Efficient Indexing Data Structures for Flash-Based Sensor Devices

SONG LIN
University of California, Riverside
DEMETRIOS ZEINALIPOUR-YAZTI
University of Cyprus
and
VANA KALOGERAKI, DIMITRIOS GUNOPULOS, and WALID A. NAJJAR
University of California, Riverside

Flash memory is the most prevalent storage medium found on modern *wireless sensor devices (WSDs)*. In this article we present two external memory index structures for the efficient retrieval of records stored on the local flash memory of a WSD. Our index structures, *MicroHash* and *MicroGF (micro grid files)*, exploit the asymmetric read/write and wear characteristics of flash memory in order to offer high-performance indexing and searching capabilities in the presence of a low-energy budget, which is typical for the devices under discussion. Both structures organize data and index pages on the flash media using a sorted by timestamp file organization. A key idea behind these index structures is that expensive random access deletions are completely eliminated. MicroHash enables equality searches by value in constant time and equality searches by timestamp in logarithmic time at a small cost of storing index pages on the flash media. Similarly, MicroGF enables spatial equality and proximity searches in constant time. We have implemented these index structures in nesC, the programming language of the TinyOS operating system. Our trace-driven experimentation with several real datasets reveals that our index structures offer excellent search performance at a small cost of constructing and maintaining the index.

Categories and Subject Descriptors: C.2.M [**Computer-Communication Networks**]: Miscellaneous; H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: Wireless sensor networks, flash memory, access methods

An earlier version of this article appeared in Zeinalipour-Yazti et al. [2005]. This work was supported in part by grants from NSF ITR Nos. 0220148 and 0330481.

Authors' addresses: S. Lin, Department of Computer Science and Engineering, University of California, 900 University Avenue, Riverside, CA 92521; D. Zeinalipour-Yazti (corresponding author), Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678, Nicosia, Cyprus; email: dzeina@cs.ucy.ac.cy; V. Kalogeraki, D. Gunopulos, W. A. Najjar, Department of Computer Science and Engineering, University of California, 900 University Avenue, Riverside, CA 92521.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or direct commercial advantage, and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permission@acm.org.
© 2006 ACM 1553-3077/06/1100-0468 $5.00

1. INTRODUCTION

Rapid developments in wireless technologies and microelectronics have spawned a new generation of economically viable embedded sensor systems for monitoring and understanding the physical world [Warneke et al. 2001; Szewczyk et al. 2004; Intanagonwiwat et al. 2000; Sadler et al. 2004; Madden et al. 2002; Xu et al. 2004; Zeinalipour-Yazti et al. 2005]. Traditional sensing devices utilized over the years in meteorology, manufacturing, and agriculture are characterized by their passive mode of operation, considerable size, and wired connection to some centralized processing unit that enables storage and analysis. *Wireless sensor devices (WSDs)*, on the other hand, are tiny computers on chips that are often no bigger than a coin or credit card. These devices, equipped with a low-frequency processor (≈4–58MHz) and a wireless radio, can sense parameters such as light, sound, temperature, humidity, pressure, noise levels, and movement at extremely high resolutions. This multitude of features constitute WSDs' powerful devices that can be used for in-network processing, filtering, and aggregation [Madden et al. 2003; Madden et al. 2002; Yao and Gehrke 2003]. The applications of sensor networks range from environmental monitoring (such as atmosphere and habitant monitoring [Szewczyk et al. 2004; Sadler et al. 2004]) to seismic and structural monitoring [Xu et al. 2004] and industry manufacturing (such as factory and process automation [Crossbow 2005; Madden et al. 2002]).

One of the key challenges in this new era of sensor networks is the storage and retrieval of sensor data [Dai et al. 2004; Zeinalipour-Yazti et al. 2005; Ganesan et al. 2005]. Traditional techniques such as Madden et al. [2003], Deligiannakis et al. [2004], and Intanagonwiwat et al. [2000], work in a centralized way: The acquisition of data from the physical world is succeeded by transmission of the respective data to the *sink* (querying node). The centralized repository, which contains the full resolution of sensor data, can then be utilized to resolve different types of queries. Such centralized data acquisition scenarios have a common problem of large energy consumption, as the whole universe of readings is transferred towards the sink, thus leading to a shorter sensor lifetime.

In long-term deployments, it is often less expensive to keep a large window of measurements *in situ* (at the generating site) [Zeinalipour-Yazti et al. 2005] and transmit specific information to the sink only when requested. For example, biologists analyzing a forest are usually interested in the long-term behavior of the environment. Therefore, sensors are not required to transmit their readings to the sink at all times. Instead, the sensors can work unattended and store their readings locally until certain preconditions are met, or when the sensors receive a query over the radio that requests the respective data. Such in-network storage conserves energy from unnecessary radio transmissions, which can be used to increase the sampling frequency of the data, and hence the fidelity of the measurements, in reproducing the actual physical phenomena.

Currently, the deployment of sensor technology is severely hampered by the lack of efficient infrastructure to store locally large amounts of sensor data measurements. The problem is that the local RAM memory of sensor nodes is both

volatile and very limited (≈2KB–64KB). In addition, the nonvolatile on-chip flash memory featured by most sensors is also very limited (≈32KB–512KB). However, the limited local storage of sensor devices is expected to change soon. Several sensor devices, such as the RISE hardware platform [Neema et al. 2005; Banerjee et al. 2005], include off-chip flash memory which supplements each sensor with several megabytes of storage. Flash memory has a number of distinct characteristics compared to other storage media: First, each page (typically, 128B–512B) can only be written a limited number of times (≈10,000–100,000). Second, pages can only be written after they have been deleted in their entirety. Additionally, a page deletion always triggers the deletion of its respective block (≈8KB–64KB per block). Due to these fundamental constraints, efficient storage management becomes a challenging task.

The problem that we investigate in this article is how to efficiently organize the flash memory of a sensing device. Our desiderata are:

(1) To provide efficient access to the data stored on flash by *time* or *value*, for both *equality* and *range* queries generated by the user.
(2) To increase the *longevity* of flash memory by spreading writes out uniformly so that the available storage capacity does not diminish at particular regions of the flash media.

We present two new indexes, *MicroHash* and *MicroGF*, which serve as primitive structures for efficiently indexing temporal environmental and geographical data. Note that the data generated by sensor nodes has two unique characteristics: (i) Records are generated at a given point in time (i.e., these are temporal records), and (ii) the recorded readings are numeric values in a limited range. For example, a temperature sensor might only record values between -40F to 250F with one decimal point precision, while the barometric pressure module used in the Mica weather board [Polastre 2003] measures pressure in the range 300mb to 1100mb, again with one decimal point precision [Polastre 2003]. Traditional indexing methods used in relational database systems [Fagin et al. 1979; Litwin 1980] are not suitable, as these are mainly geared towards magnetic disks and do not take into account the asymmetric read/write behavior of flash media. MicroHash and MicroGF have been implemented in nesC [Gay et al. 2003] and use the TinyOS [Hill et al. 2000] operating system.

This article builds on our previous work in Zeinalipour-Yazti et al. [2005], in which we presented the design and results of our MicroHash index. In this article, we introduce several new improvements, such as an online compression algorithm and experimental evidence for efficient page read techniques. Additionally, we also develop an efficient solution to the problem of indexing 2D geographical information. Specifically, we present the design of the MicroGF index structure and experimentally demonstrate the advantages of such a structure against two popular alternatives: grid files and quadtrees.

Our contributions in this article can be summarized as following:

(1) We present the design and implementation of MicroHash, a novel index structure for supporting equality queries in environmental

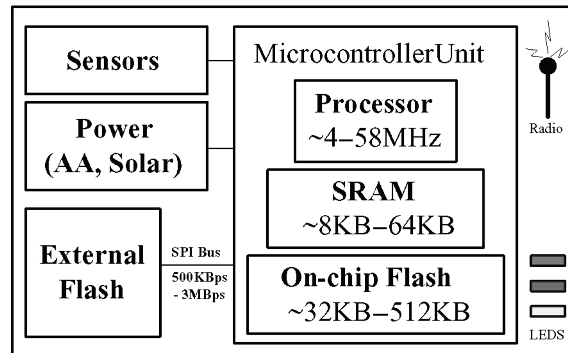

Fig. 1. The architecture of a typical wireless sensor.

sensor nodes with limited processing capabilities and a low energy budget.

(2) We propose the design and implementation of MicroGF, a novel index structure for supporting spatial queries in sensor nodes equipped with GPS capabilities.
(3) We present efficient algorithms for inserting, deleting, and searching data records stored on flash media using our algorithms.
(4) We describe the prototype implementation of MicroHash and MicroGF, and demonstrate the efficiency of our approach with an extensive experimental study using atmospheric readings from the University of Washington [ATMO 2005], the Great Duck Island study [Szewczyk et al. 2004], and geographical readings from INFATI [Jensen et al. 2005].

The remainder of the article is organized as follows: In Section 2 we present the memory hierarchy of a sensor node and a characterization of its performance characteristics using the RISE sensor. In Section 3 we formally define the indexing problem, and then describe our data structures in Section 4. Sections 5 and 6 describe the MicroHash index, search algorithms, and search optimizations, while Section 7 presents the MicroGF algorithm. Section 8 presents our experimental methodology and Section 9 the results of our evaluation. Finally, we discuss related work in Section 10 and conclude the article in Section 11.

2. THE MEMORY HIERARCHY

In this section, we briefly overview the architecture of a sensor node, with a special focus on its memory hierarchy. We also study the distinct characteristics of flash memory and address the challenges with regard to energy consumption and access time.

2.1 System Architecture

The architecture of a sensor node (see Figure 1) consists of a microcontroller unit (MCU) which is interconnected to the radio, in addition to sensors, a power source, and the LEDs. The MCU includes a processor, a static RAM (SRAM) module, and an on-chip flash memory. The processor runs at low

frequencies (≈ 4–58MHz), which reduces power consumption. SRAM is mainly used for code execution, while in the latest generation of sensors, such as Yale's 58MHz XYZ node [Lymberopoulos and Savvides 2005] and Intel's 12MHz iMote (http://www.intel.com), it can also be used for *in-memory* (or SRAM) buffering. The choice of the right energy source is application specific. Most sensors either deploy a set of AA batteries or solar panels [Sadler et al. 2004]. Therefore a sensor node might have a very long lifetime.

The on-chip flash provides a small nonvolatile storage area (32KB–512KB) for storing the executable code or for accumulating values for a small window of time [Madden et al. 2003]. A larger external storage can also be supplemented to a sensor using the *serial peripheral interface (SPI)*, which is typically found on these devices. For example, in the RISE platform, nodes feature a larger off-chip flash memory which provides the sensor with several GBs of storage.

Although it is currently not clear whether Moore's law will apply to the size and price of sensor units or their hardware characteristics, we believe that future sensor nodes will feature more SRAM and flash storage, as more complex in-network processing applications increase memory and CPU demands.

2.2 Overview of Flash Memory

Flash memory is the most prevalent storage media used in current sensor systems because of its many advantages, including: (i) nonvolatile storage, (ii) simple cell architecture, which allows easy and economical production, (iii) shock resistance, and (iv) fast read access and power efficiency. These characteristics establish flash memory as an ideal storage media for mobile and wireless devices [Dipert and Levy 1994].

There are two different types of flash memory, *NOR flash* and *NAND flash*, which are named according to the logic gate of their respective storage cells. NAND flash is the newer generation of flash memory, which is characterized by faster erase time, higher durability, and higher density. NOR is an older type of flash which is mainly used for code storage (e.g., for the BIOS). Its main advantage is that it supports writes at a byte granularity, as opposed to the page granularity used in NAND flash. NOR flash also has faster access times (i.e., ≈200ns) than NAND (50–80μs), but lacks in all other characteristics, such as density and power efficiency.

