HybridStore: An Efficient Data Management System for Hybrid Flash-Based Sensor Devices

Baobing Wang and John S. Baras

Department of Electrical and Computer Engineering
The Institute for Systems Research
University of Maryland, College Park
{briankw,baras}@umd.edu

Abstract. In this paper, we propose HybridStore, a novel efficient resource-aware data management system for flash-based sensor devices to store and query sensor data streams. HybridStore has three key features. Firstly, it takes advantage of the on-board random-accessible NOR flash in current sensor platforms to guarantee that all NAND pages used by it are *fully occupied* and written in a *purely sequential* fashion, and expensive in-place updates and out-of-place writes to an existing NAND page are *completely avoided*. Thus, both raw NAND flash chips and FTL-equipped (Flash Translation Layer) flash packages can be supported efficiently. Secondly, HybridStore can process typical joint queries involving both time windows and key value ranges as selection predicate extremely efficiently, even on large-scale datasets. It organizes a data stream into segments and exploits a novel index structure that consists of the inter-segment skip list, and the in-segment β-Tree and Bloom filter of each segment. Finally, HybridStore can trivially support time-based data aging without any extra overhead because no garbage collection mechanism is needed. Our implementation and evaluation with a large-scale real-world dataset in TinyOS reveals that HybridStore can achieve remarkable performance at a small cost of constructing the index.

1 Introduction

One of the main challenges in wireless sensor networks is the storage and retrieval of sensor data. Traditional centralized data acquisition techniques (e.g., [8]) suffer from large energy consumption, as all the readings are transmitted to the sink. In long-term deployments, it is preferable to store a large number of readings *in situ* and transmit a small subset only when requested [7]. This framework becomes practically possible with the new generation NAND flash that is very energy efficient with high capacity. Recent studies show that the NAND flash is at least *two orders of magnitude cheaper* than communication and *comparable in cost* to computation [10]. Therefore, extending the NAND flash to off-the-shelf low-end sensor platforms can potentially improve in-network processing and energy-efficiency substantially.

However, due to the distinctly different read and write semantics of the NAND flash, and tightly constrained resource on sensor platforms, designing an efficient resource-aware data management system for flash-based sensor devices is a very challenging task. Existing techniques, such as LA-Tree [1], μ-Tree [5], B-File [13], FlashDB [14]

P. Demeester, I. Moerman, and A. Terzis (Eds.): EWSN 2013, LNCS 7772, pp. 50–66, 2013.

and PBFilter [18], are not applicable to sensor platforms due to their large RAM footprints. Capsule [9] provides a stream-index object to store data stream efficiently, however, with very limited supports for general queries. Other works, such as TL-Tree [6] and FlashLogger [12], can only process time-based queries.

The most related works are Antelope [17] and MicroHash [7]. Antelope is a lightweight database management system for low-end sensor platforms, which enables runtime creation and deletion of databases and indexes. However, its main index for value-based queries, MaxHeap, requires expensive byte-addressable random writes in flash. Therefore, Antelope is more suitable for the NOR flash, which limits its performance because the NOR flash is much slower and more energy-consuming compared to the NAND flash. In addition, it can only retrieve *discrete values* in value-based range queries. MicroHash is an efficient index structure for NAND flash-based sensor devices, supporting value-based equality queries and time-based range queries *separately*. However, it suffers from out-of-place writes to existing pages, resulting in long chains of partially occupied pages. They alleviate this problem by combining multiple such pages into a fully occupied page, which induces extensive page reads and writes during insertions. More importantly, neither Antelope nor MicroHash can support joint queries involving both time windows and value ranges as selection predicate efficiently. That means, even though a query just wants to search readings within a certain value range in a small time window, they still needs to traverse the *whole global* index.

Existing works do not take advantage of both the on-board *random-accessible* NOR flash that is quite suitable for index structures, and external economical energy-efficient NAND flash with high-capacity, which is ideal for massive data storage. In this paper, we propose HybridStore, a novel efficient data management system for resource-constrained sensor platforms, which exploits both the on-board NOR flash and external NAND flash to store and query sensor data streams. In order to completely avoid expensive in-place updates and out-of-place writes to an existing NAND page, the index structure is created and updated in the NOR flash. To handle the problem that the capacity of the NOR flash on low-end sensor platforms is very limited (512KB to 1MB), HybridStore divides the sensor data stream into segments, the index of which can be stored in one or multiple erase blocks in the NOR flash. Since the NAND flash is much faster and more energy-efficient for reading, the index of each segment is copied to the NAND flash after it is full. Therefore, all NAND pages used by HybridStore are fully occupied and written in a purely sequential fashion, which means it can support both raw NAND flash chips and FTL-equipped flash packages efficiently.

