Tabular Placement of Relational Data on MEMS-based Storage Devices

Hailing Yu Divyakant Agrawal Amr El Abbadi

University of California at Santa Barbara
Computer Science Department
Santa Barbara, 93106
USA
{hailing,agrawal,amr}@cs.ucsb.edu

Abstract

Due to the advances in semiconductor manufacturing, the gap between main memory and secondary storage is constantly increasing. This becomes a significant performance bottleneck for Database Management Systems, which rely on secondary storage heavily to store large datasets. Recent advances in nanotechnology have led to invention of alternative means for persistent storage. In particular, MicroElectroMechanical Systems (MEMS) based storage technology has emerged as the leading candidate for next generation storage systems. In order to integrate MEMS-based storage into conventional computing platform, new techniques are needed for I/O scheduling and data placement.

In the context of relational data, it has been observed that access to relations be enabled both in row-wise manner as well as in column-wise fashion. In this paper, we exploit the physical characteristics of MEMS-based storage devices to develop a data placement scheme for relational data that enables retrieval in both row-wise and column-wise manner. We demonstrate that this data layout not only improves I/O utilization, but results in better cache performance.

1 Introduction

Advances in Nanotechnology hold significant promise to overcome many of the physical limitations of traditional magnetic hard disks and transistor-based memory chips. MicroElectroMechanical Systems (MEMS) are currently being developed by various companies including IBM, HP, Phillips, Seagate Technology and others [13]. MEMS are devices that have microscopic moving parts made using techniques similar to that used in semiconductor manufacturing. The IBM millipede project [13] promises to deliver by year 2005 a postage-stamp size memory card that can hold several gigabytes of fast non-volatile memory. And this project, according to the MEMS designers, only scratches the surface, as the digital bits of future generation MEMS-based storage devices will continue to shrink until they are individual molecules or even atoms [13]. As material scientists and mechanical engineers work feverishly to develop more efficient MEMS-based storage devices, the role of the computer science research is to effectively integrate such devices into computer systems for different applications. The challenges are big and intriguing, but some initial steps have been taken by various research groups, notably the CMU CHIPS project, which has explored various operating systems issues such as request scheduling, data placement, and others [11].

In this paper, we address the problem of integrating MEMS storage devices from a database point of view. Given the significant potential MEMS has for large and fast non-volatile storage of data, databases can benefit tremendously from such devices. Databases have always suffered from the increasing gap between economically viable non-volatile hard disks and expensive but fast transistor-based in-memory chips. MEMS-based storage devices hold the promise of providing a feasible compromise. More recently, it has been established that the data layout on the secondary storage in DBMSs plays a crucial role with respect to cache performance [4]. Although many issues need to be resolved before MEMS are completely integrated into computer systems in general and database applications in particular, we address, in this paper, a first step in that direction, namely data placement, which is crucial for the database problem.

The rest of this paper is organized as follows. Section 2 presents the MEMS-based storage structure. In Section 3, we review the existing data placement techniques designed for conventional hard disks. In Section 4, we describe our Flexible Retrieval Model (FRM) for MEMS-based storage devices and analyze the proposed approach. Section 5 demonstrates the effect of FRM scheme on cache and I/O performance. In Section 6, we conclude with a discussion of our results.

2 MEMS-based Storage Architecture

MEMS are extremely small mechanical structures formed by the integration of mechanical elements, actuators, electronics, and sensors. These are fabricated on silicon chips using photolithographic processes similar to those employed in manufacturing standard semiconductor devices. As a result, MEMS-based storage can be manufactured at a very low cost. They represent a compromise between slow traditional disks and expensive storage based on EEPROM technologies. Unlike traditional disks, MEMS-based storage devices [5] do not make use of rotating platters due to the difficulty in manufacturing efficient and reliable rotating parts in silicon. The emerging paradigm for such systems is that of a large-scale MEMS array which, like disk drives, has read/write heads and a recording media surface. The read/write heads are probe tips mounted on micro-cantilevers embedded in a semiconductor wafer and arranged in a rectangular fashion. The recording media is another rectangular silicon wafer (called the media sled) that can use conventional techniques for recording data. This structure gives MEMS-based storage two-dimensional characteristics.

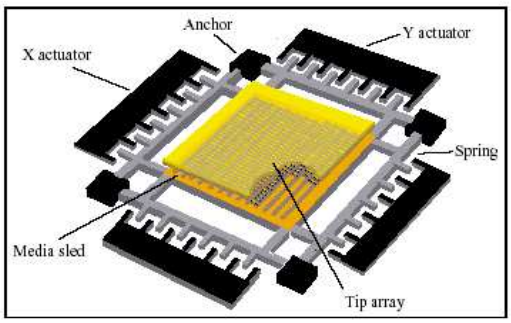


Figure 1: The architecture of CMU CHIPS

MEMS-based storage devices are composed of tens to thousands of recording heads and a recording media surface. The recording heads (probe tips) form a two-dimensional array and are fabricated on silicon chips. The recording media, referred to as the media sled, is spring-mounted above the probe tips array and can move in the X and Y dimensions. There are several different approaches for recording data. For example, IBM's Millipede uses pits in the polymers made by tip heating [9], while CMU CHIPS adopts the same technique as data recording on magnetic surface [3]. We now give an overview of the CMU CHIPS project which is dedicated to next-generation MEMS-based storage devices. To access data under this model, the media sled is moved from its current position to a specific position defined by (x, y) coordinates. After the "seek" is performed, the media sled moves in the Y direction while the probe tips access the data. The design is shown in Figure 1 (based on the CMU design [11]). The media sled also moves in the Z direction to actuate the distance between the probe tips and the media sled. The X and Y actuators provide the force for moving the media sled in the X and Y directions while the spring provides the restoring motion. These two actuators work independently.

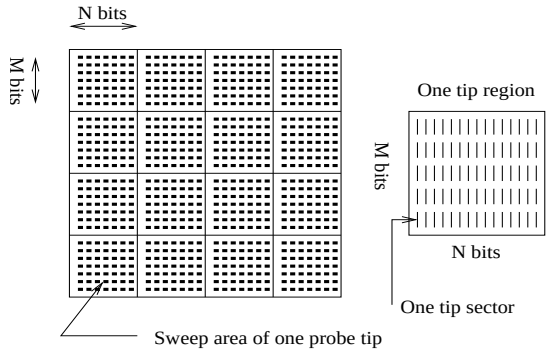


Figure 2: The media sled id divided into rectangular regions

The media sled is organized into rectangular regions at the lower level. Each of these rectangular regions contains $M \times N$ bits and is accessible by one tip. Each region corresponds to a single probe tip, which can access data in that region, as shown in Figure 2. Bits within a region are grouped into vertical 90-bit columns called *tip sectors*; each tip sector contains 10 bits of sled positioning information and 80 encoded data bits providing 8 data bytes. Thus an 8-byte tip sector is the smallest accessible unit of data in MEMS-based storage [5] and it is determined by the (x, y) coordinates of the media sled and the corresponding tip. The parameters of the devices are given in Table 1. Because of the heat-dissipation constraint, in the CMU CHIPS model, not all tips can be activated simultaneously even though it is theoretically feasible. The maximum number of tips that can be activated concurrently is limited to 1280. Each rectangular region stores 2000×2000 bits. In this paper, we design our data placement strategy based on this model.