For the rest of the article, we will focus on characteristics of NAND memory, as this is the type of memory used for the on-chip and off-chip flash of most sensors, including the RISE platform. According to Micron (http://www.micron.com/), NAND memory was the fastest growing memory market in 2005 ($8.7 billion). Although reading from a NAND flash can be performed at any granularity, ranging from a single byte to a whole block (typically, 8KB–64KB), it features a number of distinct constraints, summarized as the following:

(1) *Delete Constraint:* Deleting data stored on flash memory can only be performed at block granularity (i.e., 8KB–64KB).
(2) *Write Constraint:* Writing data can only be performed at page granularity (typically, 256B–512B) after the respective page (and its respective 8KB–64KB block) has been deleted.

Table I. Performance Parameters for NAND Flash

NAND Flash installed on a Sensor Node			
	Page **Read** 1.17mA	Page **Write** 37mA	Block **Erase** 57mA
Time	6.25ms	6.25ms	2.26ms
Data Rate	82KBps	82KBps	7MBps
Energy	24μJ	763μJ	425μJ
	Flash **Idle** 0.068mA	Flash **Sleep** 0.031mA	
Time	N/A	N/A	
Data Rate	N/A	N/A	
Energy	220μJ/sec	100μJ/sec	

These parameters are for NAND Flash using a 3.3V voltage, 512B page size, and 16KB block size.

(3) *Wear Constraint:* Each page can only be written a limited number of times (typically 10,000–100,000).

The design of our index structures in the remainder of this article considers these aforementioned constraints.

2.3 Access Time of NAND Flash

Table I presents the average measurements that we obtained from a series of microbenchmarks using the RISE platform, along with a HP E3630A constant 3.3V power supply and a Fluke 112 RMS multimeter. Our first observation is that reading is three orders of magnitude less power demanding than writing. On the other hand, block erases are also quite expensive, but can be performed much faster than the former two operations. Note that read and write operations involve the transfer of data over the SPI bus, which becomes the bottleneck in the time to complete the operation. Specifically, reading and writing on flash media without utilization of the SPI bus can be achieved in ≈50μs and ≈200μs, respectively [Wu et al. 2003b]. Finally, our results are comparable to measurements reported for the MICA2 mote in Dai et al. [2004] and the XYZ sensor in Lymberopoulos and Savvides [2005].

Although these are hardware details, the application logic needs to be aware of these characteristics in order to minimize energy consumption and maximize performance. For example, the deletion of a 512B page will trigger deletion of a 16KB block on flash memory. Additionally, the MCU has to rewrite the rest of the unaffected 15.5KB. One of the objectives of our index design is to provide an abstraction which hides these hardware-specific details from the application.

2.4 Energy Consumption of NAND Flash

Another question is whether it is less expensive to write to flash memory, rather than transmitting over the RF radio. We used the RISE mote to measure the cost of transmitting the data over a 9.6Kbps radio (at 60mA), and found that transmitting 512B (one page) takes, on average, 416ms or 82,368μJ. Comparing this with the 763μJ required for writing the same amount of data to local flash, along with the fact that transmission of one byte is roughly equivalent to

executing, 1120 CPU instructions, makes local storage and processing highly desirable.

A final question we investigated is how many bytes we can store on local flash before a sensor runs out of energy. Note that this applies only to the case where the sensor runs on batteries. Double batteries (AA) used in many current designs operate at a 3V voltage and supply a current of 2,500 mAh (milliAmp-hours). Assuming, similarly to Polastre [2003], that only 2200mAh is available and that all current is used for data logging, we can calculate that AA batteries offer $23,760J$ (2200mAh * 60 * 60 * 3). With a 16KB block size and 512B page size, we would have one block delete for every 32 page writes (16KB/512B). Writing a page, according to our measurements, requires 763μJ, whereas the cost of performing a block erase is 425μJ. Therefore, writing 16KB of data requires

$$Write_{16KB} = (32pages * 763\mu J) + (425\mu J) = 24,841\mu J. \quad (1)$$

Using the result of the preceding equation, we can derive that by utilizing the $23,760J$ offered by the batteries, we can write ≈15GB before running out of batteries ((23,760J * 16KB) / 24,841μJ). The interesting point is that even in the absence of a wear-leveling mechanism, we would be able to accommodate the 15GB without exhausting the flash media. However, this would not be true if we used solar panels [Sadler et al. 2004], which provide a virtually unlimited power source for each sensor device. Another reason why we want to extend the lifetime of flash media is that the batteries of a sensor node could be replaced in cases where the devices remain physically accessible.

3. PROBLEM DEFINITION

In this section we provide a formal definition of the indexing problems that the MicroHash and MicroGF structures address. We also describe how these cope with the distinct characteristics of flash memory.

Let S denote some sensor that acquires readings from its environment every ϵ seconds (i.e., $t = 0, \epsilon, 2\epsilon, \ldots$). At each time instance t, the sensor S obtains a temporal data record $drec = \{\underline{t}, v_1, v_2, \ldots, v_x\}$, where t denotes the timestamp (key) on which the tuple was recorded, while v_i $(1 \leq i \leq x)$ represents the value of some reading (such as humidity, temperature, light, longitude and latitude, etc.).

Also, let $P = \{p_1, p_2, \ldots, p_n\}$ denote a flash media with n available pages. A page can store a finite number of bytes (denoted as p_i^{size}), which limits the capacity of P to $\sum_{i=0}^{n} p_i^{size}$. Pages are logically organized in b blocks $\{block_1, block_2, \ldots, block_b\}$, each block containing n/b consecutive pages. We assume that pages are read on a page-at-a-time basis and that each page p_i can only be deleted if its respective block (denoted as p_i^{block}) is deleted as well (write/delete constraint). Finally, due to the wear constraint, each page can only be written a limited number of times (denoted as p_i^{wc}).

The MicroHash index supports efficient *value-based equality queries* and efficient *time-based equality* and *range* queries. These queries are defined as follows.

Definition 3.1. (*Value-Based Equality Queries*). A 1D query $Q(v_i, a)$ in which the field values of attribute v_i are equivalent to value a.

For example, the query $q = (temperature, 95F)$ can be used to find time instances (ts) and other recorded readings when the temperature was 95F.

Definition 3.2. (*Time-Based Range and Equality Queries*). A 1D query $Q(t, a, b)$ in which the time attribute t is between the lower and upper bounds a and b, respectively. The equality query is a special case of the range query $Q(t, a, b)$ in which $a = b$.

For example, the query $q = (ts, 100, 110)$ can be used to find tuples recorded in the 10 second interval.

The MicroGF index supports, similarly to MicroHash, time-based equality and range queries. In addition, it supports efficient *spatial queries*, defined as follows.

Definition 3.3. (*Spatial Queries*). A multidimensional query $Q(v_1, v_2, \ldots, vx, A_{query})$ in which the spatial attributes v_1, v_2,... v_x are in the query area A_{query}.

For example, the query $q = (x, y, city\,New\ York)$ can be used to find all the positions which appeared in New York City.

Evaluating the previous queries efficiently requires that the system maintains an index structure along with the generated data. Specifically, while a node senses data from its environment (i.e., data records), it also creates index entries that point to the respective data stored on the flash media. When a node needs to evaluate some query, it uses the index records to quickly locate the desired data. Since the number of index records might be potentially very large, these are stored on the external flash, as well. Although maintaining index structures is a well-studied problem in the database community [Fagin et al. 1979; Litwin 1980; Ramakrishnan and Gehrke 2002], the low-energy budget of sensor nodes, along with the unique read, write, delete, and wear constraints of flash memory, introduce many new challenges. In order to maximize efficiency, our design objectives are as follows.

(1) *Wear-Leveling:* Spread page writes out uniformly across the storage media P in order to avoid wearing-out specific pages.
(2) *Block Erase:* Minimize the number of *random-access deletions*, as the deletion of individual pages triggers the deletion of the whole respective block.
(3) *Fast Initialization:* Minimize the size of the in-memory structures that will be required in order to use the index.

4. THE DATA STRUCTURES

In this section we describe the data structures created in the fast (but volatile) SRAM to provide an efficient way to access data stored on the persistent (but slower) flash memory. First, we describe the underlying organization of data on the flash media and then describe the involved in-memory data structures.

```
typedef struct Page {
    uint8_t  typ:3;
    uint16_t crc:16;
    uint16_t pwc:15;
    uint8_t  siz:7;
    uint32_t ppa:23;
    union {
      RootP rootP;
      DirP  dirP;
      IdxP  idxP;
      DataP dataP;};
} __attribute__((packed));

typedef struct DataP {
    DataRec records[DREC];
} __attribute__((packed));
```

```
typedef struct IdxP {
    // optional anchor
    uint64_t lastTS;
    IdxRec records[IREC];
} __attribute__((packed));

typedef struct DataRec {
    timestamp_t ts;
    data_t      val1;
} __attribute__((packed));

typedef struct IdxRec {
    fladdress_t datap;
    // optional offset
    floffset_t  offset;
} __attribute__((packed));
```

Fig. 2. Main data structures used in our nesC implementation. The example applies to the MicroHash index, while the MicroGF index uses similar structures.

4.1 Flash Organization

MicroHash and MicroGF use a sorted-by-timestamp flash organization in which records are stored on the flash media in a circular array fashion. This allows data records to be naturally sorted based on their timestamp and therefore, our organization is *sorted by timestamp*. This organization requires the least overhead in SRAM (i.e., only one data write-out page). Additionally, as we will show in Section 5.4, this organization addresses directly the delete, write, and wear constraints. When the flash media is full, we simply delete the next block following the position of the last written page. Although other organizations in relational database systems, such as *sorted* or *hashed* on some attribute, could also be used, they would have a prohibitive cost since the sensor would need to continuously update written pages (i.e., perform an expensive random page write). On the other hand, our sorted-by-timestamp organization always yields completely full data pages, as data records are consecutively packed on the flash media.

4.2 In-Memory (SRAM) Data Structures

The flash media is segmented into n pages, each with a size of 512B. Each page consists of a 8B *header* and a 504B *payload*.

Specifically the *header* includes the following fields (also illustrated in Figure 2): (i) a 3-bit *page-type (TYP) identifier*. This identifier is used to differentiate between different types of pages such as data, index, directory, and root pages; (ii) a 16-bit *cyclic redundancy checking (CRC)* polynomial on the payload, which can be used for integrity checking. When CRC is handled by lower levels, this field can be turned off; (iii) a 7-bit *number of records (SIZ)*, which identifies how many records are stored inside a page. Note that our implementation uses fixed-size records that never span to more than one page. We chose such a scheme, as opposed to using variable-length records, because records generated by a sensor always have the same size. To avoid segmentation, variable-length

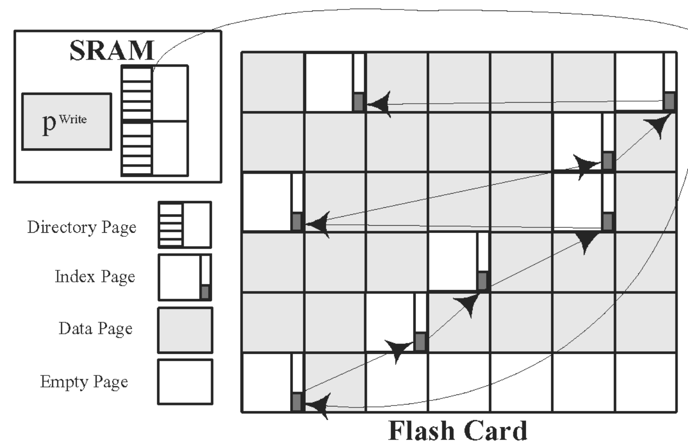

Fig. 3. Overview of the MicroHash structure. While a node senses data from its environment, it also creates index entries that point to the respective data stored on flash media. When a node needs to evaluate some query, it uses index records to quickly locate the desired data.

records would require keeping a directory inside each page, which would keep track of the available space; (iv) a 23-bit *previous page address (PPA)* stores the address of some other page on the flash media, giving in this way the capability to create linked lists on the flash; and (v) a 15-bit *page-write counter (PWC)*, which keeps the number of times a particular page has been written to flash.