HybridStore can process typical joint queries involving both time windows and key value ranges as selection predicate extremely efficiently even on large-scale datasets, which sharply distinguishes HybridStore from existing works. The key technique is a novel index structure that consists of the inter-segment skip list, and the in-segment β-Tree and Bloom filter of each segment. The inter-segment skip list can locate the desired segments within the time window of a query, and skip other segments efficiently. The β-Tree of a segment is the key data structure to support value-based queries. It exploits a simple prediction-based method to split each node in the tree adaptively to generate a rather balanced tree, even when key values are very unevenly distributed. The Bloom filter of a segment facilitates value-based *equality* queries inside that segment, which

can detect the existence of a given key value efficiently. Our index can eliminate a substantial number of unnecessary page reads when processing joint queries.

In addition, HybridStore can trivially support time-based data aging without any extra overhead, because no garbage collection mechanism is needed here, which will induce extensive page reads and writes to move valid pages within the reclaimed erase blocks to new locations. HybridStore can be used as a storage layer to provide a higher-level abstraction to applications that need to handle a large amount of data. For example, the design and implementation of Squirrel [11] can become much simpler if HybridStore is adopted for storage management.

The rest of this paper is organized as follows. We discuss the design considerations in Section 2. In Section 3, we explain the design of HybridStore. Our experimental results and analysis are presented in Section 4. Finally, Section 5 concludes this paper.

2 Design Considerations

In this section, we first discuss various constraints that make the design of HybridStore challenging. Then we discuss several design principles that result from these constraints.

2.1 Design Constraints

Flash Constraints Flash memory complicates the design of HybridStore by prohibiting in-place updates. Unlike magnetic disks, flash memories only allow bits to be programmed from 1 to 0. To reset a bit to 1, a large block of consecutive bytes must be erased, which is typically several kilobytes large [17]. There are two kinds of flash memories. The NOR flash is byte-addressable and permits random access I/O, but the erase blocks are very large. The NAND flash is page-oriented and limited to sequential writes within an erase block that can be significantly smaller than a NOR flash block. Reads and writes on the NAND flash happen at a page granularity. Since each page can be written only once after each complete block erasure, out-of-place writes to an existing NAND page are complex and very expensive. Portable flash packages such as SD cards and CF cards exploit a Flash Translation Layer (FTL) to hide many of these complexities and provide a disk-like interface. However, random page writes on current FTL-equipped devices are still *well over two orders of magnitude more expensive* than sequential writes, while semi-random writes are very efficient [13].

Table 1. Performance of flash memory operations

	Read		Write		Block Erase	
	Latency	Energy	Latency	Energy	Latency	Energy
Atmel NOR (per byte)	$12.12\mu s$	$0.26\mu J$	$12.6\mu s$	$4.3\mu J$	$12ms/2KB$	$648\mu J/2KB$
Toshiba NAND (per page)	$969.61\mu s$	$57.83\mu J$	$1081.42\mu s$	$73.79\mu J$	$2.6ms/16KB$	$65.54\mu J/16KB$

Energy Constraints. NOR flash and NAND flash are very different in speed and energy-efficiency. Table 1 shows the latency and energy consumption of each operation on the 512KB Atmel AT45DB041B NOR flash [3,10] equipped on the Mica family,

and the 128MB Toshiba TC58DVG02A1FT00 NAND flash [9] used extensively in the research community. Each NAND block consists of 32 pages of 512B each. We can observe that the NAND flash has a much larger storage capacity, and much faster and more energy-efficient I/O, while the only advantage of the NOR flash is random access and byte-addressable. These features influence the design of HybridStore extensively.

Memory Constraints. RAM is very limited on sensor platforms. Current low-end sensor platforms (e.g., MicaZ, Iris and Tmote Sky) are equipped with no more than 10KB RAM. Even on advanced sensor platforms (e.g., iMote2) with tens of megabytes RAM, RAM is still a very precious resource, because complex data processing applications with much higher RAM demands are expected to run on these platforms. Therefore, HybridStore must be designed to minimize the RAM footprint.