Table 1: Parameters of MEMS-based storage devices from CMU CHIPS

number of tips	6400
simultaneously active tips	1280
tip sector size	8 bytes
servo overhead	10 bits per tip sector
bits per tip region ($M \times N$)	2000×2000
X axis settle time	0.125ms
Average turnaround time	0.06ms

Because the MEMS-based storage devices have very different characteristics from disk devices, from the operating system point of view, the algorithms for I/O scheduling, data layout, and failure management techniques designed for traditional disks need to be revisited. Prior approaches, which are for integration of MEMS-based storage devices into computing systems, have focused on mapping MEMS-based storage devices into disk-like devices [8]. The results

showed that stand-alone MEMS-based storage devices improve the overall application run time by a factor of 1.9 to 4.4 [11]. However, as a result of this mapping, MEMS-based storage devices lose the two-dimensional property, thus further performance gains from this property are lost, especially for data-intensive applications, such as DBMSs. To integrate MEMS-based storage into DBMSs, data placement schemes, such as relational data placement and index structure placement, need to be re-examined. In this paper, we focus on relational data placement and show that by treating MEMS-based storage as a two-dimensional storage for placing relational data, the gap between the main memory and secondary storage can be reduced notably. The idea is based on the observation that most queries in relational DBMSs need only a subset of attributes; and hence I/O time is wasted retrieving unnecessary attributes by mapping MEMS-based storage to one-dimensional devices. In a recent presentation at the first Biennial Conference on Innovative Data Systems Research(CIDR'03), Michael Stonebraker [12] made a case for multiple data organization for relational data so that it is compatible with both OLTP and OLAP workloads. In particular, he observed that due to the update characteristics of OLTP transactions, relations need to be accessed in a row-wise manner. In contrast, since in OLAP queries, only a subset of attributes are of interest, the I/O system should facilitate data retrieval on a column-wise manner. Therefore, we propose a data placement scheme (Flexible Retrieval Model) for relational DBMSs. In this new scheme, data in relations are modeled as two-dimensional when they are stored on MEMS-based storage devices. This facilitates the retrieval of only the relevant subsets of the relations.

By mapping MEMS-based storage into disk-like structure, the existing data placement techniques, such as N-ary Storage Model (NSM) [10], Decomposition Storage Model (DSM) [6], and Partition Attributes Across (PAX) [4], can be easily adapted to MEMS-based storage. NSM stores data record-by-record in each disk page. For any query, disk pages are read into memory as units. However, most queries use only a subset of attributes, and hence, NSM has poor I/O utilization. In order to decrease the I/O overhead, DSM was proposed. DSM partitions a relation into sub-relations, each attribute corresponding to a sub-relation. A query loads only those sub-relations, which are needed to process the query, into memory. However DSM does not perform well for queries that involve multiple attributes from a relation because of the expensive join operation that must be performed to combine the decomposed attributes of a relation. PAX was proposed recently to improve the cache utilization by reorganizing records in one disk page. Each page is divided into mini-pages, and each mini-page stores values of one attribute in the page. Experiments show that PAX performs better than NSM under most conditions, however, PAX [4] does not reduce the I/O cost when compared to NSM. In fact, no method has been proposed to combine these factors of I/O time and cache hit ratios into one data placement policy for disk-like storage devices. Our data placement strategy for placing relations on MEMS-based storage mainly considers reducing the I/O time, but at the same time, it also performs well in reducing the processor and main memory gap. Our theoretical analysis and experimental results show that the new strategy can result in significant improvements for relational DBMSs.

3 Background

Existing data placement techniques for MEMS-based storage basically adapt disk-based techniques by mapping MEMS-based storage devices into disk-like devices. Furthermore, these data placement techniques are designed for general file systems. Our focus in this paper is to exploit the two-dimensional characteristics of MEMS-based storage devices for the placement of relational data. For this purpose, we first describe the mapping from MEMS-based storage to disks, then we review the existing relational data placement scheme.

3.1 Mapping MEMS-based Storage into Disk-like Devices

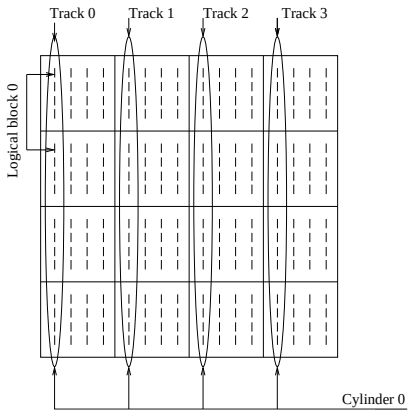


Figure 3: The definitions of cylinders, tracks, and sectors in MEMS-based storage

The media sled is logically divided into rectangular regions as shown in Figure 2. Each region contains $M \times N$ bits, and data is accessible by tip sectors (or sectors, as defined in Section 2). Using the disk terminology, a *cylinder* is defined as the set of all bits with identical x offset within a region; i.e., a cylinder consists of all bits accessible by all tips when the sled moves only in the Y direction [8]. For example, Cylinder 0 is shown in Figure 3 and is highlighted by four ellipses. In theory, all tips can be activated simultaneously to access data, however due to the power and heat considerations, only a subset of them can be activated simultaneously. Thus cylinders are divided into tracks. A *track* consists of all bits within a cylinder that can be read by a group of concurrently active tips. In Figure 3, four out of sixteen tips can be activated concurrently, and each cylinder contains 4 tracks. Track 0 to Track 3 of Cylinder 0 are shown in Figure 3. Tracks are composed of multiple sectors. Sectors on MEMS-based storage are tip sec-

tors, which contain less data than sectors of disks. Sectors can be group into logical blocks. In [8], each logical block is 512 bytes and striped across 64 tips. Therefore, by using the MEMS-based storage devices in Table 1, there are up to 20 logical blocks that can be accessed concurrently (1280 concurrent tips$/64$ tips $= 20$). Data is organized in terms of logical blocks on MEMS-based storage. During a request, only those logical blocks needed are accessed. Having mapped the MEMS-based storage into disk-like, we review the existing data placement scheme for relational database systems.

3.2 Review of existing data placement scheme

Traditionally, relational DBMSs use the N-ary Storage Model (NSM) [10] to store records in a relation in slotted disk pages, where each disk page has the same size. NSM organizes records sequentially on disk pages. At the end of each page, an offset table is used to locate the beginning of each record, to handle variable-length records. The initial part of a relation (*StudentGrade*) with four attributes is illustrated in Table 2. Figure 4 depicts an NSM page layout containing the first four records of the relation *StudentGrade*, where P_1 to P_4 are pointers pointing to the beginning of data records 1 to 4 respectively. Therefore, a record in a page can be accessed by following its corresponding pointer at the end of each page.

Table 2: Relation *StudentGrade*

attributes	name char(16)	perm ID int(8)	age int(8)	grade int(8)
record1	Mary	572	19	86
record2	John	582	18	90
record3	Bob	511	18	80
record4	Jane	537	20	91

In general, only a few attributes of each record are needed to answer most queries. For example, consider the following SQL query:

SELECT *name*
from *StudentGrade* (1)
where $grade > 90$*;*

In this query, only attribute values corresponding to $name$ and $grade$ need to be retrieved from the *StudentGrade* relation. However, due to the record-by-record layout of a relation in NSM, all records of the table *StudentGrade* will be retrieved. Furthermore, if all of these records move from the memory to the CPU L1/L2 cache, the cache will be polluted due to unnecessary parts of each record. In this example, the attribute values *perm ID* and *age* will be loaded into the cache. This results in poor utilization of the cache capacity which leads to a high number of cache misses.