While the header is identical for any type of page, the *payload* can store four different types of information: (i) *Root Page*: contains information related to the state of the flash media. For example, it contains the position of the last write (idx), the current cycle (cycle), and metainformation about the various indexes stored on flash media; (ii) *Directory Page*: contains a number of directory records (buckets), each of which contains the address of the last-known index page mapped to this bucket. In order to form larger directories, several directory pages might be chained, using the 23-bit PPA address in the header; (iii) *Index Page*: contains a fixed number of index records and the 8-byte timestamp of the last-known data record. The latter field, denoted as *anchor*, is exploited by timestamp searches which can make an informed decision on which page to follow next additionally, we evaluate two alternative index record layouts. The first, denoted as *offset* layout, maintains for each data record a respective page-id and offset, while the second layout, denoted as *nooffset*, maintains only the page-id of the respective data record; and (iv) *Data Page*: contains a fixed number of data records. For example, when the record size is 16B, then each page can contain 31 consecutively packed records.

5. INDEXING IN MICROHASH

MicroHash index is an efficient external-memory structure designed to support equality queries in sensor nodes that have limited main memory and processing capabilities. A MicroHash index structure consists of two modules (as shown in Figure 3): (i) a *directory*, and, (ii) a set of *index pages*. The *directory* consists of a set of buckets. Each bucket maintains the address of the (chronologically)

newest index page that maps to this bucket. The *index pages* contain the addresses of the data records that map to the respective bucket. Note that there might be an arbitrarily large number of data in the index pages. Therefore, these pages are stored on the flash media and fetched into main memory only when requested.

The MicroHash index is built while data is being acquired from the environment and stored on the flash media. In order to better describe our algorithm, we divide its operation in four conceptual phases: (a) the *initialization phase*, in which the root page and certain parts of the directory are loaded into SRAM; (b) the *growing phase*, in which data and index pages are sequentially inserted and organized on the flash media; (c) the *repartition phase*, in which the index directory is reorganized such that only directory buckets with the highest hit ratio remain in memory; and (d) the *deletion phase* which is triggered for garbage collection purposes.

5.1 The Initialization Phase

In the first phase, the MicroHash index locates the root page on flash media. In our current design, the root page is written on a specific page on flash (page0). If page0 is worn out, we recursively use the next available page. Therefore, a few blocks are preallocated at the beginning of the flash media for storage of root pages. The root page indicates what types of indexes are available on the system and the addresses of their respective directories. Given that an application requires the utilization of an index I, the system preloads part of $I's$ directory into SRAM (detailed discussion follows in Section 5.3). The root and directory pages then remain in SRAM for efficiency, and are periodically written out to flash.

5.2 The Growing Phase

Let us assume that a sensor generates a temporal record $drec = \{t, v_1, v_2, \ldots, v_x\}$ every ϵ seconds, where t is the timestamp on which the record was generated and v_i $(1 \leq i \leq x)$ is some distinct reading (e.g., humidity, temperature, etc.). Instead of writing $drec$ directly to flash, we use an in-memory (SRAM) buffer page p^{write} (see Figure 4(a)). When p^{write} gets full, it is flushed to the address *idx*, where *idx* denotes the address after the last page write. Note that *idx* starts out as zero and this counter is incremented by one every time a page is written out. When *idx* becomes equal to the size of the flash media n, it is reset to zero. In order to provide a mechanism for finding the relative chronological order of pages written on flash media, we also maintain the counter $cycle$, which is incremented by one every time *idx* is reset to zero. The combination of $<cycle, pageid>$ provides the chronological-order mechanism.

Next, we describe how index records are generated and stored on the flash media. The index records in our structure are generated whenever the p^{write} buffer gets full. At this point, we can safely determine the physical address of the records in p^{write} (i.e., *idx*). We create one index record $ir = [idx, offset]$ for each data record in p^{write} $(\forall drec \in p^{write})$. For example, assume that we insert the following 12byte $[timestamp, value]$ records into an empty MicroHash

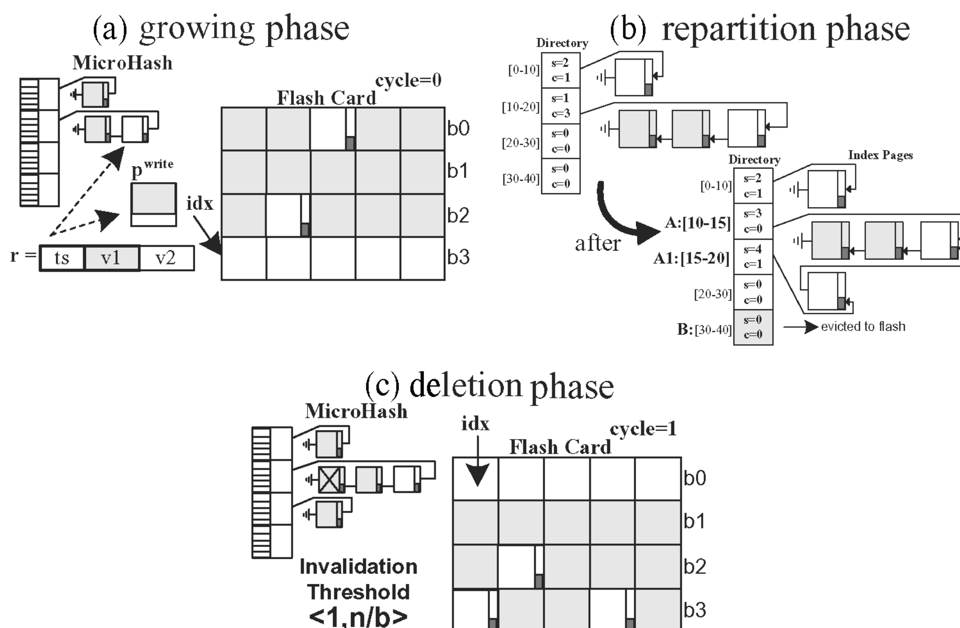

Fig. 4. The three indexing phases: (a) growing phase, (b) repartition phase and (c) deletion phase.

index: {[1000,50], [1001,52], [1002,52]}. This will trigger creation of the following index records: { [0,0],[0,12],[0,24] }. Since p^{write} is written to address idx, the index records always reference data records that have a smaller <cycle,pageid> identifier.

The MicroHash directory provides the start address of the index pages. It is constructed by providing the following three parameters: (a) a lower bound (lb) on the indexed attribute; (b) an upper bound (ub) on the indexed attribute; and (c) the number of available buckets (note that we can only fit a certain number of directory buckets in memory). For example, assume that we index temperature readings which are only collected in the following known and discrete range [−40..250], then we set $lb = -40F$, $ub = 250F$, and $c = 100$. Initially, each bucket represents exactly $\frac{|lb..ub|}{c}$ consecutive values, although this equal splitting (which we call *equiwidth splitting*) is refined in the repartition phase based on the data values collected at runtime.

5.3 The Repartition Phase

A drawback of the initial equiwidth bucket splitting approach is that some buckets may rarely be used, while others may create long lists of index records. To overcome this problem, we use the following splitting policy: Whenever a directory bucket A links to more than τ records (user parameter), we evict to flash the bucket B, which was not used for the longest period of time (see Figure 4(b)). Note that this mechanism can be implemented using only two counters per bucket (one for the timestamp and one for the number of records). In addition to the eviction of page B, we also create a new bucket $A1$. Our

objective is to provide a finer granularity to the entries in A, as this bucket is the most congested. Note that the values in A are not reassigned between A and $A1$ as would happen in dynamic hashing techniques such as *extendible hashing* [Fagin et al. 1979] or *linear hashing* [Litwin 1980]. The reason is that the index pages are on flash media and updating these pages would result in a potentially very large number of random updates (which would be extremely expensive). Our *equidepth*, rather than equiwidth, bucket splitting approach keeps in memory finer intervals for index records used more frequently.

Figure 4(b) shows that each bucket is associated with a counter s (which indicates the timestamp of the last time the buffer was used), and a counter c (which indicates the number of index records added since the last split). In the example, the $c = 3$ value in bucket 2 (A:[10-20]) exceeds the $\tau = 2$ threshold, and therefore, this bucket has to be split. Before splitting the bucket, the index forces bucket 4 (B: [30-40]) to the flash media (as this is the least-used bucket). It then proceeds with the split into A:[10-15] and A1:[15-20]. Note that the A list now contains values in [10–20], while the $A1$ list contains only values in the range [15-20]. Any future additions to A, however, will only include values in the range [10-15]. The idea is that we don't want to reassign the values of A, since these values reside on the flash media. In Section 6, we will show that this organization preserves efficient data access.

5.4 The Deletion Phase

In this phase, the index performs a garbage collection operation of the flash media in order to make space for any newly acquired data. The phase is triggered after all n pages have been written to flash media. This operation blindly deletes the next n/b pages, which is the whole block following the pointer *idx* (see Figure 4(c)). It is then triggered again whenever n/b pages have been written, where b is the number of blocks on the flash media. This leaves the index with n/b clean pages that can be used for future writes. Note that this might leave pointers from index pages referencing data which has already been deleted. This problem is handled by our search algorithm, described in the next section.

The distinctive characteristic of our garbage collection operation is that it satisfies directly the delete constraint because pages are deleted in blocks (which is less expensive than deleting a page-at-a-time). This makes it different from similar operations of flash file systems [Dai et al. 2004; Woodhouse 2006] that perform page-at-a-time deletions. Additionally, this mode provides the capability to "blindly" delete the next block, without the need to read or relocate any deleted data. The correctness of this operation is established by the fact that the index records always reference data records that have a smaller <cycle,pageid> identifier. Thus, when an index page is deleted, we are sure that all associated data pages are already deleted.

6. SEARCHING IN MICROHASH

In this section we show how records can efficiently be located by their values or timestamps.

6.1 Searching by Value

The first problem we consider is how to perform value-based equality queries. Finding records by their value involves: (a) locating the appropriate directory bucket from which the system can extract the address of the last index page, (b) reading the respective index pages on a page-by-page basis, and (c) reading the data records referred by index pages on a page-by-page basis. Since SRAM is extremely limited on a sensor node, we adopt a *record-at-a-time* query return mechanism in which records are reported to the caller on record-by-record basis. This mode of operation requires three available pages in SRAM: one for the directory (dirP) and two for the reading pages (idxP,dataP), which only occupies 1.5KB. If more SRAM was available, the results could have been returned at other granularities, as well. The complete search procedure is summarized in Algorithm 1.

Algorithm 1 EqualitySearch

```
Input: value: the query (search predicate).
Output: The records that contains value.
 1: Procedure EqualitySearch(value)
 2:    bucket = hash(value);
 3:    address = dirP[bucket].idxP;
 4:    while((idxP = loadPage(address)) != NULL)
 5:        for i = 0 to |idxP.size|
 6:            If ((dataP=loadPage(idxP[i].dataP))==NULL)
 7:              address=0; break;
 8:            If (dataP.record[idxP[i].offset]==value)
 9:              signal dataP.record[idxP[i].offset];
10:        end for
11:        address = idxP.ppa;
12:  end while
13:  signal finished;
14: end procedure
```

Note that the loadPage procedure in lines 4 and 6 returns NULL if the fetched page is not in valid chronological order (with respect to its preceding page) or if the fetched data records are not within the specified bucket range. We use these termination conditions, as the index records might point to deleted data pages. Recall that we do not update the index records during deletions for performance reasons. The validations applied by loadPage ensure that we can safely terminate the search procedure. Finally, since the MicroHash index returns records on a record-at-a-time basis, we use a *signal finished* at the end to indicate that the search procedure has been completed.