2.2 Design Principles

Given the above constraints, the design of HybridStore should follow a few design principles. Firstly, the system should take advantage of both the on-board NOR flash and external NAND flash. To support both raw NAND flash and FTL-equipped devices, random page writes should be avoided. To increase the energy-efficiency, out-of-place writes to an existing NAND page should be eliminated as well. Secondly, writes should be batched to match the write granularity of the NAND flash, which can be satisfied by using a page write buffer in RAM. In addition, since the NAND flash is much faster and more energy-efficient, most or even all reads should happen in the NAND flash. Thirdly, the system should support multiple storage allocation units and align them to erase block boundaries to minimize reclamation costs. Moreover, the system should maintain most data structures and information in flash whenever possible to minimize the RAM footprint. Finally, HybridStore should support data aging to reclaim space for new data when the NAND flash starts filling up with minimum overhead.

3 HybridStore

HybridStore provides the following interface to insert and query sensor readings:

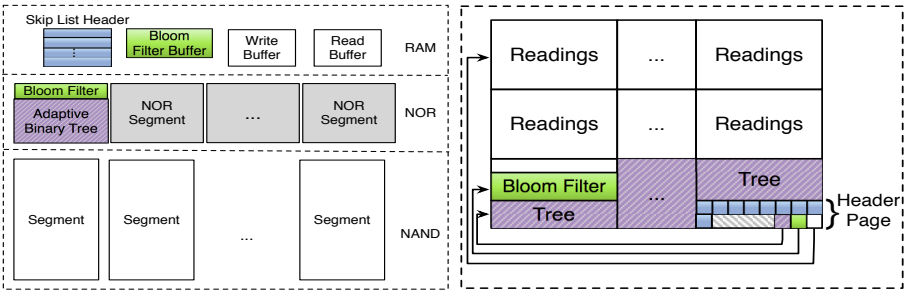
- command error_t insert(float key, void* record, uint8_t length)
- command error_t select(uint64_t t_1, uint64_t t_2, float k_1, float k_2)

The `select` function supports joint queries involving both time windows ($[t_1, t_2]$) and key ranges ($[k_1, k_2]$) as their *selection predicate*. We assume that a sensor mote generates readings periodically or semi-periodically as in adaptive sensing, and each reading can contain measurements from multiple types of sensors.

HybridStore consists of the following main components: Storage Manager, Index Manager, Query Processor, and Data Aging and Space Reclamation Module.

3.1 Storage Manager

The Storage Manager allocates storage space from the NOR flash and the NAND flash for index construction and data storage upon request. Fig. 1a shows the storage hierarchy. Both the NOR flash and the NAND flash are organized as circular arrays, resulting in the minimum RAM overhead, because we do not need to maintain a data structure in RAM to track free blocks. In addition, this organization directly addresses the write constraints, space reclamation, and wear-leveling requirements (Section 3.4).



(a) Storage hierarchy (b) NAND segment structure

Fig. 1. System architecture

At the highest level, the NOR flash is divided into equally-sized segments, each of which consists of one or multiple consecutive erase blocks. The NOR flash is allocated and reclaimed at the granularity of a segment. The NAND flash is allocated at the granularity of an erase block, but reclaimed at the granularity of a segment, which *logically* consists of several consecutive erase blocks storing readings, the index copied from the corresponding NOR segment (colored in green and purple), and the header page, as shown in Fig. 1b.

To minimize the RAM footprint of HybridStore, and comply with the write and read granularity of the NAND flash, only four absolutely necessary data structures are maintained in RAM. The write buffer is of one page size to batch the writes of readings to the NAND flash, and the read buffer is two pages large (one for index page reads and the other for data page reads). The other two data structures are the skip list header and Bloom filter buffer that are discussed in the next section.

3.2 Index Manager

In this section, we present the most important component of HybridStore: the *Index Manager*. HybridStore leverages a memory hierarchy to achieve more efficient index operations. Specifically, HybridStore divides a sensor data stream into dynamically-sized partitions, each of which is stored in a logical NAND segment. The index for this partition is first "cached" in a NOR segment, which is then copied to the corresponding logical NAND segment when it is filled. Next, HybridStore allocates a new NOR segment for the next partition, and stores its readings in a new NAND segment.

HybridStore exploits an *inter-segment skip list* to locate the segments covered by $[t_1, t_2]$ efficiently. Within each segment, HybridStore maintains an *in-segment β-Tree* to locate all readings within $[k_1, k_2]$ efficiently. To speed up the processing of value-equality queries (i.e., $k_1 = k_2$), an *in-segment Bloom filter* is also created for each segment to quickly detect the existence of a given key.

