free LBNs in E_{last}, some attributes may spill into the next equivalence class. However, this capsule can still be accessed sequentially.

Allowing a single capsule to have LBNs in two different equivalence classes does not waste any space. However, accessing all attributes of these split capsules is accomplished by two separate parallel accesses, the latter being physically sequential to the former. Given capsule size in LBNs, c, there is one split capsule for every $|E| \,\%\, cp$ capsules. If one wants to ensure that *every* capsule is always accessible in a single parallel operation, one can waste $1/\left(|E| \,\%\, cp\right)$ of device capacity. These unallocated LBNs can contain tables with smaller capsule sizes, indexes or database logs.

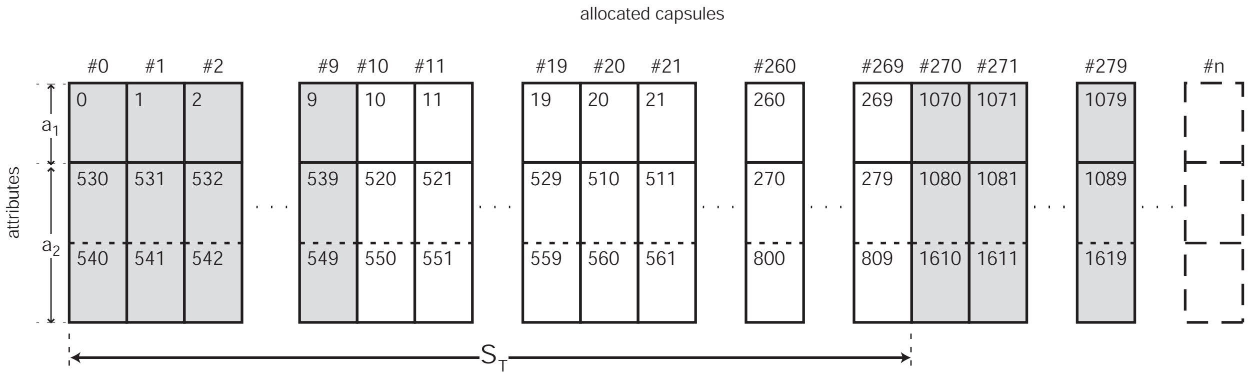

Figure 6: **Capsule allocation for the G2 MEMStore.** This picture shows capsules with two attributes a_1 and a_2 whose sizes are 8 and 16 bytes, respectively. Given an LBN size of 512 bytes, and a level of parallelism, $p = 10$, a single capsule contains 64 records and maps to three LBNs. Note that each row for capsules 0 through 269 contains contiguous LBNs of a single track: a_1 spans track 0-269, and a_2 spans two tracks with LBN ranges 270-539 and 540-809. The shaded capsules belong to the same equivalence class. Thanks to the *get_equivalent()* and *get_boundaries()* functions, a database system does not have to keep track of all these complicated patterns. Instead, it only keeps the capsule's starting LBN. From this LBN, all other values are found by the MEMStore interface function calls.

Because of the MEMStore layout, l_{new} is not always equal to $l_{last} + 1$. This discontinuity occurs at the end of each track.[3] Calling *get_boundaries()* determines if l_{last} is the last LBN of the current track. If so, the `CapsuleAlloc()` simply offsets into E_{last} to find the proper l_{new}. The offset is a multiple of p and the number of blocks a capsule occupies. If l_{last} is not at the end of the track, then $l_{new} = l_{last} + 1$.

Figure 6 illustrates the allocation of capsules with two attributes a_1 and a_2 of size 1 and 2 units, respectively, to the LBN space of a G2 MEMStore using the sequential-optimized layout. The depicted capsule stores a_1 at LBN capsule offset 0, and the two blocks of a_2 at LBN offsets p and $2p$. These values are offset relative to the capsule's LBN position within E_{LBN}.