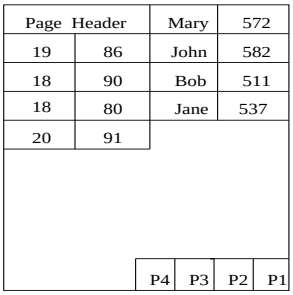


Figure 4: An NSM page layout for Relation *StudentGrade*

To optimize the I/O utilization, the Decomposition Storage Model (DSM) [6] was proposed. Instead of placing all record attributes in one page, DSM vertically partitions a relation into sub-relations based on the number of attributes. For a given relation with k attributes, DSM creates k sub-relations corresponding to each attribute. Each sub-relation has two attributes, a logical record number and an attribute value from the relation. Each sub-relation is organized into pages in the same way as NSM. Figure 5 shows how DSM partition the relation *StudentGrade*. Each sub-relation is accessed when the corresponding attribute is needed. Hence, for aggregate queries, DSM performs very well. However, if multiple attributes are involved in a single query, DSM needs to execute expensive join operation for the needed sub-relations [4]. In particular, for the example query considered above, the system will fetch the sub-relations corresponding to $name$ and $grade$. Before the query can be executed, the two sub-relations need to be composed by performing a join operation, which in general has a large performance penalty. Because of this drawback, most DBMSs use the NSM data placement strategy. In our experiments, we will not consider DSM further.

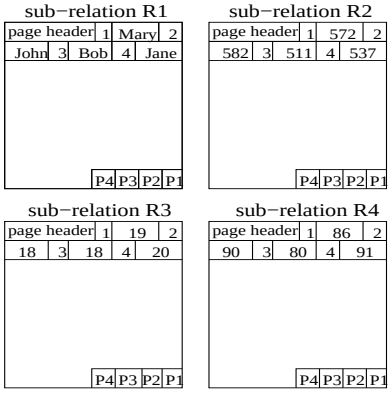


Figure 5: Placing the data of the relation *StudentGrade* in DSM pages

When considering predicate evaluations under modern database workloads, NSM does not perform well. This is because, when a query needs only a fraction of the attributes, NSM loads useless data into the cache and pollutes the cache. Hence, NSM is not cache-friendly. Optimizing cache utilization and performance (i.e., bringing as

much useful data as possible into cache) is becoming increasingly important for modern DBMSs. Based on this observation, Ailamaki, et al., proposed Partition Attributes Across (PAX) [4] recently. Within each page, unlike NSM, PAX groups all values of each attribute into a mini-page. A page is divided into mini-pages based on the number of attributes. So PAX stores the same data as NSM in each page. An example is shown in Figure 6. PAX optimizes inter-record spatial locality in the cache for each attribute in a page. Therefore, when only a subset of attributes is involved in the queries, PAX would not pollute the cache with values of unused attributes in queries, PAX has better cache utilization than NSM. Even though DSM can achieve similar cache utilization, DSM needs to spend a large amount of CPU time to join the involved attributes in queries [4]. For this reason, even though DSM can save I/O time, experimental results showed that DSM performs worse than NSM and PAX when multiple attributes are needed during query processing.

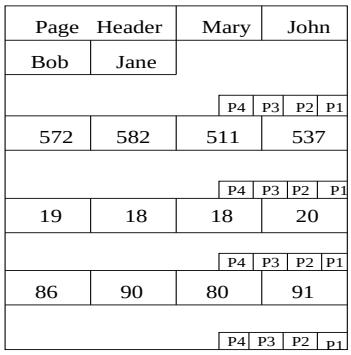


Figure 6: An PAX page layout for Relation *StudentGrade*

All of these three data placement strategies are in the context of a single page. Data on disks are mapped to a single dimension as required by disk structures. So if the MEMS-based storage devices are adapted to disk-like devices, relations in DBMSs will lose their two-dimensional property. During predicate evaluation, when only a subset of the attributes is needed, both NSM or PAX bring values of all attributes in a page into the memory. Consequently, a large amount of memory is used to store unnecessary data. Since unnecessary data is retrieved from disk in NSM and PAX, I/O utilization is poor. To address this problem, we propose a new strategy for placing relational data on MEMS-based storage devices. In the new data layout, we bring in only the needed data into the memory for query processing by exploring the two-dimensional physical characteristic of the media sled in MEMS devices. This approach can optimize cache performance as well as improve the I/O utilization by exploiting the physical property of MEMS-based storage devices.

4 New data placement for MEMS-based storage devices

In this section, we first describe our new data placement scheme, Flexible Retrieval Model (FRM) for MEMS-based storage devices. Then we show how to retrieve data from MEMS where FRM is used for the data layout. Finally we analyze FRM on the basis of I/O utilization, memory usage, and cache performance in comparison to NSM and PAX.

4.1 Data Layout

Since MEMS-based storage devices are not commercialized yet, there is no standard specification. In this paper, we use the CMU CHIPS [3] as our physical device model. The parameters are given in Table 1. In this model, only 20% (1280 out of 6400) of the total tips can be activated simultaneously because of power and heat-dissipation constraints. This places a constraint on the maximum number of records that can be retrieved simultaneously by a given query from a relation R that is stored on a MEMS-based storage device. MEMS-based storage is organized into logical blocks. Given the parameters in Table 1 and assuming that the logical block size is 512 bytes, there are 20 logical blocks which can be retrieved simultaneously. In each logical block, the number of records is $\lfloor 512/S \rfloor$, where S is the size of a record in R in bytes. For example, given the relation *StudentGrade* in Table 2, $\lfloor 512/40 \rfloor = 12$, then 240 records (20 logical blocks $\times$ 12 records/logical block) can be retrieved concurrently. However, if only a subset of the attributes is involved in a query, for example, attributes *perm ID* and *grade*, the ideal goal will be that all the 1280 tips are used to retrieve the values corresponding to *perm ID* and *grade*. For this case, $1280 * 8/16 = 640$, i.e., 640 records can be retrieved simultaneously. Thus I/O utilization increases by more than a factor of 2 (640/240=2.67) in terms of the number of records retrieved from the relation *StudentGrade*. Based on this observation, we propose FRM, where data can be accessed by tip sectors instead of logical blocks, thus it is possible to maximize the number of concurrent tips on the data relevant to the query.

In a MEMS-based storage device, the smallest accessible unit is a tip sector, which is determined by the (x, y) coordinates of the media sled and a tip number. The media sled has 6400 tips which are divided into 80×80 tip regions, where each tip corresponds to one tip region. In the FRM scheme, a relation R is striped across the various tip regions. We first determine how many tip sectors each attribute field needs, and distribute the attribute over tip sectors with the same (x, y) coordinates of the consecutive (neighboring) tip regions. For example, in our case, a tip sector is 8 bytes and if an attribute needs 32 bytes, then this attribute is placed on 4 tip sectors with the same (x, y) coordinates in four consecutive tip regions. Based on this, we can determine the number of tip sectors with the same coordinates in consecutive tip regions a single record in a relation will occupy. Hence in the relation *StudentGrade*, based on the size of its attributes ($16 + 8 + 8 + 8 = 40$),