HybridStore chooses the partition-based index scheme instead of a global index as in [7,17] for the following reasons. Firstly, typical queries on sensor data always involve time windows. Since each logical NAND segment only stores the readings of a partition that corresponds to a small time window, many logical NAND segments outside the query time window can be skipped, reducing a substantial number of unnecessary page reads during query processing. Secondly, the number of readings in a partition is very limited compared to that in the whole steam. This allows index structure optimization and much cheaper index construction costs. Thirdly, the range of the key values of readings in a partition is very small. When processing a query with a value range within its selection predicate, many logical NAND segments outside the query value range can be skipped as well, further reducing many unnecessary page reads. Therefore, HybridStore is extremely efficient to process joint queries with both time windows and value ranges as their selection predicates. Finally, since all logical NAND segments are relatively independent of each other, HybridStore can support time-based data aging without any garbage collection mechanism, resulting in the substantially reduced overhead.

Now we discuss the index structure of HybridStore in details, which consists of three main modules: the inter-segment skip list, the in-segment β-Tree, and the in-segment Bloom filter. In the last subsection, the procedure to copy the "cached" index from the NOR flash to the NAND flash is described as well.

Inter-segment Skip List. The key issue to process a query is to locate the segments containing the readings within $[t_1, t_2]$. A naive approach is to scan the headers of all the segments one by one, if the time window of all the readings in a segment is available in its header and all segments are chained using previous segment address pointers. The expected cost is linear with the number of segments. However, we actually can do much better by exploiting the fact that the time windows of all segments are naturally ordered in descending order. To efficiently locate the desired segments, we organize all segments as a skip list [16]. A skip list is an ordered linked list with additional forward links added randomly, so that a search in the list can quickly skip parts of the list. The expected cost for most operations is $O(\log n)$, where n is the number of items in the list. Since segments are created in descending timestamp order, a new segment is always inserted at the front of the skip list, which can be efficiently implemented in a flash.

The skip list consists of a header node in RAM and a node in the header page of each segment (colored in blue in Fig. 1). Each node keeps *MaxLevel* number of forward pointers, each of which references the address of the header page of a segment and the timestamp of the first (oldest) reading stored in that segment. All pointers in the header node are initialized to *null* at first. When a segment is full, the skip-list node in its header page is created and inserted into the skip list before a new segment starts as follows. Firstly, a level $l \in [1, MaxLevel]$ is generated for it, randomly, such that a fraction p ($p = \frac{1}{2}$ typically) of the nodes with level i can appear in level $i + 1$. The maximum level of all segments in the current system, *curMaxLevel*, is updated if it is smaller than l.

Then every pointer in level $i \in [1, l]$ in the skip-list header, is copied as the level i pointer to the header page. Finally, the timestamp of the first reading in this segment and the header page address are written as the new level i pointer to the skip-list header.

The header page of each segment contains the timestamps of the first and the last readings stored in this segment. To search the segments containing readings within [t_1, t_2], we first locate the most recent segment with a *start* timestamp smaller than t_2, using an algorithm similar to the search algorithm in [16]. The subsequent segments can be located by following the level 1 pointer in the skip-list node of each segment, until a segment with a start timestamp smaller than t_1 is encountered.

In-segment β-Tree. To support value-based equality and range queries, HybridStore exploits an adaptive binary tree structure, called β-Tree, to store the index for each segment. The β-Tree consists of a set of equally-sized buckets, each of which stores index entries within a certain value range. The header of a bucket consists of its value range, the bucket IDs of its two children, and the value to split its value range to obtain the value ranges for its children. The β-Tree is first created and updated in a NOR segment, and then copied to the corresponding logical NAND segment.

An index entry $< key, addr >$ is inserted into the β-Tree as follows. Suppose the current bucket is b and $key \in (b.min, b.max]$, then this entry is appended to b. Otherwise, we traverse the β-Tree to locate the leaf bucket b' such that $key \in (b'.min, b'.max]$, and append this entry to b'. If b (or b') is full, its value range is split into $(min, mid]$ and $(mid, max]$, and a new bucket b_{new} is allocated as its left child if $key \leq mid$, or as its right child otherwise. Then the headers of both b (or b') and b_{new} are updated correspondingly and this entry is inserted into b_{new}. Since the children of a bucket are allocated only if necessary, it is possible to have $b' = null$ in the above case if, for example, b' is the left child of its parent but only the right child of its parent has been allocated since its splitting. In this case, a new bucket is allocated for b' first.