5.2.4 Access

For each capsule, the RDBS maintains its starting LBN from which it can determine the LBNs of all attributes in the capsule. This is accomplished by calling the *get_equivalent()* function. Because of the allocation algorithm, the capsules are laid out such that sequential scanning through records of the attribute a_1 results in sequential access in LBN space as depicted in Figure 6. This sequential access in LBN space is realized by p batched reads executing in parallel. When accessing both a_1 and a_2, up to p/c capsules can be accessed in parallel where capsule size $c =$ *size*$(a_1 + a_2)$.

Streaming a large number of capsules can be also accomplished by pipelining reads of S_T sequential LBNs of attribute a_1 followed by $2S_T$ sequential LBNs of a_2. Setting a scatter-gather list for these sequential I/Os ensures that data are put into proper places in the buffer pool. The residual capsules that span the last segment smaller than S_T are then read in parallel using batched I/Os.

[3]This discontinuity also occurs at the boundaries of equivalence classes, or every p capsules, when mapping capsules to LBNs on even tracks of a MEMStore with the sequential-optimized layout depicted in Figure 2 The LBNs of one attribute, however, always span only one track.

Operation	Data Layout	
	normal	*capsule*
entire table scan	22.44 s	22.93 s
a_1 scan	22.44 s	2.43 s
a_1 + a_2 scan	22.44 s	12.72 s
100 records of a_1	1.58 ms	1.31 ms

Table 5: **Database access results.** The table shows the runtime of the specific operation on the 10,000,000 record table with 4 attributes for the *normal* and *capsule*. The rows labeled a_1 scan and $a_1 + a_2$ represent the scan through all records when specific attributes are desired. the last row shows the time to access the data for attribute a_1 from 100 records.

5.2.5 Implementation details

The parallel scan operator is implemented as a standalone C++ application. It includes the allocation and layout routines described in Section 5.2.3 and allows an arbitrary range of records to be scanned for any subset of attributes. The allocation routines and the scan operator use the interface functions described in Section 3.3. These functions are exported by linked-in stub, which communicates via a socket to another process. This process, called *devman*, emulates the functionality of a MEMStore device manager running firmware code. It accepts I/O requests on its socket, and runs the I/O through the DiskSim simulator configured with the G2 MEMStore parameters. The *devman* process synchronizes DiskSim's simulated time with the wall clock time and uses main memory for data storage.

5.2.6 Results

To quantify the advantages of our parallel scan operator, this section compares the times required for different table accesses. It contrasts their respective performance under three different layouts on a single G2 MEMStore device. The first layout, called *normal*, is the traditional row-major access optimized page layout. The second layout, called *vertical*, corresponds to the vertically partitioned layout optimized for column-major access. The third layout, called *capsule*, uses the layout and access described in Section 5.2.3. We compare in detail the *normal* and *capsule* cases.

Our sample database table consists of 4 attributes a_1, a_2, a_3, and a_4 sized at 8, 32, 15, and 16 bytes respectively. The *normal* layout consists of 8 KB pages that include 115 records. The *vertical* layout packs each attribute into a separate table. For the given table header, the *capsule* layout produces capsules consisting of 9 pages (each 512 bytes) with a total of 60 records. The table size is 10,000,000 records with a total of 694 MB of data.

Table 5 summarizes the table scan results for the *normal* and *capsule* cases. Scanning the entire table takes respectively 22.44 s and 22.93 s for the *normal* and *capsule* cases and the corresponding user-data bandwidth is 30.9 MB/s and 30.3 MB/s. The run time difference is due to the amount of actual data being transfered. Since the *normal* layout can pack data more tightly into its 8 KB page, it transfers a total of 714 MB at a rate of 31.8 MB/s from the MEMStore. The *capsule* layout creates, in effect, 512-byte pages which waste more space due to internal fragmentation. This results in a transfer of 768 MB. Regardless, it achieves a sustained bandwidth of 34.2 MB/s, or 7% higher than *normal*. While both methods access all 10 LBNs in parallel most of the time, the data access in the *capsule* case is more efficient due to smaller repositioning overhead at the end of a