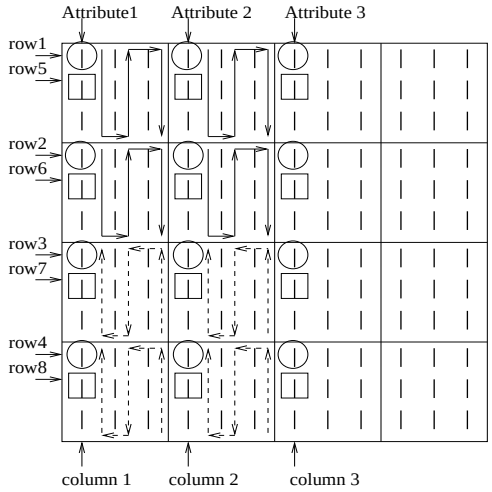


Figure 7: An example of FRM

a record occupies 5 tip sectors with the same coordinates in consecutive tip regions. For maximum concurrency, we then place subsequent records on the remaining tip sectors with the same coordinates in the same row without splitting a record over two different rows (hence, there may be some columns left if there is not enough space to store a whole record), and then repeat this placement scheme row by row, all of tip sectors with the same (x, y) coordinates. In our example, in an 80×80 MEMS device, we can place $6400/5 = 1280$ records at the same (x, y) coordinates of the 6400 tip regions. To have better understanding of how FRM works, we give the formal description of FRM in the following two steps.

Step 1: For each relation, allocate tip sectors for its attributes to guarantee that each row of consecutive tip regions with the same coordinates contain multiple entire records.

Given a relation R, we use the following formulas to determine how many tip sectors a single attribute needs and how many records can be stored in a row of tip regions. (Note, when we say a row of tip sectors, we always mean a row of consecutive tip sectors with the same coordinates.)

An attribute i of size S_i needs to be stored on an integer multiple of tip sectors. In our model a tip sector is of size 8 bytes, hence attribute i needs $\lceil S_i/8 \rceil$, call this T_i. Hence a record needs

$$n = \sum_{i=1}^{n_r} T_i.$$

tip sectors, where n_r is the number of attributes in R. The number of records that can now be stored in a row of tip regions is:

$$recordsInRow = \lfloor T_r/n \rfloor.$$

where T_r is the number of tip regions in a row, in our case 80.

An example is shown in Figure 7. In this example, there is a relation with three attributes, the size of each attribute is 8 bytes. We are trying to place this relation in 4×4 MEMS-based storage. Based on Step 1, each row of tip sectors with the same coordinates can only hold one record. A row of tip sectors highlighted with circles/rectangles contains one record of this relation respectively.

Step 2: Allocate a new row of tip sectors to a relation.

In FRM, the data is placed in terms of a unit of a row of tip sectors with the same coordinates, (for simplicity, we use "a row of tip sectors" to denote "a row of tip sectors with the same coordinates"). We next describe how to allocate a new row of tip sectors to a given relation R. Based on the property of MEMS-based storage, all the tip sectors with the same (x, y) coordinates have the potential to be activated simultaneously, we will choose the row of tip sectors which has the same coordinates as the previously allocated row as the new one. For example, in Figure 7, row 1 is allocated to record 1, then we will allocate row 2 to record 2 (The row numbers are given for the convenience of description). However, if there is no row of tip regions with the same coordinates as the last allocated row available, we will consider the physical movement property of MEMS-based storage. Based on the data provided in [7], in order to decrease seek time (time for moving media sled from one position to another), we should first consider a new row which has the same x coordinate, and y coordinate offset by one from the previously allocated row. For example, in Figure 7, row 5 has the same x coordinate as row 4, and y coordinate differs by one with that of row. Thus we allocate row 5 for record 5. If there is no such rows available, which means the media sled has to be moved in the X direction, then we will allocate a new row which has the same y coordinate, but x coordinate differs by one from the previously allocated row. The algorithm is given in Appendix A.

4.2 Data Retrieval

Having described how to place the records by rows of tip regions, we now explain how to retrieve the data. Based on the property of MEMS-based storage, all the rows of tip regions with the same coordinates have the potential to be accessed concurrently. However, due to the constraint that only part of the tips can be activated simultaneously, the rows of tip regions may need to be accessed multiple times to retrieve all the useful data. There are two cases to consider. If the number of tips (N_a) for the involved attributes is smaller than the maximum number of concurrent tips, these tips can be activated simultaneously to retrieve the needed data at the rows of tip sectors at the same coordinate. In this case, the media sled first moves in Y direction to read all of the rows with different y coordinate, then it moves in X direction and turns around to read the next column of rows of tip sectors[1]. However, if N_a is larger than

[1]Tip sectors can be read in both directions.

the maximum number of concurrent tips, then the number of retrievals for the rows of tip regions with one coordinates is equal to $\lceil N_a /$the number of maximum concurrent tips$\rceil$. The algorithm is given in Appendix B. For example, in Figure 7, assume that the maximum concurrent tips number is four. If only Attribute 1 is involved in a query, to read the values of attribute 1 in row 1 to 4, we first activate four tips at row 1 to 4 in column 1; then move the media sled up and down to retrieve the values of Attribute 1. If Attribute 1&2 are involved in a query, to read values of attribute 1&2 at row 1 to row 4, we need to first activate tips at row 1&2 and column 1&2, then move the media sled up and down to retrieve all the values of Attribute 1&2 at these four tip regions (the solid lines in Figure 7 shows the media sled's movement). Next, the tips at the row 3&4 and column 1&2 are needed to be activated to retrieve the values of Attribute 1&2 at these four tip regions, the dashed lines show the media sled's movement at this part. Without introducing more seek time, we move the media sled in the reverse direction of the solid line direction instead of moving the media sled back to the start position.

4.3 Analysis of FRM

The data placement scheme affects the data flows from secondary storage to main memory, and from main memory to cache. Hence, we focus on evaluating and analyzing FRM in terms of memory usage, I/O performance, and cache utilization. As discussed before, one of the primary savings is achieved in terms of memory usage. Because the new data placement scheme is relation conscious, which allows retrieval of only that data which is explicitly needed for a query, the smaller the number of attributes involved in queries the more memory is saved for other data. The I/O time savings are due to the fact that we maximize the concurrent tips to retrieve only the necessary data. In other words, more records are retrieved at each access. On the other hand, since the existing data placement techniques are based on disk pages or logical blocks which are storage access units, data in a complete disk page or a logical block has to be retrieved together. Hence, a lot of memory and I/O time can potentially be wasted to retrieve the unrelated data in disk pages or logical blocks.

Based on the memory hierarchy, there are two bottlenecks: secondary storage and main memory, main memory and processors (cache misses). We have shown that FRM alleviates the first bottleneck. Next we are going to analyze cache performance of FRM. In FRM, after retrieval of data from MEMS-based storage into memory, the records are organized into the similar way as NSM. Because the basic access unit of MEMS-based storage is a tip sector, the retrieved data is organized by rows of tip sectors in memory. Thus, the data is placed in NSM-like manner in the memory pages. However, it can achieve better cache utilization than NSM. In FRM, because a relation is placed in two dimension, retrieving the data of a subset of attributes is supported. For a query, only the data of the involved attributes are brought into the memory, and then into the cache. If all tuples are processed, all data brought into cache will be used due to the intra-record spatial locality. The data in the cache for the example query (2) is shown in Figure 8.