6.2 Searching by Timestamp

In this section we investigate time-based equality and range queries. First, note that if index pages were stored in a separate physical location, and thus not interleaved with data pages, the sorted (by timestamp) file organization would

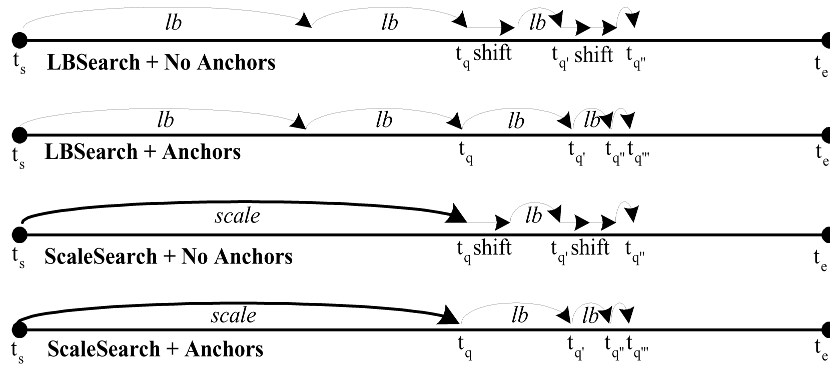

Fig. 5. Searching by timestamp, where t_s: oldest timestamp on flash (t_e: newest), t_q: the query (timestamp), lb: the lower bound obtained using either idx_{lb} or idx_{scaled}.

allow us to access any data record in $O(1)$ time. However, this would also violate our wear-leveling mechanism, as we wouldn't be able to spread out the page writes uniformly among data and index pages. Another approach would be to deploy an in-memory address translation table such as that one used in Wu et al. [2003a; 2003b], which would hide details of the wear-leveling mechanism. However, such a structure might be too big, given the memory constraints of a sensor node, and would also delay the sensor boot time.

Efficient search can be supported by a number of different techniques. One popular technique is to perform a binary search over all pages stored on the flash media. This would allow us to search in $O(logn)$ time, where n is the size of the media. For large values of n, such a strategy is still expensive. For instance, with 512MB flash media and a page size of 512B, we would need approximately 20 page reads before we could find the expected record.

In our approach, we investigate two binary search variants, named *LBSearch* and *ScaleSearch*. LBSearch starts by setting a pessimistic lower bound on which page to examine next, and then recursively refines the lower bound until the requested page is found. ScaleSearch, on the other hand, exploits knowledge about the underlying distribution of data and index pages in order to offer a more aggressive search method that usually executes faster. ScaleSearch is superior to LBSearch when data and index pages are roughly uniformly distributed on the flash media, but its performance deteriorates for skewed distributions.

For the remainder of this section, we assume that a sensor S maintains locally some indexed readings for the interval $[t_a..t_b]$. Also, let $x < y$ (and $x > y$) denote that the $<cycle_x, idx_x>$ pair of x is smaller (and respectively greater) than the $<cycle_y, idx_y>$ of y. When S is asked for a record with the timestamp t_q, it follows one of the following approaches: (i) LBSearch: S starts out by setting the lower bound

$$idx_{lb}(t_q, t_s) \begin{cases} \left\lceil \frac{t_q - t_s}{\Re} \right\rceil, & \text{if } cycle == 0; \\ idx + \left\lceil \frac{t_q - t_s}{\Re} \right\rceil & \text{otherwise,} \end{cases}$$

where *idx* is the address of the last written page and $\Re$ a constant indicating the maximum number of data records per page. It then deploys the LBSearch (ts, idx_{lb}) procedure, as illustrated in Algorithm 2. It is easy to see that in each recursion step, LBSearch always moves clockwise (increasing time order) and that $idx_{lb} \leq idx_{t_q}$.

Algorithm 2 LBSearch (No Anchors)

Input: t_q: the query (timestamp), *current*: begin search address
Output: The page that contains t_q.

```
1:  Procedure LBSearch(t_q, current)
2:    p = readPage(current);
3:    if (isIndexPage(p))
4:        // logical right shift
5:        return LBSearch(t_q, current + 1);
6:    else
7:        t_1 = P.record[0].ts;
8:        t_2 = P.record[P.lbu].ts;
9:        if (t_1 ≤ t_q ≤ t_2)
10:           return P;
11:       end if
12:       return LBSearch(t_q, current + idx_lb(t_q, t_2));
13:   end if
14: end procedure
```

It is important to note that a lower bound can only be estimated if the fetched page, on each step of the recursion, contains a timestamp value. Our discussion so far assumes that the only pages that carry a timestamp are data pages which contain a sequence of data records $\{[ts_1, val_1] \ldots [ts_1, val_{\Re}]\}$. In such a case, the LBSearch has to *shift* right until a data page is located. In our experiments, we noted that this deficiency could add, in some cases, three to four additional page reads. In order to correct the problem, we store the last known timestamp inside each index page (named *Anchor*).

(ii) ScaleSearch: When index pages are uniformly spread out across the flash media, then a more aggressive search strategy might be more effective. In ScaleSearch, which is the technique we deployed in MicroHash, instead of using idx_{lb} in the first step, we use idx_{scaled}:

$$idx_{scaled}(t_q, t_s) \begin{cases} \left\lceil \frac{t_q - t_a}{t_b - t_a} * idx \right\rceil, & \text{if } cycle == 0; \\ idx + \left\lceil \frac{t_q - t_a}{t_b - t_a} * n \right\rceil, & \text{otherwise.} \end{cases}$$

We then use LBSearch in order to refine the search. Note that idx_{scaled} might in fact be larger than idx_{t_q}, in which case LBSearch might need to move counter clockwise (in decreasing time order).

Performing a range query by timestamp $Q(t_q, a, b)$ is a simple extension of the equality search. More specifically, we first perform a ScaleSearch for the upper bound b (i.e., $Q(t_q, b)$) and then sequentially read backwards until a is found. Note that data pages are chained in reverse chronological order (i.e.,

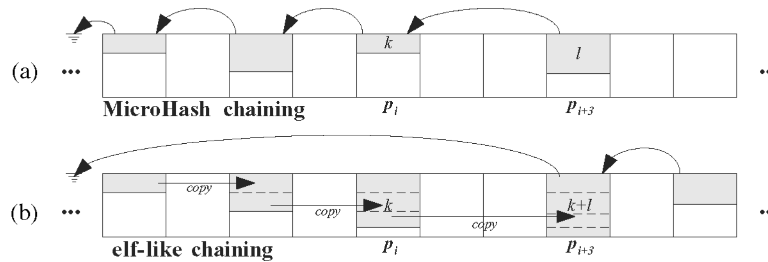

Fig. 6. Index chaining methods: (a) MicroHash chaining and (b) elf-like chaining.

each data page maintains the address of the previous data page) and therefore this operation is very simple.

6.3 Search Optimizations

In this section we present three optimizations that increase the performance of the basic MicroHash approach. The first two methods alleviate the performance penalty that incurs because of index pages which are not fully occupied. Note that searching over partially full index pages results in an unnecessary transfer of data between the MCU and flash cells. The first method, named *elf-like chaining* (*ELC*), eliminates nonfull index pages, which as a result decreases the number of pages required to answer a query. The second method, named *two-phase read*, minimizes the number of bytes transferred from the flash media. The third method attempts to minimize the amount of data that is read or written to the flash media. This is achieved by deploying some basic run-length encoding compression scheme while the sensor acquires the data.

6.3.1 *Elf-Like Chaining* (*ELC*). In MicroHash, index pages are chained using a back-pointer, as illustrated in Figure 6. This method is named *MicroHash chaining*. Inspired from the update policy of the ELF filesystem [Dai et al. 2004], we also investigate, and later experimentally evaluate, the *elf-like chaining* (*ELC*) mechanism. The objective of ELC is to create a linked list in which each node, other than the last node, is completely full. This is achieved by copying the last nonfull index page into a newer page whenever new index records are added to the index. This continues until an index page becomes full, at which point it is not further updated.

To better understand the two techniques, consider the following scenario (see Figure 6): An index page on flash (denoted as p_i $(i \leq n)$) contains k $(k < p_i^{size})$ index records $\{ir_1, ir_2, \ldots, ir_k\}$ that in our scenario map to some directory bucket v. Suppose that we create a new data page on flash at position p_{i+1}. This triggers the creation of l additional index records, which in our scenario map to the same bucket v. In MicroHash chaining (MHC), the buffer manager simply allocates a new index page for v and keeps the sequence $\{ir_1, ir_2, \ldots, ir_l\}$ in memory until the LRU replacement policy forces the page to be written out. If we assume that the new index sequence is forced out of memory at p_{i+3}, then p_i will be back-pointed by p_{i+3}, as shown in Figure 6. In elf-like chaining (ELC), the buffer manager reads p_i in memory and then augments it with the l new index records

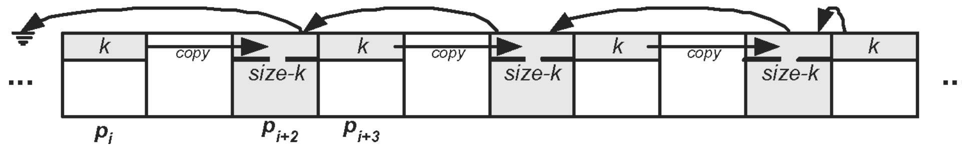

Fig. 7. Sequential trashing in ELC.

(i.e., $\{ir_1, \ldots, ir_k, \ldots, ir_{l+k}\}$). However, p_i is not updated due to the write and wear constraints. Instead, the buffer manager writes the new $l + k$ sequence to the end of flash media (i.e., at p_{i+3}). Note that p_i is now not back-pointed by any other page and will not be utilized until the block delete, guided by the *idx* pointer, erases it.

The optimal compaction degree of index pages in ELC significantly improves the search performance of an index, as it is not required to iterate over partially full index pages. However, in the worse case, ELC might introduce an additional page read per indexed data record. Additionally, we observed in our experiments (presented in Section 9) that ELC requires, on average, 15% more space than typical MicroHash chaining. In the worst case, the space requirement of ELC might double the requirement of MHC.

To understand the worst-case scenario in ELC, consider the scenario in Figure 7. This time, assume that the buffer manager reads p_i in memory and then augments p_i^{size} (a full page) new index records. This will evict p_i to some new address (in our scenario p_{i+2}). However, some additional k records are still in the buffer. Assume that these pages are now evicted from main memory to some new flash position (in our scenario p_{i+3}). So far, we utilized three pages (p_i, p_{i+2} and p_{i+3}), while the index records could fit into only two index pages (i.e., $k + p_i^{size}$ records, $k < p_i^{size}$). When the same scenario is repeated, we say that ELC suffers from *sequential trashing* and ELC will require double the required space to accommodate all index records.

6.3.2 *Two-Phase Page Reads.* Our discussion so far assumes that pages are read from the flash media on a page-by-page basis (usually, 512B per page). When pages are not fully occupied, such as index pages, then many empty bytes (padding) is transferred from the flash media to main memory. In order to alleviate this burden, we exploit the fact that reading from flash can be performed at any granularity (i.e., as small as a single byte). More specifically, we propose the deployment of a *two-phase page read* in which the MCU reads a fixed header of a page from flash in the first phase, and then reads the exact amount of bytes in the next.

The performance of two versus single-phase reads has been experimentally evaluated on the RISE platform [Neema et al. 2005], as is shown in Figure 8. Note that in order to initiate a read over the flash media, there is a fixed 9-byte overhead for accessing the SPI bus. From our experimental analysis, it can be concluded that two-phase reading is almost always superior to its single-phase counterpart, with the exception of pages which are adequately full (i.e., >90%). Minimizing the page reading time significantly minimizes energy consumption during searches.

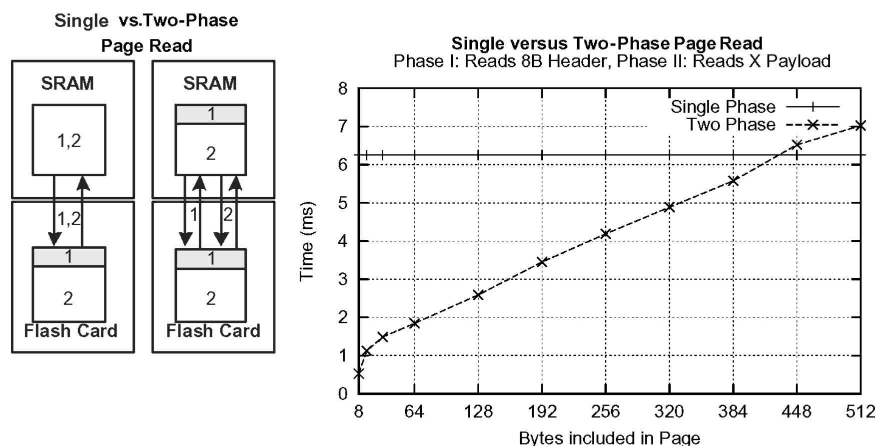

Fig. 8. (a) Illustration of the single-phase and two-phase page read strategies; (b) performance comparison of strategies on the RISE platform.