Instead of splitting the value range evenly as in [7,17], HybridStore uses a prediction-based adaptive bucket splitting method, because readings are temporally correlated, which can be used to predict the value range of the following readings based on the most recent readings, and split a bucket range accordingly. This method is preferred for the following reasons. Firstly, each partition contains readings in a small value range. If the very large range for all possible key values is split evenly in each step, the index tree of a segment may degenerate to a long list at the beginning, resulting in more time and energy consumption to traverse the index. Secondly, although the whole range is very large, most readings will belong to a much smaller range due to their uneven distribution. As shown in Fig. 11 in [7], over 95% of the temperature measurements belong to [30 °F, 80 °F], while the whole range is [−60 °F, 120 °F]. Again, the evenly splitting method will result in a rather unbalanced tree.

HybridStore exploits the Simple Linear Regression estimator for prediction due to its simplicity in computation, negligible constant RAM overhead, and high accuracy for temporally correlated data. HybridStore buffers the *keys* of the most recent m readings and predicts the value range for the following $2m$ readings, where m is the number of entries that can be stored in a bucket. Suppose the range of the current bucket is $(x, y]$ and the predicted range is $[l, h]$, the splitting point *mid* is computed as:

$$mid = \begin{cases} (l+h)/2 & \text{if } [l,h] \subseteq (x,y] \\ (x+y)/2 & \text{if } (x,y] \subseteq [l,h] \\ (x+h)/2 & \text{if } l \le x < h \le y \text{ and } 2m * (h-x)/(h-l) > m \\ \max(h,(x+y)/2) & \text{if } l \le x < h \le y \text{ and } 2m * (h-x)/(h-l) \le m \\ (l+y)/2 & \text{if } x \le l < y \le h \text{ and } 2m * (y-l)/(h-l) > m \\ \min(l,(x+y)/2) & \text{if } x \le l < y \le h \text{ and } 2m * (y-l)/(h-l) \le m \end{cases}$$

Compared to MaxHeap [17], an evenly splitting scheme, HybridStore can generate a more balanced tree. For example, suppose each bucket can store the index entries generated in half an hour, the current temperature is 80 °F and will increase 1 °F every half an hour, and the whole range is [−60 °F, 120 °F]. Assuming HybridStore can predict accurately, the root will be split with $mid = 82$ °F, and its right child will be split with $mid = 84$ °F. The resulting β-tree will have three layers for readings in the following 2.5 hours, while MaxHeap degenerates to a list with 5 buckets. In addition, MaxHeap allocates two child buckets at the same time when a bucket needs to be split, resulting in more wasted space with empty buckets. To handle the uneven distribution of key values, MaxHeap selects a bucket for an index entry based on the hashed key value, but store the unhashed key. As a result, MaxHeap can only retrieve *discrete values* in a range search. More importantly, however, any spatial correlations of index insertions are destroyed. After hashing, the index entries for consecutive readings are very likely to be stored in many different buckets, which will increase the read costs both for bucket locating and query processing substantially. On the contrary, these entries will be stored in the same bucket in the β-Tree. Finally, different from [7,17], our prediction-based adaptive splitting scheme does not require a priori knowledge of the whole range, which makes HybridStore more suitable for general applications.

In-segment Bloom Filter. A special case of value-based queries is value-based *equality* search that is also often desired [17]. Although β-Trees can support this kind of queries, HybridStore needs to traverse the whole β-Tree even when the given key does not exist in a segment. To better support these queries, HybridStore creates a Bloom filter [4] for each segment to detect the existence of a key value in this segment efficiently.

A Bloom Filter (BF) is a space-efficient probabilistic data structure for membership queries in a set with low false positive rate but no false negative. It uses a vector of v bits (initially all set to 0) to represent a set of n elements, and q independent hash functions, each producing an integer $\in [0, v-1]$. To insert an element a, the bits at positions $h_1(a), \ldots, h_q(a)$ in the bit vector are set to 1. Given a query for element a', all bits at positions $h_1(a'), \ldots, h_q(a')$ are checked. If any of them is 0, a' cannot exist in this set. Otherwise we assume that a' is in this set.

Since a Bloom filter requires bit-level random writes, it must be buffered in RAM. However, if a Bloom filter is used to represent all readings in a segment, this buffer size may be very large in order to keep p very low. For example, suppose a segment can store 4096 readings and three hash functions are used, in order to keep $p \approx 3.06\%$, then the size of its Bloom filter buffer must be at least 4KB.