SELECT *perm ID*
from *StudentGrade* (2)
where $grade > 90$*;*

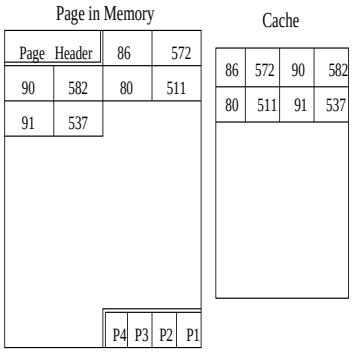


Figure 8: FRM approach page layout and cache utilization

In fact, FRM has similar cache performance as PAX except that PAX minimizes data cache miss delays by making use of the inter-record spatial locality. If all attributes are involved in projection queries, NSM, PAX, and FRM have the same cache performance, because none of them bring useless data into the cache. When considering selection range queries, all of these three approaches cause cache pollution. NSM may bring useless data into cache when the predicate is not satisfied or some values of non-selected attributes are brought into cache with the necessary data. PAX pollutes cache when data of some unsatisfied records are brought into the cache along with some satisfied records. FRM causes cache pollution when the predicate fails, the values of some other attributes of the record pollute the cache. Because FRM does not contain data of useless attributes, FRM would not pollute the cache with the values of uninvolved attributes. Thus, FRM always performs better than NSM in terms of cache utilization.

5 Evaluation of FRM

In this section, we conduct a set of experiments to evaluate the cache performance and analyze the I/O utilization and memory usage of FRM in comparison to NSM and PAX. First we analyze FRM performance on I/O utilization and memory usage. Then we evaluate FRM's cache performance. For our evaluation, we generated a dataset with 1.28 million records, and each record consists of sixteen 8-byte attributes. Ailamaki, et al. [4], used a similar data setting for evaluating PAX. In particular, the authors used a relation with eight 8-byte attributes containing 1.2 million records.

The queries we conducted are the variations of the following range selection query:

SELECT $A_1, A_2, \ldots, A_n$
from *R*
where $A_1 > Bound$; (3)

where A_i are attributes in the relation R. Because the FRM approach does not retrieve useless attributes in queries, its cache performance would not be affected by the relative positions of A_i ($i = 1, \ldots, n$) in queries. We know that PAX is also not affected [4], while NSM is significantly impacted. In our experiments, we give NSM advantage by always posing queries over physically adjacent attributes.

5.1 Memory Utilization

In general, given a relation with n attributes and a query which involves m attributes (where $m \leq n$), the memory space saving in FRM approach is

$$(\text{the size of m attributes}/\text{the size of n attributes}) \times M,$$

where M is the memory space used for NSM and PAX approaches. In Figure 9 we show the result for the relation R. For example, if queries for R involve one attribute, the FRM approach saves 93.75% ($(1 - (1 \times 8)/(16 \times 8)) \times 100\%$) of the memory space used by NSM and PAX methods. NSM and PAX have identical memory requirements, since the storage access unit is a disk page or a logical block, and they need to retrieve the same set of pages, with the same content (the difference between the two is the organization within each page).

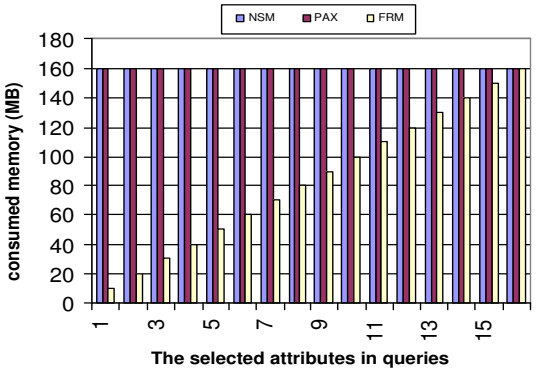


Figure 9: Comparison of Memory Space Usage in NSM/PAX/FRM

5.2 I/O Performance

In MEMS-based storage devices, we have the following time equations [7]:

$$time_{service} = time_{seek} + time_{transfer}.$$

Since all data for a relation are placed continuously in storage device, we have

$$\begin{aligned} time_{seek} &= time_{trackswitchtime} \\ &= time_{settletime} + time_{turnaround}. \end{aligned}$$

In the NSM and PAX models, the number of records in a page or a logical block is the same, so they have the same I/O time. The I/O time corresponding to different queries for a relation does not vary, because the data is accessed based on logical blocks. However, in our FRM model, the records of a relation are placed by the unit of tip sectors, each attribute is striped over multiple tip sectors. Thus, based on the property of MEMS-based storage (multiple tips can be activated concurrently), subsets of attributes of a relation can be accessed. Due to the constraint of maximum concurrent tips, the I/O time may be different corresponding to different queries. Given relation R, we perform a theoretical I/O time analysis for NSM, PAX, and FRM.

First, we compute the I/O time for NSM and PAX. For a realistic comparison of the three approaches, for NSM and PAX, we assume pages are stored on a MEMS-based storage device with the same characteristics as the one used for FRM. However, for NSM and PAX, the MEMS-based storage device is treated as though it simulates a disk [8]. We therefore need to calculate the transfer and seek time for returning the disk pages or logical blocks to answer a given query. As we mentioned before, NSM and PAX have the same I/O time due to the page-based access. Each page size or logical block size is 512 bytes which can be stored in 64 tip sectors. The number of the maximum concurrent tips is 1280. Hence each access can, at most, retrieve 20 logical blocks. The number of records in relation R is 1,280,000 and each record is 16×8 bytes. So R populates 320,000 pages, which can be retrieved in 16,000 ($320,000/20$) transfers. Each access can be done in 0.129 ms [7]. The total of $time_{transfer}$ for retrieving relation R is

$$time_{transfer} = 16,000 \times 0.129 = 2064 \text{ ms}.$$

Next, we compute the $time_{seek}$. In each tip sector, there are 2000×2000 bits. Because some bits are used for error correction, each column can have twenty-two 8-byte tip sectors. The tip regions are arranged as 80×80 tips. Thus each column of tip regions can have 1760 (80×22) tip sectors. The relation R populates $16,000 \times 1280$ tip sectors. Because the number of the maximum concurrent tips is 1280, and data is accessed in Y direction, we can use the first 16 ($16 \times 80 = 1280$) column tip regions for R. So the number of columns n_c in a tip region is

$$n_c = \lceil 16,000 \times 1280/(1760 \times 16) \rceil = 728.$$

Moving the media sled from one column to the next needs some time to turn around and settle the media sled. From Table 1, the time for turning and settling is 0.275 ms ($0.215 + 0.06$). So the total seek time is

$$time_{seek} = 728 \times 0.275 = 200.2 \text{ ms}.$$

Thus, the I/O time of NSM and PAX for relation R is 2264.2 ms, as shown in Figure 10.