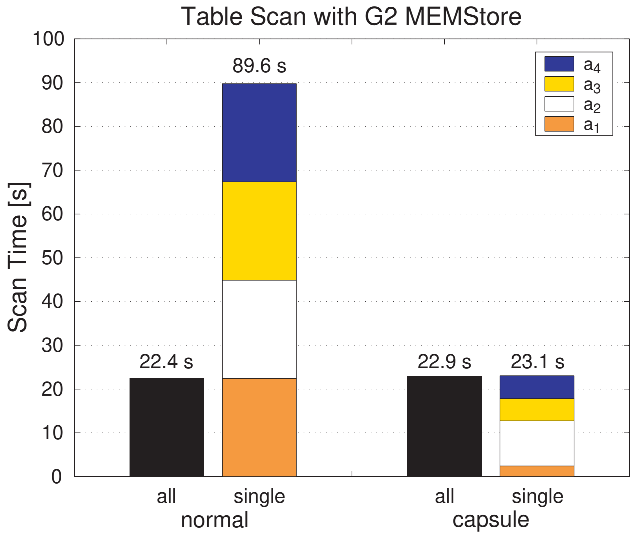

Figure 7: **Table scan with different number of attributes.** This graph shows the runtime of scanning 10,000,000 records using G2 MEMStore. For each of the two layouts the left bar, labeled all, shows the runtime of the entire table with 4 attributes. The right bar, labeled single, is composed of four separate scans of each successive attribute, simulating the situation where multiple queries access different attributes. Since the *capsule* layout takes adavantage of MEMStore's parallelism, each attribute scan runtime is proportional to the amount of data occupied by that attribute. The *normal*, on the other hand, must read the entire table to fetch one of the desired attributes.

cylinder.

As expected, *capsule* is highly efficient when only a subset of the attributes are required. A table scan of a_1 or $a_1 + a_2$ in the *normal* case always takes 22.44 s, since entire pages including the undesired attributes must be scanned. The *capsule* case only requires a fraction of the time corresponding to the amount of data due to each desired attribute. Figure 7 compares the runs of a full table scan for all attributes against four scans of individual attributes. The total runtime of four attribute scans in the *capsule* case takes the same amount of time as the full table scan. In contrast, the four successive scans take four times as long as the full table scan with the *normal* layout.

Most importantly, a scan of a single attribute a_1 in the *capsule* case takes only one ninth (2.43 s vs. 22.93 s) of the full table scan since all ten parallel accesses read records of a_1. On the other, scanning the full table in the *normal* case requires a transfer of 9 times as much data and uses the parallelism p to access.

Short scans of 100 records (e.g., in queries with high selectivity) are 20% faster for *capsule* since they fully utilize the MEMStore's internal parallelism. Furthermore, the latency to the first record is shorter due to smaller access units, compared to *normal*. Compared to *vertical*, the access latency is also shorter due to the elimination of the join operation. In our example, the vertically partitioned layout must perform two joins before being able to fetch an entire record. This join, however, is not necessary in the *capsule* case, as it accesses records in lock-step, implicitly utilizing the available MEMStore internal parallelism.

The *vertical* case exhibits similar results for individual attribute scans as the *capsule* case. In contrast, scanning the entire table requires additional joins on the attributes. The cost of this join depends on the implementation of the join algorithm which is not the focus of this paper.

Comparing the latency of accessing one complete random record under the three different scenarios shows an interesting behavior. The *capsule* case gives an average access time of 1.385 ms, the *normal* case 1.469 ms, and the *vertical* case 4.0 ms. The difference is due to different access patterns. The *capsule* access includes a random seek to the capsule's location followed by 9 batched accesses to one equivalence class proceeding in parallel. The *normal* access involves a random seek followed by a sequential access to 16 LBNs. Finally, the *vertical* access requires 9 accesses each consisting of a random seek and one LBN access.

5.2.7 Effects of micropositioning

As demonstrated in the previous section, scanning a_1 in a data layout with capsules spanning 10 LBNs will be accomplished in one tenth of the time it would take to scan the entire table. While using micropositioning does not reduce this time to one-hundredth (it is still governed by p), for specific accesses, it can provide 10 times more choices (or more precisely Mp) choices, resulting in up to 100-times benefit to applications.