To reduce the RAM footprint, HybridStore *horizontally* partitions the large Bloom filter of a segment into a sequence of small fix-sized Bloom filters sections, and allocates

a small buffer in RAM for a section. Suppose the buffer size is v bits, the number of hash functions is q, and the desired false positive rate is p, the maximum number n of readings that a BF section is able to represent can be calculated from the equation $p = \left(1 - \left(1 - \frac{1}{v}\right)^{qn}\right)^q$. Whenever n readings have been inserted into the current BF section, the BF buffer is flushed to the current NOR segment, and then initialized for the next section. In our implementation, $v = 2048$ bits, $q = 3$, $p = 3.06\%$, and $n = 256$.

Algorithm 1. checkBF($addr, l, k$)

Input: *addr*: start address of the BF pages, l: length of a BF fragment in bytes, k: key value
Output: *true* if there is a record with the given key in this segment; *false* otherwise

```
1:  h ← hashcode(k); bv ← createBitVector(⌈NAND_Page_Size / l⌉);
2:  for i = 0 → h.size do
3:      f ← loadPage(addr + ⌊h[i] / 8l⌋ * NAND_Page_Size); bv.setAll();
4:      for j = 0 → bv.size do
5:          mask ← 0x80 >> (h[i] % 8); offset ← h[i] % 8l;
6:          if f[j * l + ⌊offset / 8⌋] & mask == 0 then bv.clear(j); end if
7:      end for
8:      exist ← false;
9:      for i = 0 → bv.size do
10:         exist = exist | bv.get(i);
11:     end for
12:     if !exist then return false; end if
13: end for
14: return true;
```

A drawback of horizontal partitioning is that all BF sections of a segment must be scanned to decide whether the given key exists in this segment. HybridStore addresses this drawback by *vertically* splitting these BF sections into fragments and group them into pages when the NOR segment is copied to the logical NAND segment. Assume there are s BF sections in the current segment when it is full. Then the size of a fragment is $l = \left\lfloor \frac{NAND\_Page\_Size}{s} \right\rfloor$ bytes, so that the bits in the range $[i * 8l, (i + 1) * 8l - 1]$ from every BF section are grouped to page $i \in \left[0, \lceil \frac{NAND\_Page\_Size}{l} \rceil - 1\right]$. Thus HybridStore only needs to scan at most q pages at $\left\lfloor \frac{h_1(k)}{8l} \right\rfloor, \ldots, \lfloor \frac{h_q(k)}{8l} \rfloor$ when checking key k, as shown in Algorithm 1. For each hash code $h_i(k)$, HybridStore firstly loads the page containing all the $h_i(k)$-th bits of every BF section (Line 3), and then check the corresponding bit in each BF fragment (Line 4–7). If the corresponding bit is not set in any fragment in that page (Line 9–11), we can conclude that this key does not exist (Line 12). Note that a segment cannot store more than $n * NAND\_Page\_Size$ readings to guarantee $l \neq 0$.

Copy Index from the NOR Flash to the NAND Flash. Since the NAND flash is much faster and more energy-efficient, the index of a segment that is created and updated in the NOR flash is copied to the NAND flash after this segment is full as follows. Firstly, the β-Tree is copied and multiple consecutive buckets are written to the same page if they can fit in. The Query Processor is able to translate the bucket ID to the right page address and offset to read a desired bucket. Therefore, the bucket size should be $\frac{1}{2^i}$ of the NAND page size. Secondly, the BF sections are copied as described above. Finally, the time window

and value range of all readings in this segment, the addresses of the first page for readings, of the β-Tree, and of the Bloom filter, the length of a BF fragment, and the skip-list node are written to the *next* page, which is the header page of this segment. Therefore, all page writes in the NAND flash are purely sequential; the reason why HybridStore can support both raw NAND flash chips and FTL-equipped NAND flash cards efficiently.

3.3 Query Processor

Algorithm 2 describes how HybridStore can efficiently process joint queries. The basic idea is to skip all the segments that do not satisfy the selection predicate by checking their header pages, or do not contain the given key by checking their Bloom filters.