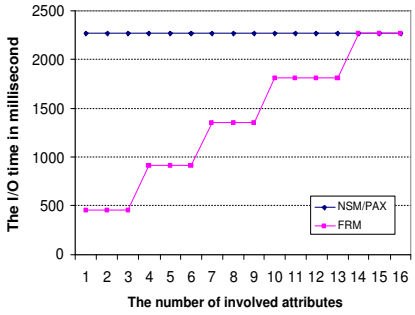


Figure 10: Comparison of I/O time for NSM/PAX/FRM

Now, we compute the I/O time for FRM. For relation R with sixteen 8-byte attributes, we store 5 (80 tip regions in a row$/16$) tuples per row of tip sectors with the same coordinates. Hence, all the rows of the tip sectors with the same coordinates can store 400 (80 rows $\times$ 5 records per row) records. Hence, we need 3200 ($1,280,000$ total records$/400$) these kind of rows with the same coordinates. Based on Step 2, since there are 22 tip sectors in each column of a tip region, the number of columns used in a tip region is 146 ($\lceil 3200/22 \rceil$). Based on these parameters, if one attribute is involved in a query, there are 80 rows $\times$ 5 records per row tip sectors allocated for this attribute in all the rows with the same coordinates. (which is smaller than maximum concurrent tips number, 1280), then these tips can be activated concurrently. Each access can read 400 records. There are 3200 accesses and each access takes 0.129 ms. Thus we have

$$time_{transfer} = 3200 \times 0.129 = 412.8 \text{ ms},$$

$$time_{seek} = 146 \times 0.275 = 40.15 \text{ ms}.$$

Thus, if only one attribute needs to be retrieved, the I/O time of FRM is 452.95 ms. The same I/O time can be achieved when two or three attributes are involved in queries, because the number of tips that need to be activated to retrieve the data of these attributes is still smaller than 1280 (which leads to a flat line segment in Figure 10). However, if four attributes are involved in a query, the total number of tips for the four attributes is 80 rows $\times$ 4 attributes $\times$ 5 records in a row, which is larger than 1280; then we need to access every rows-of-tip-sectors with the same coordinates twice to retrieve the data (however, there is no extra seek time to be considered as we explained in Section 4.3). The corresponding transfer time is 412.8×2, and seek time is 40.15×2. By doing the similar analysis for all kinds of projection queries, we get the results in Figure 10. From Figure 10, we can see that FRM performs much better than NSM and PAX in terms of I/O time especially when the projectivity is low; i.e., only a few attributes are needed to process the query.

5.3 Cache Utilization Analysis

In the main memory, the data placement scheme only affects the data flow from main memory to processor, which is evaluated by cache misses. Thus, we conducted experiments to compare the cache behavior of NSM, PAX, and FRM. First we describe our experimental setup, then we report the results of our experiments based on different selectivity and projectivity.

5.3.1 Experiment Setup

The experiments were conducted on a system with Pentium II Celeron 433×2 processors.This computer has two-level caches. The L1 cache is 16KB with 32 bytes long cache line. A cache miss at this level results in a 20-ns penalty. The L2 cache is 128KB with 32 bytes long cache line. The cache miss penalty at this level is 200 ns. We collected this information by using the calibrator provided by Stefan Manegold [1]. In the experiments, we counted the L1 and L2 data cache misses using PAPI [2], which provides a programming access to the hardware performance counters available in most microprocessors.

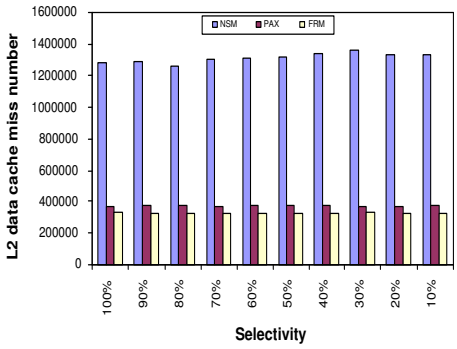


Figure 11: Data cache penalty of NSM/PAX/FRM with one selected attribute

We use relation R and query (3) described at the beginning of this section. In order to verify the cache performance for each approach, we simulated selection and projection queries on relation R which was resident in the memory during our experiments. In projection queries, we vary the number of attributes involved, and those attributes that are involved in the query are physically adjacent to each other to favor NSM.

5.3.2 Experimental results for selection queries

We executed query (3) by varying the number of attributes, and changed the value of $Bound$ to control the selectivity. Figure 11 shows the results of NSM/PAX/FRM L2 data cache miss number with one attribute involved (The L1 cache miss number is almost the same as the L2 one, and cache miss penalty of L1 cache is much smaller than L2 cache miss. Thus we only show the L2 cache miss number in the result figures). Since only one attribute, A_1, is

involved in the query, every record is examined to check if $value(A_1) > Bound$. When accessing every record, NSM brings the values of four attributes of this record into a cache line, 75% space of a cache line is wasted because the values of other three attributes are useless for this query. However, PAX and FRM (FRM contains only one attribute A_1) bring the value of attribute A_1 into cache. Because there is only one attribute A_1 in this query, the query has to check the value of A_1 in every record; selectivity has no effect on these three approaches. This analysis is supported in Figure 11, which shows that FRM and PAX perform three times better than NSM (the minor difference between PAX and FRM is due to the implementation).

For Figure 12, we evaluated query (3) with two attributes. As explained, the relative position of the two selected attributes does not affect the cache performance of FRM and PAX. To simplify the analysis of NSM, we select attribute A_2 with attribute A_1. Because the value of A_1 and A_2 in one record can be in one cache line, the cache performance of NSM does not change compared to the query with one attribute (as shown in Figure 11), the result is shown in Figure 12(a). Without considering the selectivity, FRM and PAX have similar cache performance, because both of them do not pollute the cache whereas NSM brings two useless attributes' values in each cache line. Thus, PAX and FRM incur 50% less cache penalty than NSM. However if the selectivity is not 100%, PAX and FRM may pollute the cache too. We analyze this by examining the placement of data in cache lines for the two models. In Figure 12(b), $A_{i,j}$ is a value of attribute A_i of the j^{th} record. From this figure, we can see that FRM pollutes the cache because all values of attribute A_2 are brought into cache with the values of A_1, thus the cache performance of FRM does not vary with the selectivity. In the PAX approach, some values of attribute A_2 are not brought into cache if the corresponding values of attribute A_1 are not satisfied. In Figure 12(b), assume the four values of A_1 ($A_{1,1}, A_{1,2}, A_{1,3}, A_{1,4}$) are not satisfied, then none of the corresponding values of A_2 are brought into cache. So in this query, FRM has more cache pollution than PAX and does not change with different selectivity.

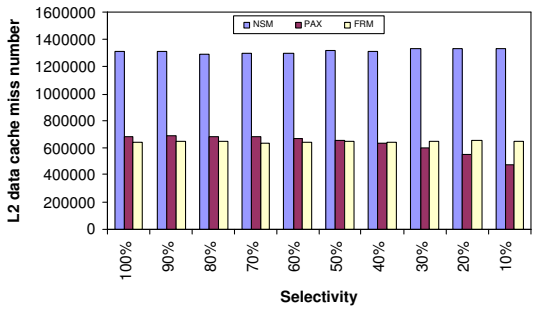
From Figure 12 we may conclude that PAX has better cache utilization than FRM at lower selectivities. But this is not always the case. PAX also pollutes the cache under some conditions. Figure 13(a) shows the cache performance of a query with 13 attributes involved; we observe that PAX does not perform as well as NSM and FRM. In this query, A_1 to A_{13} are the selected attributes. As described previously, $A_{i,j}$ is a value of attribute A_i of the j^{th} record. Figure 13(b) shows the cache behavior of NSM/PAX/FRM. When $A_{1,1}$ is evaluated, the values of attribute A_1 of the next three records are brought into one cache line with $A_{1,1}$. Assume $A_{1,1}$ satisfies the condition, then the values of the other twelve attributes, $A_{i,1}(i = 2, \ldots, 13)$, of this record need to be accessed. Each access of the attribute $A_{i,1}(i = 2, \ldots, 13)$ will bring the other three values of the same attribute into cache line based on the layout of PAX mini-pages, as shown in Figure 13(b) (i.e., the next three records are brought into cache with the record $A_{i,1}$). If any value of $A_{1,j}(j = 2, 3, 4)$ does not satisfy the qualifier, the cache will be polluted by the values of the other twelve attributes in the corresponding record. NSM, on the other hand, does not bring any value of other records even though it pollutes the cache with the data of the three unselected attributes, as shown in Figure 13(b). For FRM, it brings the data of the 13 selected attributes into the cache, and it also bring the part of the data of the next record. In both of NSM and FRM, if a record does not satisfy the qualifier, the cache will be only at most polluted with the values of the three other involved attributes of the record. In general, the degree of cache pollution depends on the size of selected attributes, projectivity, selectivity, and the size of the cache line.