6 Related work

MEMS-based storage is a recent concept, independently conceived by several groups in the mid-nineties [3, 30]. Most designs conform to the description given in Section 2. Descriptions and simulation models suitable for computer systems research have been developed [11], capturing the anticipated performance characteristics. Subsequent research has developed analytic solutions to these performance models [8, 14] and models of less aggressive seek control mechanisms [19]. These models have been used for trade-off analyses of MEMStore design parameters [8, 27] and architectural studies of MEMStore uses in the storage hierarchy [26] and in disk arrays [29].

MEMStore simulation models have also allowed several explorations of how software systems would interact with such devices. Most relevant here is the work of Griffin et al. [12], which shows that traditional disk-based OS algorithms are sufficient for managing MEMStore performance, assuming that a SCSI-like interface is used. This paper provides a case for a less restrictive interface. Other studies have looked at power management. MEMStore characteristics make power management straightforward: Schlosser et al. [26] describe simple policies, and Lin et al. [17] evaluate them in greater detail.

Of course, many research efforts have promoted exploitation of low-level device characteristics to improve performance. Ganger [10] and Denehy et al. [6] promote an approach to doing this, which our work shares, wherein host software is informed of some device characteristics. The most relevant specific example proposes exposing the individual device boundaries in disk array systems to allow selective software-managed redundancy [6]. Other examples include exposing disk track boundaries [25] and exposing relative request latency expectations [21].

The decomposition storage model [5] provides efficient table scans for a (small) a subset of columns by vertically partitioning data into separate tables containing one of n attributes. While it can efficiently stream individual attributes, it is not used in commercial databases, because of the high cost for one record updates or reads, which require accesses to n locations. To alleviate the high cost of accessing a complete record, Ramamurthy et al. [23] suggest a table layout with two replicas. One table replica is vertically partitioned, while the other uses a traditional page layout optimized for row-major access. This solution requires twice the capacity and still suffers high update costs in the vertically partitioned layout, which must be kept consistent with the other replica. The capsule data layout for MEMStores allows efficient record reads or updates in a single parallel access as well as efficient scans through arbitrary subsets of table attributes.

Allowing the storage system to schedule overall task sets has been shown to provide more robust performance. Disk-directed I/O [16] consists of applications exposing substantial access patterns and allowing the storage system to provide data in an order convenient to it. Generally, cursors (i.e., iterator abstractions) in database systems [22] allow similar storage system flexibility. For example, the River system [2] exploits this flexibility to provide robust stream-based access to large data sets. For individual accesses, dynamic set interfaces [28] can provide similar flexibility. The approaches described here may fit under such interfaces.

Active disk systems [1, 15, 24] enhance storage systems with the ability to perform some application-level functions. Data filtering is one of the more valuable uses of active disk capabilities, and the "project" database function is a column-based access pattern. Thus, such a use of active disk interfaces would allow the device to internally handle the request batching portion of the parallelism interface; applications would still have to lay data out appropriately.

7 Summary

This paper describes an approach to utilizing the unique parallelism available in MEMStores. With a traditional storage interface augmented by two simple constructs that hide the complexity of LBN mappings, application writers can exploit a MEMStore's parallelism features. For example, efficient selective table scans and background scans are shown to achieve substantial benefit.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998), pages 81–91. ACM, 1998.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: making the fast case common. *Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[3] L. Richard Carley, James A. Bain, Gary K. Fedder, David W. Greve, David F. Guillou, Michael S. C. Lu, Tamal Mukherjee, Suresh Santhanam, Leon Abelmann, and Seungook Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.

[4] Center for Highly Integrated Information Processing and Storage Systems, Carnegie Mellon University. http://-www.ece.cmu.edu/research/chips/.

[5] George P. Copeland and Setrag Khoshafian. A decomposition storage model. *ACM SIGMOD International Conference on Management of Data* (Austin, TX, 28–31 May 1985), pages 268–279. ACM Press, 1985.