Algorithm 2. select(t_1, t_2, k_1, k_2)

Input: Time window $[t_1, t_2]$ and key value range $[k_1, k_2]$ of a query
Output: The records that satisfy the query criteria

```
 1: addr ← skipListSearch(t2);
 2: while addr ≥ 0 do
 3:     addr ← segmentSearch(addr, t1, t2, k1, k2);
 4: end while
 5: signal finished;
 6: function SEGMENTSEARCH(addr, t1, t2, k1, k2)
 7:     f ← loadPage(addr);
 8:     if [k1, k2] ∩ [f.minK, f, maxK] ≠ ∅ then
 9:         if t1 == t2 then                          ▷ Simple time-based equality query
10:             scaleBinarySearch(f.dataBList, t1);
11:             return -1;
12:         else if k1 == k2 then                     ▷ Value-based equality query
13:             if checkBF(f.bfAddr, f.bfFragSize, k1) == false then
14:                 return (f.startT > t1 && f.sl[0].time ≥ System.minT) ? f.sl[0].addr : −1;
15:             end if
16:         end if
17:         q ← createQueue(f.idxBList[0]);           ▷ Traverse β-tree
18:         while !q.empty() do
19:             b ← loadBucket(q.dequeue());
20:             for i = 0 → b.record.size do
21:                 if (b.record[i] ≠ null) && (b.record[i].key ∈ [k1, k2]) then
22:                     dataP ← loadPage(b.record[i].addr);
23:                     if dataP[b.record[i].addr % P].timestamp ∈ [t1, t2] then
24:                         signal dataP[b.record[i].addr % P];
25:                     end if
26:                 end if
27:             end for
28:             if (b.left ≠ null) && (b.middle ≥ k1) then q.enqueue(b.left); end if
29:             if (b.right ≠ null) && (b.middle < k2) then q.enqueue(b.right); end if
30:         end while
31:     end if
32:     return (f.startT > t1 && f.sl[0].time ≥ System.minT) ? f.sl[0].addr : −1;
33: end function
```

HybridStore starts by locating the most recent segment within the time window using the inter-segment skip list (Line 1), and then scans segments sequentially until the whole time window has been covered (Line 2–4). For each segment, its header page is loaded first. If this segment potentially contains readings within the value range of the query (Line 8), HybridStore begins to traverse its index. Otherwise, this segment will be skipped. Two special cases are treated separately. Time-based *equality* queries are processed using a scale binary search (Line 9) on the pages storing readings, which exploits the fact that sensor readings are generated periodically or semi-periodically to refine the middle in each iteration, similar to [7]. For value-based *equality* queries, HybridStore first checks the existence of the key in this segment using Algorithm 1 (Line 12). If this key does not exist, this segment will be skipped as well. For general joint queries, the β-Tree of this segment is traversed using the Breadth-First Search algorithm (Line 17–30).

To reduce the RAM footprint, HybridStore returns readings on a record-by-record basis. Actually, the β-Tree traversal is also implemented in a split-phase fashion (bucket-by-bucket using the *signal-post* mechanism), although a queue-based implementation is presented here for clarity. In addition, to take advantage of the temporal correlations and spatial locality of readings, a small record pool is applied here to buffer the data page addresses that will be loaded in increasing order. Therefore, instead of loading the data page immediately (Line 22), the address of each reading is translated to the corresponding page address that is then added to the pool. When the pool is full or the β-Tree traversal is finished, HybridStore loads these pages in order and scans each page to return the readings that satisfy the selection predicates.

3.4 Data Aging and Space Reclamation

As shown in [10], a sensor mote can store over 10GB data during its lifetime. If the capacity of the external NAND flash is not big enough to store all these data, some data need to be deleted to make room for future data as the flash starts filling up. HybridStore exploits a simple time-based data aging mechanism to discard the oldest data. When no space is available on the NAND flash to insert the current reading, HybridStore will locate the oldest segment and erase all the blocks in that segment. Then the minimum timestamp of all the readings currently stored in the system (i.e., `System.minT`) is updated. Since the NAND flash is organized as a circular array, wear leveling is trivially guaranteed. In addition, since segments are independent of each other, no garbage collection mechanism is needed here. On the contrary, other index schemes (e.g., [2,9,17]), require extensive page reads and writes to move valid pages within the reclaimed erase blocks to new locations, and maintain extra data structures in flash or RAM.

Note that we do not need to delete the pointers referencing the reclaimed segment from the skip list, even though they become invalid now. This problem is handled by the `select` algorithm (Line 14 and 32). Whenever a pointer with a timestamp smaller than `System.minT` is encountered, the `select` algorithm knows that this pointer is invalid and the query processing is completed successfully. On the contrary, for *each invalid index entry*, MicroHash [7] must load the referenced data page to learn that this page has been deleted and re-used, resulting in many unnecessary page reads.