6.3.3 *Lossless Compression by Exploiting Temporal Locality*. One common characteristic of the real signals (or records) generated by sensing devices is that consecutive time instances are correlated [Deligiannakis et al. 2004; Tang and Raghavendra 2004; Szewczyk et al. 2004]. For instance, a sensor device might generate the same temperature reading of $70F$ for every second in the span of an hour. Therefore, the temperature time series is characterized by temporal locality.

In this subsection, we describe how a sensor device might exploit this additional parameter in order to perform some basic compression of the generated data values. Although we might argue that compression is of less importance, since flash storage can be very large and potentially very economical, its importance originates from the fact that the predominant cost in the operation of a sensing device (that records readings on local storage) comes from the bytes that are read or written to flash media. Therefore, we seek to reduce the amount of data stored on the flash media by using some energy-efficient compression algorithm that might significantly prolong the lifetime of the sensing device.

Before outlining our solution, it is important to mention that any such technique has to operate in an *online* fashion: The compression must be performed before the data is stored on flash media and while being generated by the sensing device. With this mode of operation, we avoid the only other alternative, which is expensive offline compression method in which the data is compressed in a postprocessing step. Finally, we seek to preserve the temporal order of data records in order to keep the searching procedures, outlined in Section 6, unaffected.

We propose the deployment of tools from the field of information theory in order to address the online compression problem. Specifically, we utilize the *run-length encoding* scheme in order to eliminate the repetitive sequences that are a result of temporal locality. In run-length encoding, consecutive values are replaced by a single value of the repeated value. For example the sequence {50,60,60,60,60,60,61,61} can be represented with {1:50,5:60,2:62}, where x :

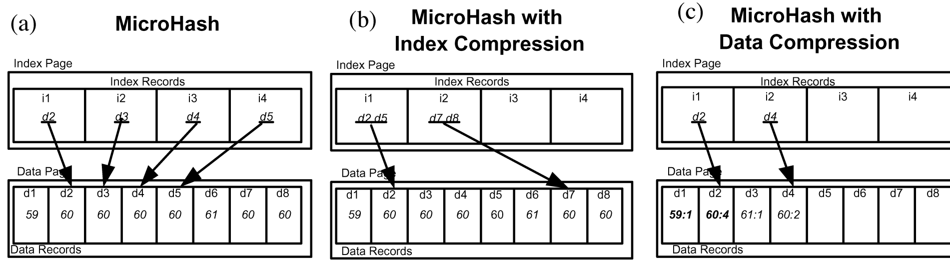

Fig. 9. Exploiting temporal locality in order to compress MicroHash records.

y denotes that value y is repeated for x consecutive time instances. In the example, this yields a savings of two integers.

We examine how the encoding scheme can be applied in two different cases: *(i) index record compression* and *(ii) data record compression*. In the former case, illustrated in Figure 9(b), we identify the correlated intervals and represent them using a single index record. Recall that index records are generated as soon as the in-memory (SRAM) buffer page p^{write}, that contains the data records, gets full. Therefore the correlated intervals are identified only within p^{write}. In this example, we can see that each index record now stores the ranges $[d_2, d_5]$ and $[d_7, d_8]$, in which the temperature 60F was recorded, rather than one index record per data record (illustrated in Figure 9(a)). In the data record compression case, in Figure 9(c), we inverse the situation and record ranges of data values, instead of ranges of index values. The compression is applied incrementally and directly on the data series, which at this point resides in p^{write}. Specifically, every time p^{write} gets full, we execute in time $O(k)$ the run-length encoding technique on the k uncompressed data records of p^{write}, where $k \leq n$ and $n = |p^{write}|$. When k equals zero, then this iterative procedure terminates, the respective index records are generated (using the typical MicroHash algorithm), and p^{write} is written to flash.

Although data record compression might be more storage efficient than index record compression, the former can only be applied on 1D data records (i.e., (timestamp, val_1) pairs). The N-dimensional case, where each data record has the following format: r = (timestamp, $val_1, val_2, \ldots, val_n$), turns out to be more complicated, as the run-length encoding is designated to only run on a single correlated attribute rather than N attributes.

Finally, we mention that it would not be practical to apply any compression method directly on the binary sequence of records, as this prevents us from having direct access to each data record. For instance, suppose that we are given the following two sensor readings in their binary representation ([ts,val]): {[00000,00000], [00001,00000]}. Using run-length encoding, we could have encoded this sequence using three numbers: {14, 1, 5} (14 consecutive "0", 1 consecutive "1", and 5 consecutive "0"). However, such an encoding would not allow us to have direct access to the individual timestamps or values of the two data records, unless the data is first decoded.

7. INDEXING AND SEARCHING IN MICROGF

In spatial query processing, the objective is to locate the data records which were recorded when the sensing device was within some predicate geographic region. For instance, a sensor device equipped with a GPS might provide the geographic coordinates of a moving zebra [Sadler et al. 2004] or car [Jensen et al. 2005]. A query might then be to locate the objects that were close to some predicate landmark.

A naive method to cope with this kind of spatial query is to apply the MicroHash index directly on two (or more) spatial dimensions. This would construct one index record for each respective spatial dimension. For example, data record $< ts, x, y >$ can be indexed by keeping two different index records for x and y, respectively. However, such an index structure is not very efficient, firstly due to redundant space allocation, as we create multiple index records for each data record, and secondly because such a division cannot efficiently capture the locality of spatial information for both equality and range queries.

In this section we propose the MicroGF (*micro grid file*) index structure, which is an external memory index structure designated for the efficient execution of spatial queries. The MicroGF index uses, similarly to MicroHash, a set of directory and index pages to annotate where spatial data records are located on the flash memory. The directory consists of a set of buckets which segments the recording space into multiple cells. Each bucket maintains the address of the (chronologically) newest index page that maps to the corresponding cell. Index pages contain addresses of the data records that map to the respective bucket.

7.1 MicroGF Index Data Structures

Directory page data structure: The directory of the MicroGF index structure consists of an N-dimensional *grid* of *cells*, where N denotes the number of spatial dimensions in the data records. To simplify discussion, we assume a 2D grid of cells in the xy-plane, which represents the coordinate space of some geographic area in which the sensing device generates spatial records. Therefore, the MicroGF directory page consists of n*n square cells, where n is the size of some geographic area, and where each cell contains the address of the last-known index page that mapped into this geographic region.

Index page data structure: The index pages contain, similarly to MicroHash, the addresses of the respective data pages. In order to improve search performance, each index record is divided into four equal-size quadrants. This essentially divides the recording space of a grid cell into four regions. Each quadrant then maintains the address for K records which map to the specific area.

7.2 Indexing in MicroGF

The indexing procedure is triggered when the in-memory data record buffer p^{write} gets full. For each spatial record R, we follow the following insertion algorithm (see Algorithm 3). First, we locate the correct grid cell to which data record R has to be assigned. This information is encapsulated in MicroGF's directory. From there, we extract the flash address of the (chronologically) newest

index page (Idx_P) that maps to the corresponding cell.

We then utilize the *find_quadrant*(Idx_R, x, y) function, which identifies the quadrant Q to which the data record R maps. If this quadrant has enough

Algorithm 3 Insertion

Input: MicroGF directory, data record $< ts, x, y >$. **Output:** updated MicroGF with $< ts, x, y >$ inserted.

```
1:  Procedure Insert(ts, x, y)
2:    Dir_P = the directory entry that x, y resides in;
3:    Idx_P = Dir_P.idx, Idx_R = the first index record in Idx_P;
4:    Q = find_quadrant(Idx_R, x, y);
5:    if(Quadrant_Insert(Q, x, y) == success) signal finished;
6:    else
7:      Q.BID = find_borrow_quadrant(Q)
8:      if(Quadrant_Insert(Q.BID, x, y) == success) signal finished;
9:      else
10:       ifIdx_P is not full
11:         Idx_R' = new index record, Q.SID = Idx_R',
12:         Q' = find_quadrant(Idx_R', x, y), Quadrant_Insert(Q', x, y);
13:       else
14:         Idx_P' = new index page, Idx_P' → next = Idx_P,
15:         Dir_P.idx = Idx_P', INSERT(ts, x, y);
16:       end if
17:     end if
18:   end if
19:   signal finished;
20: end procedure
```

space to accommodate R (i.e., if the number of indexed records is less than K), then the insertion is completed. Otherwise, the record R is assigned to adjacent quadrants having no records in it. These adjacent quadrants are named, for convenience, *borrow quadrants*. Note that borrow quadrants are allocated only within the same index record. If the number of records assigned to a cell does not fit into the adjacent borrow quadrants, then a new index record (Idx_R') is created. In this scenario, the original quadrant Q is repartitioned into four subquadrants, and the new index record is identified by a subregion address (SID).

To better understand the indexing process, assume that we have the trajectory shown in Figure 10 (left). The trajectory consists of seven data points, denoted with numbers 1 to 7. Also assume that the index record filling parameter K is set to 2. The first two records, 1 and 2, are inserted into quadrants 01 and 00 separately (Figure 10 (right)). Records 3 and 4 are then both inserted into quadrant 10. As records 5 and 6 belong to quadrant 10 (but quadrant 10 is full), we check all 4 quadrants and find an empty quadrant 11, then we borrow the quadrant of 11 to index records 5 and 6 and update the borrow quadrant identifier (BID) of quadrant 10 as 11. When data record 7 comes, there is no space in either the mapping quadrant (10) or borrow quadrant (11). Therefore, we create a new index record which accommodates the new record. Note that the BID of quadrant 10 is 11, which means we utilize 11 as the

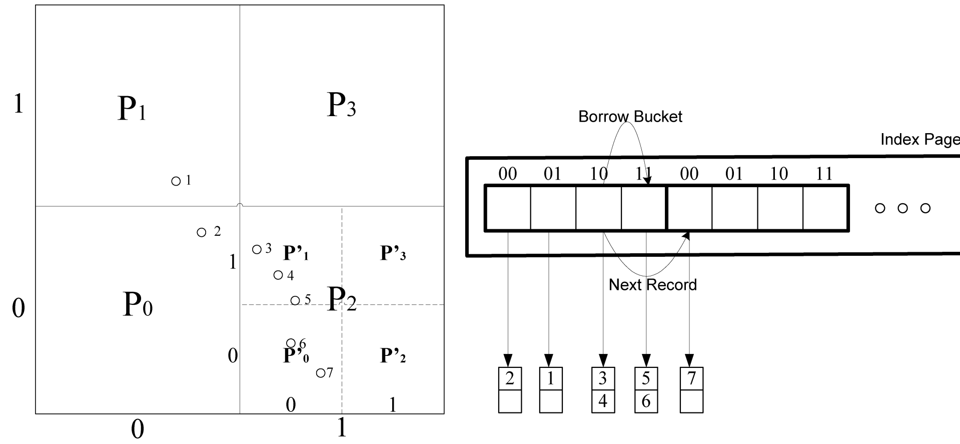

Fig. 10. Creation of a MicroGF index page. The two figures show at a high level how the trajectory on the left is encoded into a set of two index records.

borrow quadrant in which to store the data records mapped into quadrant 10. In addition, the subregion identifier (SID) of 10 in the first index record is 2, which means that there is no more space to index data record 7 in the first index record, and so we use the second index record to index it with smaller granularity.

7.3 Searching in MicroGF

Searching in MicroGF is performed by first finding the directory cell Dir_P to which the query A belongs to. Then, all the index pages that map to Dir_P are searched. For each index page, we only have to check the quadrant Q with which A overlaps, the borrow quadrant of Q ($Q.BID$), and Q's subregion ($Q.SID$). The pseudocode of the MicroGF searching procedure is detailed in Algorithm 4.