[6] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. *Summer USENIX Technical Conference* (Monterey, CA, 10–15 June 2002), 2002.

[7] The DiskSim Simulation Environment (Version 3.0). http://www.pdl.cmu.edu/DiskSim/index.html.

[8] Ivan Dramaliev and Tara Madhyastha. Optimizing probe-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003). USENIX Association, 2003.

[9] G. K. Fedder, S. Santhanam, M. L. Reed, S. C. Eagle, D. F. Guillou, M. S.-C. Lu, and L. R. Carley. Laminated high-aspect-ratio microstructures in a conventional CMOS process. *IEEE Micro Electro Mechanical Systems Workshop* (San Diego, CA), pages 13–18, 11–15 February 1996.

[10] Gregory R. Ganger. *Blurring the line between OSs and storage devices*. Technical report CMU–CS–01–166. Carnegie Mellon University, December 2001.

[11] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. Modeling and performance of MEMS-based storage devices. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 17–21 June 2000). Published as *Performance Evaluation Review*, **28**(1):56–65, 2000.

[12] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. Operating system management of MEMS-based storage devices. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 227–242. USENIX Association, 2000.

[13] Hewlett-Packard Laboratories Atomic Resolution Storage. http://www.hpl.hp.com/research/storage.html.

[14] Bo Hong and Scott A. Brandt. *An analytical solution to a MEMS seek time model*. Technical Report UCSC–CRL–02–31. September 2002.

[15] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, **27**(3):42–52, September 1998.

[16] David Kotz. Disk-directed I/O for MIMD multiprocessors. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 61–74. USENIX Association, 14–17 November 1994.

[17] Ying Lin, Scott Brandt, Darrell Long, and Ethan Miller. Power conservation strategies for MEMS-based storage devices. *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (Fort Worth, TX, October 2002), 2002.

[18] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[19] Tara M. Madhyastha and Katherine Pu Yang. Physical modeling of probe-based storage. *IEEE Symposium on Mass Storage Systems* (April 2001). IEEE, 2001.

[20] Nadim Maluf. *An introduction to microelectromechanical systems engineering*. Artech House, 2000.

[21] Rodney Van Meter and Minxi Gao. Latency management in storage systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 103–117. USENIX Association, 2000.

[22] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2000.

[23] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. *International Conference on Very Large Databases* (Hong Kong, China, 20–23 August 2002), pages 430–441. Morgan Kaufmann Publishers, Inc., 2002.

[24] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. *International Conference on Very Large Databases* (New York, NY, 24–27 August, 1998). Published as *Proceedings VLDB*, pages 62–73. Morgan Kaufmann Publishers Inc., 1998.

[25] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.

[26] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):1–12, 2000.

[27] Miriam Sivan-Zimet and Tara M. Madhyastha. Workload based optimization of probe-based storage. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Marina Del Rey, CA, 2002). Published as *ACM SIGMETRICS Performance Evaluation Review*, **30**(1):256–257. ACM Press, 2002.

[28] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggreggate file I/O latency. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):252–263. ACM, 1997.

[29] Mustafa Uysal, Arif Merchant, and Guillermo A. Alvarez. Using MEMS-based storage in disk arrays. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003). USENIX Association, 2003.

[30] P. Vettiger, G. Cross, M. Despont, U. Drechsler, U. Dürig, B. Gotsmann, W. Häberle, M. A. Lantz, H. E. Rothuizen, R. Stutz, and G. K. Binnig. The ”Millipede” – nanotechnology entering data storage. *IEEE Transactions on Nanotechnology*, **1**(1):39–55. IEEE, March 2002.

[31] P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The “Millipede” – more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, **44**(3):323–340, 2000.

[32] Peter Vettiger and Gerd Binnig. The Nanodrive project. *Scientific American*, **228**(1):46–53. Scientific American, Inc., January 2003.

[33] Kensall D. Wise. Special issue on integrated sensors, microactuators, and microsystems (MEMS). *Proceedings of the IEEE*, **86**(8):1531–1787, August 1998.