5.3.3 Experimental results for projection queries

In this section, we discuss experiments conducted to evaluate the relationship between cache performance and projectivity. In the first experiment, as shown in Figure 14, selectivity is set to 100% and the number of involved attributes is varied; PAX and FRM have similar cache performance, because both of them do not bring useless data into cache and maximize the cache utilization. For PAX and FRM, the number of cache misses is proportional to the number of attributes involved in queries and the number of records in R. Both PAX and FRM exhibit linear increase in the cache penalty as the number of attributes is increased. However, NSM exhibits a stepwise behavior. In Figure 14, each step demonstrates that accessing each record results in one more data cache miss than before. For example, let us analyze the first step where the number of selected attributes is changed from 4 to 5. In this query, when the value of A_1 in each record is examined (there is one cache miss), the next three attributes' values are brought into one cache line with the value of A_1 in the same record. So, if only the first four attributes are involved in the query, there is only one cache miss per record, which results in the flat segment from 1 to 4 of NSM curve. However if the query involves the first five attributes, when the program accesses the fifth attribute value, which is not in the cache (because each cache line can only hold the first four attributes values), one more cache miss will result for this record. Hence, the cache miss penalty of the query with five attributes examined is twice of the cache miss penalty of the query with four attributes.

In real DBMSs, most queries involve both selection and projection operations. From section 5.2, we saw the relation between selectivity and cache performance with given projectivity. In the following we explore the effects of projectivity on cache performance of NSM/PAX/FRM with a given selectivity.

We conducted an experiment with selectivity set to 50% as shown in Figure 15. The result shows that when the query involves six attributes or fewer, PAX and FRM have fewer data cache misses than NSM. However, when the



(a)

The cache behavior of FRM

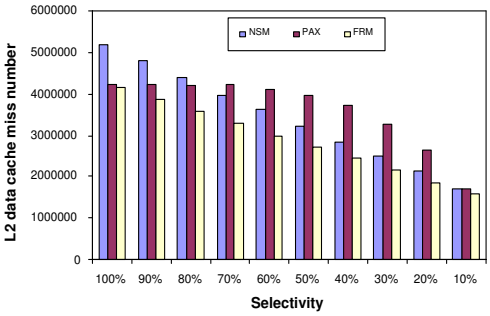
$A_{1,1}$	$A_{2,1}$	$A_{1,2}$	$A_{2,2}$
$A_{1,3}$	$A_{2,3}$	$A_{1,4}$	$A_{2,4}$

The cache behavior of PAX

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$

(b)

Figure 12: Data cache penalty of NSM/PAX/FRM with two selected attributes



(a)

Cache behavior of FRM

$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{4,1}$
$A_{5,1}$	$A_{6,1}$	$A_{7,1}$	$A_{8,1}$
$A_{9,1}$	$A_{10,1}$	$A_{11,1}$	$A_{12,1}$
$A_{13,1}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
⋮	⋮	⋮	⋮

Cache behavior of NSM

$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{4,1}$
$A_{5,1}$	$A_{6,1}$	$A_{7,1}$	$A_{8,1}$
$A_{9,1}$	$A_{10,1}$	$A_{11,1}$	$A_{12,1}$
$A_{13,1}$	$A_{14,1}$	$A_{15,1}$	$A_{16,1}$
⋮	⋮	⋮	⋮

Cache behavior of PAX

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
⋮	⋮	⋮	⋮
$A_{13,1}$	$A_{13,2}$	$A_{13,3}$	$A_{13,4}$

(b)

Figure 13: Data cache penalty of NSM/PAX/FRM with 13 selected attributes

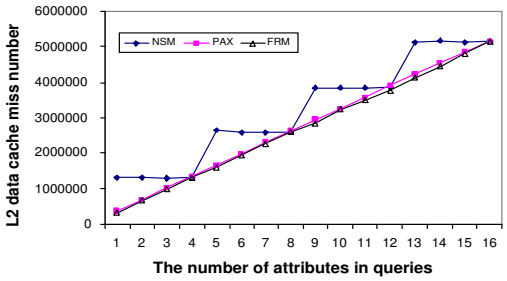


Figure 14: Data cache penalty of NSM/PAX/FRM with selectivity = 100%

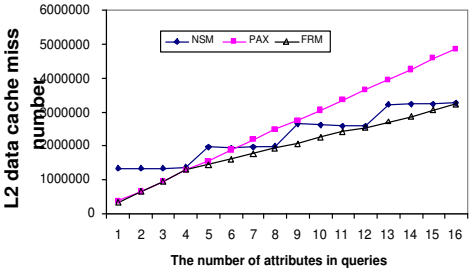


Figure 15: Data cache penalty of NSM/PAX/FRM with selectivity = 50%

projectivity is increased, NSM and FRM have better cache utilization than PAX. In order to better understand the effect of NSM/PAX/FRM on cache performance, we set selectivity to be 10% and vary the projectivity; the results are shown in Figure 16. When both the selectivity and the projectivity are low, PAX has better cache performance than FRM and NSM. However, when we increase the projectivity, the cache performance of PAX becomes worse than FRM.

5.3.4 Discussion

Based on the analysis and experimental results, we conclude that NSM and PAX require the same I/O time and

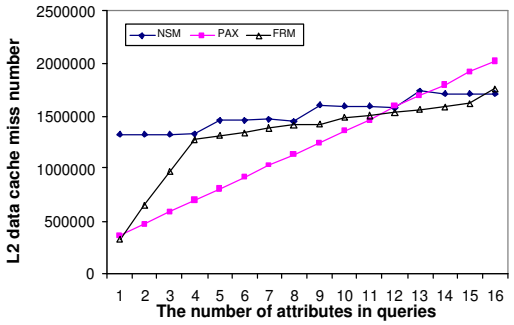


Figure 16: Data cache penalty of NSM/PAX/FRM with selectivity = 10%

memory resources for all queries. In FRM, the fewer the number of attributes involved in the queries, the more I/O time and memory space are saved. In terms of cache utilization, all these three methods are sensitive to query selectivity and projectivity. When selectivity and projectivity are very low, PAX has better cache performance than FRM. However, under this condition, FRM saves more I/O time compared to PAX. Time savings by memory access is much smaller than the savings by I/O time. Let's take a look at an example. In Figure 16, PAX outperforms FRM by 150 milliseconds when the number of involved attributes is four [2]. However, in this case, FRM saves 1358.3 milliseconds more than PAX for I/O time. Thus, by accounting for both I/O savings and cache performance, FRM outperforms NSM and PAX.