Algorithm 4 Query

Input: MicroGF directory, query area A.
Output: data records $< ts, Px, Py >$ within area A.

```
1:  Procedure Query(A)
2:    Dir_P = the directory entry that A overlaps;
3:    Idx_P = Dir_P.idx;
4:    while(Idx_P != NULL)
5:      Q = find_quadrant(Idx_P, A);
6:      while(Q != NULL)
7:        search(Q, A), search(Q.BID, A);
8:        Q = Q.SID;
9:      end while
10:     Idx_P = Idx_P → next;
11:   end while
12:   signal finished;
13: end procedure
```

7.4 The Advantages of MicroGF

Two other popular methods to index spatial records are grid files [Nievergelt et al. 1984] and quadtrees [Samet 1984]. In grid files, a grid directory is utilized to partition space into rectanglular partitions and to index the data page containing the desired data records. However, the directory in grid files is very large and thus cannot be maintained in SRAM, as in MicroGF. In quadtree, the recording space is recursively decomposed into quadrants. Each of the four quadrants becomes a node in the quadtree. A larger quadrant is a node at a higher hierarchical level of the quadtree, and smaller quadrants appear at lower levels. A problem with quadtree is its poor index space utilization for certain biased data distributions. For example, if all the data records are located under only one quadrant at each node, then only 25% of the index space is utilized. MicroGF, however, overcomes this problem by introducing the concept of borrow quadrants, which results in better space utilization of the index pages for any dataset. With more compact index pages, query processing in MicroGF is faster and more economical. These advantages of the MicroGF index structure are experimentally validated in Section 9.5.

8. EXPERIMENTAL METHODOLOGY

In this section we describe the details of our experimental methodology.

8.1 Experimental Testbed

We have implemented MicroHash along with a tiny LRU BufferManager in nesC [Gay et al. 2003], the programming language of TinyOS [Hill et al. 2000]. TinyOS is an open-source operating system designed for wireless embedded sensor nodes. It was initially developed at UC-Berkeley and has been deployed successfully on a wide range of sensors, including our RISE mote. TinyOS uses a component-based architecture that enables programmers to wire together, in an on-demand basis, the minimum required components. This minimizes the final code size and energy consumption as sensor nodes are extremely power- and memory-limited. Moreover, nesC [Gay et al. 2003], the programming language of TinyOS, realizes the operating system structuring concepts and execution model.

Our implementation consists of approximately 5,000 lines of code and requires at least 3KB in SRAM. Specifically, we use one page as a write buffer, two pages for reading (i.e., one for an index page and one for a data page), one page as an indexing buffer, one for the directory, and one final page for the root page. In order to increase insertion performance and index page compactness, we also supplement additional index buffers (i.e., 2.5KB–5KB).

We had to write a library that simulates the flash media using an operating system file in order to run our code in TOSSIM [Levis et al. 2003], the simulation environment of TinyOS. We additionally wrote a library that intercepts all messages communicated from TinyOS to the flash library and prints out various statistics, and one final library that visualizes the flash media using bitmap representations.

8.2 PowerTOSSIM—Energy Modeling

PowerTOSSIM is a power modeling extension to TOSSIM presented in Shnayder et al. [2004]. In order to simulate the energy behavior of the RISE sensor, we extended PowerTOSSIM and added annotations to the MicroHash structure that accurately provide information when the power states change in our environment. We have focused our attention on precisely capturing flash performance characteristics, as opposed to capturing the precise performance of other, less frequently used modules (e.g., the radio stack).

Our power model follows our detailed measurements of the RISE platform [Neema et al. 2005], which are summarized as the following: We use a 14.8 MHz 8051 core operating at 3.3V with a current consumption of 14.8mA (on), 8.2mA (idle), 0.2μA (off). We utilize a 128MB flash media, unless otherwise mentioned, with a page size of 512B and a block size of 16KB. The current to read, write, and block delete was 1.17mA, 37mA, 57μA, respectively, and the time to read in the three aforementioned states was 6.25ms, 6.25ms, and 2.27ms.

Using these parameters, we performed an extensive empirical evaluation of our power model and found that PowerTOSSIM is indeed a very useful and quite accurate tool for modeling energy in a simulation environment. For example, we measured the energy required to store 1MB of raw data on an RISE mote and found that this operation requires $1526mJ$, while the same operation in our simulation environment returned $1459mJ$, which has an error of only 5%.

8.3 Dataset Descriptions

Since we cannot measure environmental conditions such as temperature or humidity in a simulation environment, we adopt a trace-driven experimental methodology in which a real dataset is fed into the TOSSIM simulator. More specifically, we use the following datasets:

—*Washington State Climate:* This is a a real dataset of atmospheric data collected by the Department of Atmospheric Sciences at the University of Washington [ATMO 2005]. Our 268MB dataset contains readings on a per minute basis between January 2000 and February 2005. The readings, which are recorded at a weather logging station in Washington, include barometric pressure, wind speed, relative humidity, cumulative rain, and others. Since many of these readings are not typically measured by wireless sensor nodes, we only index the temperature and pressure readings, and use the rest as part of the data included in a record. Note that this is a realistic assumption, as sensor nodes may concurrently measure a number of different parameters. Figure 11 shows the time series for the readings used in our experiments, along with the respective value distributions.

—*Great Duck Island (GDI 2002):* A real dataset from the habitat monitoring project on Great Duck Island in Maine.[1] We use readings from one of the 32 nodes that were used in the Spring 2002 deployment, which included the following readings: light, temperature, thermopile, thermistor, humidity, and voltage.

[1]http://www.greatduckisland.net/

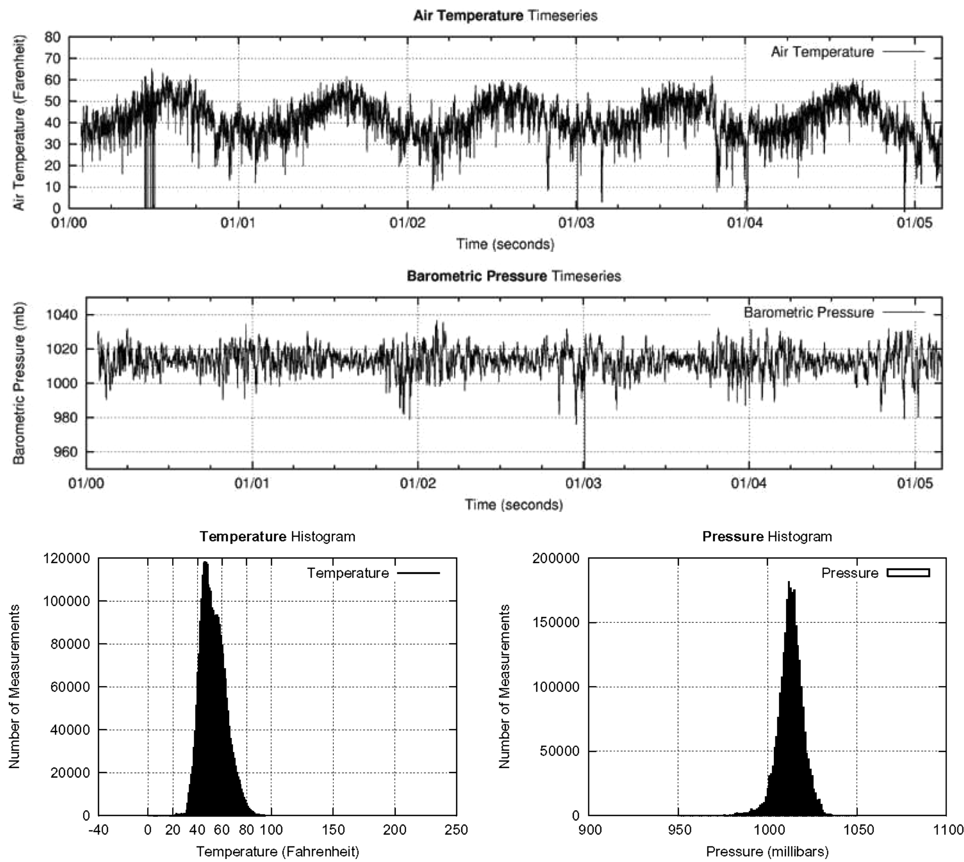

Fig. 11. Temperature (F) and barometric pressure (mb) readings recorded at an atmospheric monitoring site in Washington. The last row indicates the respective distribution histograms for the two time series.

Our dataset includes approximately 97,000 readings recorded between October and November, 2002.

—*INFATI:* This is a real dataset derived from the INFATI Project [Jensen et al. 2005] carried out by Aalborg University. The readings are the GPS positions of 24 different cars moving in the city of Aalborg, Denmark in 2001. The readings include car-id, timestamp, x-coordinate, y-coordinate, etc. Our dataset includes approximately 250,000 readings recorded between January and March, 2001.

9. EXPERIMENTAL EVALUATION

In this section we present extensive experiments to demonstrate the performance effectiveness of the MicroHash index structure. The experimental evaluation described in this section focuses on three parameters: (i) *space overhead* of maintaining the additional index pages; (ii) *search performance*, which is defined as the average number of pages accessed for finding the required record; and (iii) *energy consumption* for indexing the data records. Due to the design of

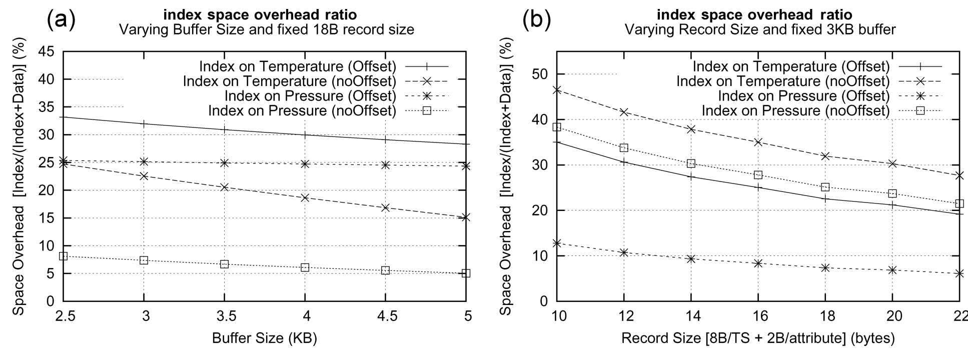

Fig. 12. Space overhead of index pages with (a) varying buffer size, and (b) varying record size.

the MicroHash and MicroGF indices, each page is written exactly once during a cycle. Therefore, there is no need to experimentally evaluate the wear-leveling performance. Finally, we compare the performance of MicroGF with other spatial indexing techniques.

9.1 Overhead of Index Pages

In the first series of experiments, we investigate the overhead of maintaining additional index pages on the flash media. For this reason, we define the overhead ratio Φ as follows: $\Phi = \frac{IndexPages}{DataPages+IndexPages}$. We investigate the parameter Φ using: (a) an increasing buffer size, and (b) an increasing data record size.

We also evaluate two different index record layouts: (a) *Offset*, in which an index record has the form {pageid,offset}, and *NoOffset*, in which an index record has the form {pageid}. We use the five-year time series from the Washington State climate dataset and we index data records based on their temperature and pressure attributes. The data record on each of the 2.9M time instances was 18 bytes (i.e., 8B timestamp + 5x2B readings).

9.1.1 *Increasing Buffer Size.* Figure 12 (left) presents our results using a varying buffer size. The figure shows that in all cases, a larger buffer helps in fitting more index records per page, which therefore also linearly reduces the overall space overhead. In both the pressure and temperature cases, the *NoOffset* index record layout significantly reduces the space overhead, as less information is required to be stored inside an index record.

The figure shows that indexing on pressure achieves a lower overhead. This is attributed to the fact that pressure changes slower than temperature over time. This leads to fewer evictions of index pages during the indexing phase, which consequently also increases the index page occupancy.

We found that a 3KB buffer suffices to achieve an occupancy of 75–80% in index pages. This can be viewed in the bitmap illustrations of the flash media in Figure 13. The figures show two characteristics: (a) the number of index pages on flash, and (b) the occupancy of these pages using a 256-bit grayscale pixel (where black denotes an empty page). As we can see, providing a larger buffer during the indexing phase not only decreases the number of index pages

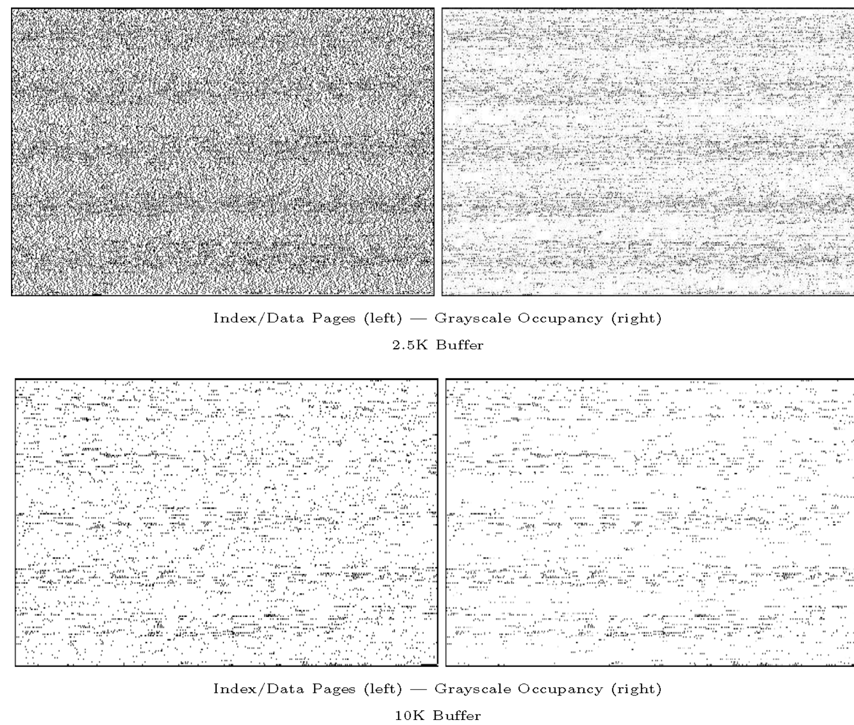

Fig. 13. Bitmap illustrations of the flash media. Each pixel represents a page on flash media. The left column shows the index pages (in black) and data pages (in white). The right column uses a 256-bit grayscale image to color the fullness of index pages (black=empty).

(i.e., less black pixels), on flash, but also makes these index pages more highly occupied (i.e., less darker pixels).

9.1.2 *Increasing Data Record Size.* Sensor nodes usually deploy a wide array of sensors, such as a photo sensor, magnetometer, accelerometer, and others. Hence, the data record size on each time instance might be larger than the minimum 10B size (8B timestamp and 2B data value). Figure 12 (right) presents our results using a varying data record size. The figure shows that in all cases, a larger data record size decreases the space overhead proportion. Therefore, it does not become more expensive to store larger data records on flash.

9.2 Searching By Timestamp

In the next experimental series, we investigate the average number of pages that must be read in order to find a record by its timestamp. Note that if we did not use an index, and thus had only data records on the flash, then we could find the expected record in O(1) time, as we could manipulate the position of the record directly. However, this would also violate our wear-leveling mechanism, since as we wouldn't be able to spread out the page writes evenly among data and index pages.

We evaluate the proposed search-by-timestamp methods LBSearch and ScaleSearch under two different index page layouts: (a) *Anchor*, in which every

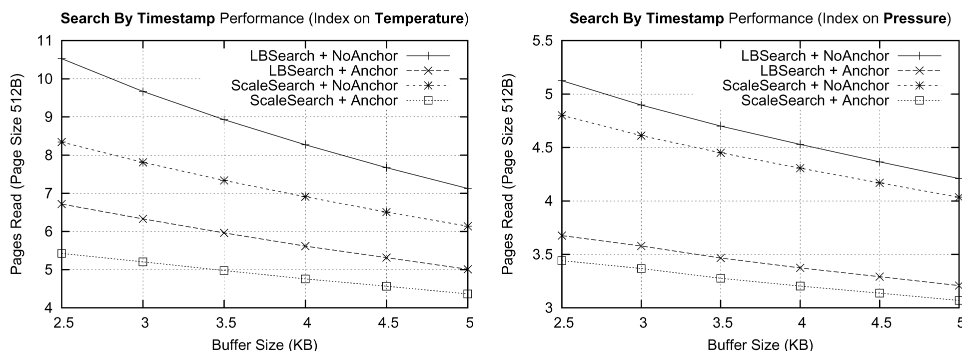

Fig. 14. Search-by-timestamp performance of the MicroHash index.

index page stores the last-known data record timestamp, and (b) *NoAnchor*, in which an index page does not contain any timestamp information.

Figure 14 shows our results using the Washington State climate dataset for both an index on temperature (left) and an index on pressure (right). The figure shows that using an anchor inside an index page is a good choice, as it usually reduces the number of page reads by two, while it does not present a significant space overhead (only 8 additional bytes).

The figure also shows that ScaleSearch is superior to LBSearch as it exploits the uniform distribution of index pages on the flash media. This allows ScaleSearch to get closer to the result in the first step of the algorithm.

Finally, the figure shows that even though the time window of the query is quite large (i.e., 5 years or 128MB), ScaleSearch is able to find a record by its timestamp in approximately 3.5–5 page reads. Given that a page read takes 6.25ms, this operation requires according to the RISE model, only 22–32ms or 84–120μJ.

9.3 Searching by Value: MicroHash versus ELF-Like Chaining

The cost of searching a particular value on the flash media is linear with respect to the size of flash media. However, a simple linear scan over 256 thousand pages found on a 128MB flash media would result in an overwhelmingly large search cost. One factor that significantly affects search performance is the occupancy of index pages. In the basic MicroHash approach, index pages on the flash might not be fully occupied. If index pages are not fully utilized, then a search requires iterating over more pages than necessary.

In this section, we perform an experimental comparison of the index chaining strategies presented in Section 6.3. We evaluate both MicroHash chaining (MHC) and elf-like chaining (ELC) using a fixed 3KB buffer. We deploy the chaining methods when temperature is utilized as the index (we obtained similar results for pressure). Our evaluation parameters are: (a) indexing performance (pages written) and (b) search performance (pages read).

Figure 15 (left) shows that MHC always requires less page writes than ELC. The reason is that ELC's strategy results in about 15% of sequential trashing, which is the characteristic presented in Section 6.3. Additionally, ELC requires

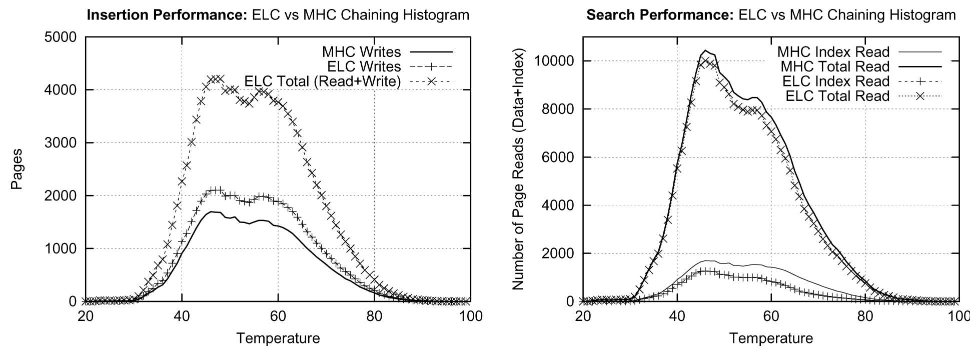

Fig. 15. Comparing MicroHash chaining (MHC) with eLF-like chaining (ELC) using (a) insertion performance and (b) searching performance by value.

a large number of page reads in order to replicate some of the index records. This is presented in the ELC total plot, which essentially shows that it requires as many page reads as page writes in order to index all records.

On the other hand, ELC's strategy results in more linked lists of fully occupied index pages than MHC. This has as a result an improved search performance, since the system is required to fetch less index pages during search. This can be observed in Figure 15 (right), in which we present the number of index pages read and the total number of pages (index + data). Nonetheless, we also observe that ELC only reduces the overall read gain by about 10%. This happens because the reading of data pages dominates the overall reading cost. However, when searches are more frequent, then the 10% is still an advantage and therefore ELC is more appropriate than its counterpart, MHC.

9.4 Great Duck Island Trace

We index measurements from the Great Duck Island study described in Section 8.3. For this study, we allocate a fixed 3KB index buffer, along with a 4MB flash media that has adequate space to store all the 97,000 20-byte data readings.

In each run, we index on a specific attribute (i.e., light, temperature, thermopile, thermistor, humidity, and voltage). We then record the overhead ratio of index pages Φ, the energy required by the flash media to construct the index, as well as the average number of page reads that were required to find a record by its timestamp. We omit the search-by-value results, since these are very similar to those presented in the previous subsection.

Table II shows that index pages never require more that 30% more space on the flash media. For some readings that do not change frequently (e.g., humidity), we observe that the overhead is as low as 8%. The table also shows that indexing the records has only a small increase in energy demand. Specifically, the energy cost of storing the records on flash without an index was 3042mJ, which is, on average, only 779mJ less than using an index. Therefore, maintaining index records does not impose a large energy overhead. Finally, the table

Table II. Performance Results from Indexing and Searching the Great Duck Island Dataset

Index On Attribute	Overhead Ratio Φ(%)	Energy Index (mJ)	ScaleSearch Avg Page Read
Light	26.47	4,134	4.45
Temperature	27.14	4,172	5.45
Thermopile	24.08	4,005	6.29
Thermistor	14.43	3,554	5.10
Humidity	7.604	3,292	2.97
Voltage	20.27	3,771	4.21

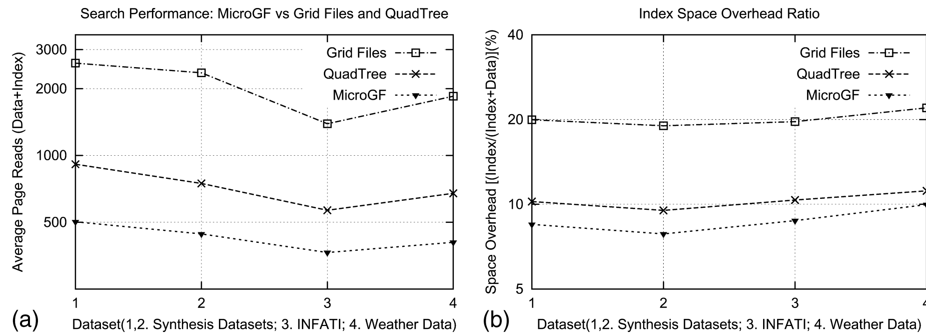

Fig. 16. Comparing MicroGF with grid files and quadtree using (a) searching performance by value (b) space overhead of index pages.

shows that we were able to find any record by its timestamp with 4.75 page reads, on average.

9.5 MicroGF versus Grid File and Quadtree

In this section we compare the performance of the MicroGF structure with two hash-based spatial indexing techniques: grid files and quadtrees. We utilize four datasets: 2 synthetic datasets with 250K random points in a 2D space, the INFATI dataset [Jensen et al. 2005], and the weather dataset [ATMO 2005]. We segment the recording space into 36 subregions and generate a random query area for each subregion. We then compare the average page access number for a query and the space overhead of index pages ($\Phi = \frac{IndexPages}{DataPages+IndexPages}$). As shown in Figure 16, on average, the MicroGF algorithm accesses 40% less pages than quadtree and 80% less pages than grid files in query processing. In addition, the index space overhead of MicroGF is 15% less than quadtree and 56% less than grid files. This is mainly attributed to the following two reasons: (a) Borrow quadrants can utilize the index page space more efficiently; and (b) subregion segmentation adjusts the index structure dynamically, according to the distribution of data.

10. RELATED WORK

In this section, we review prior work on storage and indexing techniques for sensor networks. While our work addresses both problems jointly, most prior work has considered them in isolation.