6 Conclusion

MEMS-based storage devices are being developed to alleviate the storage/memory bottleneck. Most research up-to-date has been conducted from an operating system centric point of view [8]. In order to integrate MEMS-based storage devices into traditional computing environment, this approach advocates simplifying the usage of such novel devices by mapping them to a disk-like setup. Although beneficial in the short term, this approach may not be able to fully exploit some of the intrinsic properties of MEMS-based storage, especially from a database point of view. MEMS-based storage devices are two-dimensional in nature and present a unique opportunity for storing two-dimensional relational databases. In this paper, we presented a first attempt to approach the data placement problem for MEMS-based storage devices in a manner that exploits potential properties of relational databases. Our development shows that in terms of I/O utilization, the Flexible Retrieval Model (FRM) is very beneficial and results in significant performance improvements when compared to existing placement techniques for relational data. It is also cache-friendly by retrieving only the relevant values required for evaluating a query; in this way it further pushes some of the ideas earlier developed in PAX [4] by not only avoiding cache pollution, but also memory pollution. As our performance results also show, there is still room for improvement. We plan to explore other cache-friendly techniques for MEMS-based storage devices that can further benefit from the PAX approach, thus further avoiding unnecessary cache pollution in some cases. In conclusion, we believe this work represents a significant first step towards the incorporation of an important and significant storage device into the DBMS architecture. We also believe that our approach may have some impact on the early stages of architectural design of MEMS-based storage devices.

[2] Given the complexity of converting cache misses to real system time, we use an estimation. The value is computed by multiplying data cache misses number with each cache miss penalty, which is larger than the real one without consideration of system architectures.

References

[1] A cache-memory and tlb calibration tool. 2001. http://www.cwi.nl/ manegold/Calibrator/calibrator.shtml.

[2] Performance application programming interface. 2001. http://www.ece.cmu.edu/research/chips.

[3] CMU CHIP project. 2002. http://www.lcs.ece.cmu.edu/research/MEMS.

[4] A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. Weaving relations for cache performance. *In proceedings of the 27th Conference on Very Large Databases*, pages 169–180, September 2001.

[5] L. Richard Carley, Gregory R. Ganger, and David F. Nagle. MEMS-based integrated-circuit mass-storage systems. *Communication of the ACM*, 43(11), November 2000. http://www.lcs.ece.cmu.edu/research/MEMS/.

[6] G.P. Copeland and S. F. Khoshafian. A decomposition storage model. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, May 1985.

[7] J. Griffin, S. Schlosser, G. Ganger, and D. Nagle. Modeling and performance of MEMS-Based storage devices. *Proceedings of ACM SIGMETRICS*, pages 56–65, June 2000. http://www.lcs.ece.cmu.edu/research/MEMS/.

[8] J. Griffin, S. Schlosser, G. Ganger, and D. Nagle. Operating systems management of MEMS-based storage devices. *Symposium on Operating Systems Design and Implementation(OSDI)*, October 2000. http://www.lcs.ece.cmu.edu/research/MEMS/.

[9] P.Vettider, M.Despont, U.Durig, W.Haberle, M.I. Lutwyche, H.E.Rothuizen, R.Stuz, R.Widmer, and G.K.Binnig. The "millipede"-more than one thousand tips for future afm storage. *IBM Journal of Research and Development*, pages 44(3):323–340, May 2000.

[10] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, 2000. WCB/McGraw-Hill and 2^{nd} edition.

[11] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. *ASPLOS*, November 2000. http://www.lcs.ece.cmu.edu/research/MEMS/.

[12] Michael Stonebraker. *Placement of Relational Data on Secondary Storage*, 2003. The Gong Show: First Biennial Conference on Innovative Data Systems Research.

[13] P. Vettiger and G. Binnig. The nanodrive project. *Scientific American*, pages 47–53, January 2003.

Appendix A The Algorithm for Data Layout

In this appendix, we give the algorithm for placing the relational data on MEMS-based storage as follows.

Begin
The number of tip regions of a MEMS-based storage is $T_r \times T_c$;
T_r is the number of tip regions in a row;
T_c is the number of tip regions in a column;
Relation R with attributes $A_1, A_2, A_i, \ldots, A_{n_r}$;
The size of each attribute is $S_i, i = 1, \ldots, n_r$;
The number of records is $numberOfRecords$;
The size of one tip sector is S_{tip};
The number of tip sectors for each attribute is T_i;
The number of tip sectors for a record is n;
the number of tip sectors in a row allocated for R is $sectorInRow$;
The number of records in a row is $recordsInRow$;

$T_i = \lceil S_i / S_{tip} \rceil$;
$n = \sum_{i=1}^{n_r} T_i$;
$sectorInRow = \lfloor T_r / n \rfloor \times n$;
$recordsInRow = \lfloor T_r / n \rfloor$;

Move the media sled to the initial position with coordinates (x_{start}, y_{start});
$x = x_{start}, y = y_{start}$;
$moveDirection = down$;
repeat
  Allocate $sectorInRow \times T_c$ tip sectors which the coordinates (x, y) for R;
  $leftRecordsNumber = numberOfRecords - T_c \times sectorInRow$;
  if $moveDirection = down$ **then**
    $y = y + 1$;
    if y is out of range of movement of the media sled **then**
      $y = y - 1, x = x + 1$;
      Move the media sled to the position (x, y), and turn it around;
      $moveDirection = up$
    end if
  else
    $y = y - 1$;
    if y is out of range of movement of the media sled **then**
      $y = y + 1, x = x + 1$;
      Move the media sled to the position (x, y), and turn it around;
      $moveDirection = down$
    end if
  end if
until $leftRecordsNumber \leq 0$
Done.

Appendix B The Algorithm for Data Retrieval

We describe the data retrieval algorithm in this appendix. We use the same names for the same parameters used in Appendix A, thus, we will not repeat their definitions in this algorithm.

Begin
Given a query Q;
The number of tip sectors in all rows with the same coordinates for the involved attributes is N_a;
The number of maximum concurrent tips is N_c;
Move the media sled to the initial position (x_{start}, y_{start});
$x = x_{start}, y = y_{start}$;
$moveDirection = down$;
$leftRecordsNumber = numberOfRecords$
if $N_a \leq N_c$ **then**
  repeat
    Activate all these tips and access the data;
    $leftRecordsNumber = numberOfRecords - T_c \times sectorInRow$;
    if $moveDirection = down$ **then**
      $y = y + 1$;
      if y is out of range of movement of the media sled **then**
        $y = y - 1, x = x + 1$;
        Move the media sled to the position (x, y), and turn it around;
        $moveDirection = up$
      end if
    else
      $y = y - 1$;
      if y is out of range of movement of the media sled **then**
        $y = y + 1, x = x + 1$;
        Move the media sled to the position (x, y), and turn it around;
        $moveDirection = down$
      end if
    end if
  until $leftRecordsNumber \leq 0$
else
  $numberOfAccess = \lceil N_a \leq N_c$;
  Divide the rows of tip sectors with the same coordinates into $numberOfAccess$ parts;
  repeat
    repeat
      Activate the next part of tips in the rows;
      $leftRecordsNumber = numberOfRecords - T_c \times sectorInRow$;
      if $moveDirection = down$ **then**
        $y = y + 1$;
        if y is out of range of movement of the media sled **then**
          $y = y - 1, x = x + 1$;
          Move the media sled to the position (x, y), and turn it around;

$moveDirection = up$
end if
else
$y = y - 1$;
if y is out of range of movement of the media sled **then**
$y = y + 1, x = x + 1$;
Move the media sled to the position (x, y), and turn it around;
$moveDirection = down$
end if
end if
until $leftRecordsNumber \leq 0$
Turn around the media sled, let $moveDirection$ equal its opposite;
$numberofAccess = numberOfAccess - 1$;
until $numberOfAccess = 0$;
end if
Done.