A large number of flash-based file systems have been proposed in the last few years, including the Linux-compatible journaling flash file system (JFFS and JFFS2) [Woodhouse 2006], and the yet another flash file system (YAFFS) [Wookey 2006] specifically designed for NAND flash, since it is portable under Linux, uClinux, and Windows CE.

Wear-leveling techniques for flash memory have been reported by flash card vendors such as Sandisk [Sandisk 2006]. These techniques are executed by a microcontroller located inside the flash card. The wear-leveling techniques are only executed within 4MB zones and are thus *local*, rather than *global*, which is the case in MicroHash and MicroGF. A main drawback of local wear-leveling techniques is that the writes are no longer spread out uniformly across all available pages. Finally, these techniques assume a dedicated controller, while our techniques can be executed by an ordinary microcontroller of the sensor device. Other vendors might utilize their own proprietary wear-leveling techniques. However, it is difficult to compare our structure with these techniques because most vendors usually don't disclose any of the details with regard to their architectures or algorithms.

Recently, various techniques have been proposed for data storage and indexing in sensor networks. Matchbox is a simple file system packaged with the TinyOS distribution [Hill et al. 2000]. It hides the lower details of wear-leveling and provides a pointer to each file (or page, in our context) on the flash memory. Had we used such an approach, this would have required a very large footprint to keep track of these pointers. The efficient log-structured flash file system (ELF) [Dai et al. 2004] is a log-like file structure designed for wear-leveling. It works by updating the desired file page and writing it into a new flash memory space. A few other indexing schemes have been proposed in the context of sensor networks. One such scheme is $TSAR$ in Desnoyers et al. [2005], which stores data locally at sensor nodes and indexes them by higher-tier nodes called proxies. Distributed index of features in sensor networks (DIFS [Greenstein et al. 2003]) and multidimensional range queries in sensor networks (DIM [Li et al. 2003]) extend the data-centric storage approach to provide spatially distributed hierarchies of indexes to data. All these techniques provide index topologies at the network level, but do not provide details on how to efficiently write the index information into sensor flash memories, as we do in our approach.

An R-tree index structure for flash memory on portable devices, such as PDAs and cell phones, has been proposed in Wu et al. [2003b]. These structures provide an efficient algorithm to compact several consecutive index units (R-tree nodes) into a page for better space utilization and search. In addition, they use an in-memory address translation table which hides details of the wear-leveling mechanism. However, such a structure has a very large footprint (3–4MB), rendering it inapplicable for all the sensor nodes (2KB–64KB RAM) we have so far. Other hash-based techniques (like Z-ordering and grid files) have been proposed for spatial data indexing. Z-ordering [Orenstein and Merrett 1984] divides each attribute into equal segments and imposes a linear ordering on the domain. The position in high-dimensional space is therefore mapped to

a 1D array. A problem with Z-ordering is that not all the points close in the *xy*-plane are close in Z-value. Another problem is that the bucket boundary of Z-ordering is fixed at the very beginning, which makes it inefficient on nonuniformly distributed data. Grid file [Nievergelt et al. 1984] is another kind of hash-based spatial index structure. The grid file partitions the space into rectanglular partitions and utilizes a grip directory (a matrix) to link to the data page containing a desired point. The size of the grid directory is adjusted as in extendible hashing. Several variants of grid files have been proposed [Seeger and Kriegel 1990; Whang and Krishnamurthy 1985] to improve the performance of grid files for biased distributed data. However, these algorithms read flash data records multiple times due to bucket update, which is inefficient in our context. As has been shown in our article, the MicroGF algorithm is robust on biased distributed data, while overcoming small RAM and wear-leveling problems.

Systems such as TinyDB [Madden et al. 2003] and Cougar [Yao and Gehrke 2003] achieve energy reduction by pushing aggregation and selections in the network, rather than processing everything at the sink. Both approaches propose a declarative approach for querying sensor networks. These systems are optimized for sensor nodes with limited storage and relatively short epochs, while our techniques are designated for sensors with larger external flash memories and longer epochs. Note that in TinyDB, users are allowed to define fixed-size materialization points through the `STORAGE POINT` clause. This allows each sensor to gather locally in a buffer some readings which cannot be utilized until the materialization point is created in its entirety. Therefore, even if there was enough storage to store MBs of data, the absence of efficient access methods makes retrieval of the desired values quite expensive.

In *data centric routing (DCR)*, such as directed diffusion [Intanagonwiwat et al. 2000], low-latency paths are established between the sink and sensors. Such an approach is supplementary to our framework. In data-centric storage (DCS) [Shenker et al. 2003], data with the same name (e.g., humidity readings) is stored at the same node in the network, hence offering efficient location and retrieval. However, the overhead of relocating data in the network will become huge if the network generates many MBs of GBs of data. Finally, local signal-based compression techniques, such as that proposed in Deligiannakis et al. [2004], could improve the compression efficiency of our framework and their investigation will be a topic of future research.

11. CONCLUSIONS

In this article we describe Microhash and MicroGF, which are efficient external memory index structures that address the distinct characteristics of flash memory in wireless sensor systems. We provide an extensive study of NAND flash memory when this is used as a storage media of a sensor node, and validate various design principles using our RISE platform. Our proposed access methods might provide a powerful new framework to realize *in situ* data storage in sensor networks. Additionally, we expect that these index structures

will enable new types of queries, such as temporal or top-k [Zeinalipour-Yazti et al. 2005] queries that have not been addressed adequately to-date. Our experimental testbed, written in nesC, with real traces from environmental and habitant monitoring, shows that the structures we propose are both efficient and practical.

REFERENCES

ATMO 2005. Live from Earth and Mars project. University of Washington, Seattle. http://www-k12.atmos.washington.edu/k12/grayskies/.

Banerjee, A., Mitra, A., Najjar, W., Zeinalipour-Yazti, D., Kalogeraki, V., and Gunopulos, D. 2005. Rise co-s : High performance sensor storage and co-processing architecture. In *Proceedings of the 2nd Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*.

Crossbow. 2005. Crossbow Technology, Inc. http://www.xbow.com/.

Dai, H., Neufeld, M., and Han, R. 2004. Elf: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. 176–187.

Deligiannakis, A., Kotidis, Y., and Roussopoulos, N. 2004. Compressing historical information in sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 527–538.

Desnoyers, P., Ganesan, D., and Shenoy, P. 2005. Tsar: A two tier sensor storage architecture using interval skip graphs. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*. 39–50.

Dipert, B. and Levy, M. 1994. *Designing with Flash Memory*. Annabooks.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. 1979. Extendible hashing—A fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3, 315–344.

Ganesan, D., Greenstein, B., Perelyubskiy, D., Estrin, D., and Heidemann, J. 2005. Multi-Resolution storage and search in sensor networks. *ACM Trans. Storage 1*, 3, 277–315.

Gay, D., Levis, P., Von, B., Welsh, M., Brewer, E., and Culler, D. 2003. The nesc language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Greenstein, B., Estrin, D., Govindan, R., Ratnasamy, S., and Shenker, S. 2003. Difs: A distributed index for features in sensor networks. In *Proceedings of the 1st IEEE International Workshop on Sensor Network Protocols and Applications*.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. *SIGPLAN Not. 35,* 11, 93–104.

Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the ACM IEEE Conference on Mobile Computing and Networking.*. 56–67 Tech. Rep. TR = 79.

Jensen, C., Lahrmann, H., Pakalnis, S., and Runge, J. 2005. The infati data. Time Center.

Levis, P., Lee, N., Welsh, M., and Culler, D. 2003. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems*.

Li, X., Kim, Y., Govindan, R., and Hong, W. 2003. Multi-Dimensional range queries in sensor networks. In *Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems*.

Litwin, W. 1980. Linear hashing: A new tool for file and table addressing. In *6th International Conference on Very Large Data Bases*. 212–223.

Lymberopoulos, D. and Savvides, A. 2005. Xyz: A motion-enabled, power aware sensor node platform for distributed sensor network applications. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks.*

Madden, S., Franklin, M., Hellerstein, J., and Hong, W. 2002. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Oper. Syst. Rev. 36*, 131–146.

MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. 2003. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 491–502.

NEEMA, S., MITRA, A., BANERJEE, A., NAJJAR, W., ZEINALIPOUR-YAZTI, D., GUNOPULOS, D., AND KALOGERAKI, V. 2005. Nodes: A novel system design for embedded sensor networks. In *Proceedings of the Internatonal IEEE Conference on Information Processing in Sensor Networks (IPSN)*.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. 1984. The grid file: An adaptable, symmetric multi-key file structure. *ACM Trans. Database Sys. 9*, 1, 38–71.

ORENSTEIN, J. AND MERRETT, T. 1984. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 181–190.

POLASTRE, J. 2003. Design and implementation of wireless sensor networks for habitat monitoring. Master's Thesis. University of California, Berkeley.

RAMAKRISHNAN, R. AND GEHRKE, J. 2002. Database management systems, 3rd ed. McGraw-Hill, New York.

SADLER, C., ZHANG, P., MARTONOSI, M., AND LYON, S. 2004. Hardware design experiences in zebranet. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. 227–238.

SAMET, H. 1984. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2, 187–260.

SANDISK 2006. Sandisk flash memory cards—Wear leveling. http://sandisk.com/pdf/oem/WPaperWearLevelv1.0.pdf.

SEEGER, B. AND KRIEGEL, H. 1990. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Databases*. 590–601.

SHENKER, S., RATNASAMY, S., KARP, B., GOVINDAN, R., AND ESTRIN, D. 2003. Data-Centric storage in sensornets. *ACM SIGCOMM Comput. Commun. Rev. 33*, 1, 137–142.

SHNAYDER, V., HEMPSTEAD, M., CHEN, B., WERNER-ALLEN, G., AND WELSH, M. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. 188–200.

SZEWCZYK, R., MAINWARING, A., POLASTRE, J., ANDERSON, J., AND CULLER, D. 2004. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. 214–226.

TANG, C. AND RAGHAVENDRA, C. 2004. Compression techniques for wireless sensor networks. In *Wireless Sensor Networks*. Kluwer Academic, Norwell, MA, 207–231.

WARNEKE, B., LAST, M., LIEBOWITZ, B., AND PISTER, K. 2001. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer*. *34*, 1, 44–51.

WHANG, K. AND KRISHNAMURTHY, R. 1985. Multilevel grid files. *Res. Rep. RC11516, IBM*, Yorktown Heights, New York.

WOODHOUSE, D. 2006. Jffs: The journalling flash file system. http://sources.redhat.com/jffs2/jffs2.pdf.

WOOKEY. 2006. Yaffs - A filesystem designed for nand flash. In *Linux Conference of Tutorials*. Leeds, UK.

WU, C., CHANG, L., AND KUO, T. 2003a. An efficient b-tree layer for flash memory storage systems. In *the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*.

WU, C., CHANG, L., AND KUO, T. 2003b. An efficient r-tree implementation over flash-memory storage systems. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*. 17–24.

XU, N., RANGWALA, S., CHINTALAPUDI, K., GANESAN, D., BROAD, A., GOVINDAN, R., AND ESTRIN, D. 2004. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. 13–24.

YAO, Y. AND GEHRKE, J. 2003. Query processing in sensor networks. In *Conference on Innovative Data Systems Research*.

ZEINALIPOUR-YAZTI, D., LIN, S., KALOGERAKI, V., GUNOPULOS, D., AND NAJJAR, W. 2005. Microhash: An efficient index structure for flash-based sensor devices. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*. 31–44.

ZEINALIPOUR-YAZTI, D., NEEMA, S., GUNOPULOS, D., KALOGERAKI, V., AND NAJJAR, W. 2005. Data acquisition in sensor networks with large memories. In *1st IEEE International Workshop on Networking Meets Databases (NetDB)*.

ZEINALIPOUR-YAZTI, D., VAGENA, Z., GUNOPULOS, D., KALOGERAKI, V., TSOTRAS, V., VLACHOS, M., KOUDAS, N., AND SRIVASTAVA, D. 2005. The threshold join algorithm for top-k queries in distributed sensor networks. In *Proceedings of the 2nd International Very Large Data Base Workshop on Data Management for Sensor Networks*. 61–66.

Received June 2006; revised ; accepted July 2006