4 Implementation and Evaluation

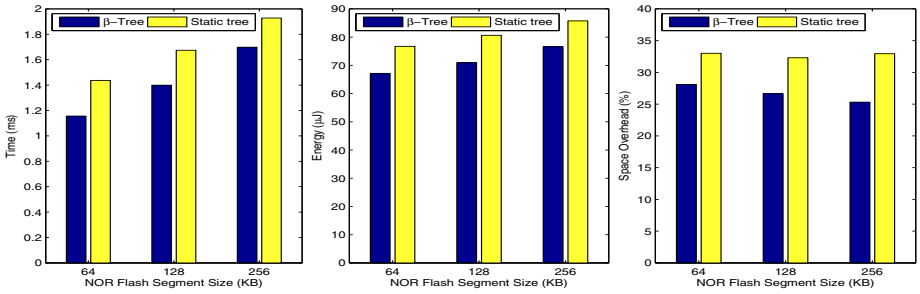
In this section, we describe the details of our experiments. HybridStore is implemented in TinyOS 2.1 and simulated in PowerTOSSIMz [15], an accurate power modeling extension to TOSSIM for MicaZ sensor platform. We additionally developed an emulator for a Toshiba TC58DVG02A1FT00 NAND flash (128MB), and a library that intercepts all communications between TinyOS and flash chips (both the NOR and the NAND flash) and calculate the latency and energy consumption based on Table 1. With all features included, our implementation requires approximately 16.5KB ROM and 3.2KB RAM, which is well below the limit of most constrained sensor platforms.

We adopt a trace-driven experimental methodology in which a real dataset is fed into the PowerTOSSIMz simulator. Specifically, we use the Washington Climate Dataset, which is a real dataset of atmospheric information collected by the Department of Atmospheric Sciences at the University of Washington. Our dataset contains 2, 630, 880 readings on a per-minute basis between 01/01/2000 and 12/31/2004. Each reading consists of temperature, barometric pressure, etc. We only index the temperature values and use the rest as part of the data records, each of which is of 32 bytes. To simulate data missing (e.g., reading drops due to the long latency during long queries and block erasures, or adaptive sensing), 5% readings are deleted randomly. For each kind of queries, 1000 instances are generated randomly and their average performance is presented here.

We compare HybridStore with MicroHash [7], Antelope [17] and the system in [2]. Since we do not have enough details to reproduce their complete experiments, we directly use the results reported in their papers if necessary. We use the same dataset as MicroHash, and fully implement the static bucket splitting scheme used in [7,17].

4.1 Insertions

We first insert all readings into the sensor mote and record the performance of each insertion. Fig. 2 shows the average performance of the β-Tree and the static bucket splitting scheme used in [7,17]. Compared to the β-Tree, the latter scheme consumes 13.24% more energy, induces 16.76% more space overhead, and results in 18.47% more latency on average. The latency and energy consumption of each insertion approximately equal to the write of 1.31 NAND pages and 0.91 NAND pages, respectively.



(a) Latency (b) Energy (c) Space Overhead

Fig. 2. Performance per insertion

Fig. 3 shows part of the timeline for insertions at the beginning when the NOR segment size is 64KB. Our key observations are as follows. First, although it is very energy-consuming ($26.8mJ$) to transfer the index from NOR flash to NAND flash, it only happens once every 3–4 days and is independent of the data record size. Second, during regular operation, each insertion consumes only $34.4\mu J$. When a reading is not within the current bucket range, the proper bucket can be located or created after traversing about 8–10 bucket headers in β-Tree, even when the current segment is almost full. Since there are about 236 buckets in the β-tree for each segment, our adaptive bucket splitting scheme generates a rather balanced tree. The points corresponding to around $0.15mJ$ and $1.2mJ$ additionally include the energy consumption to flush the write buffer to NAND flash and the Bloom filter buffer to NOR flash, respectively.

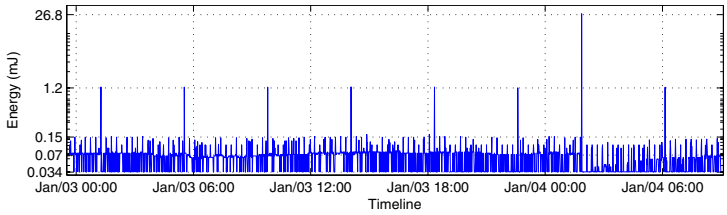


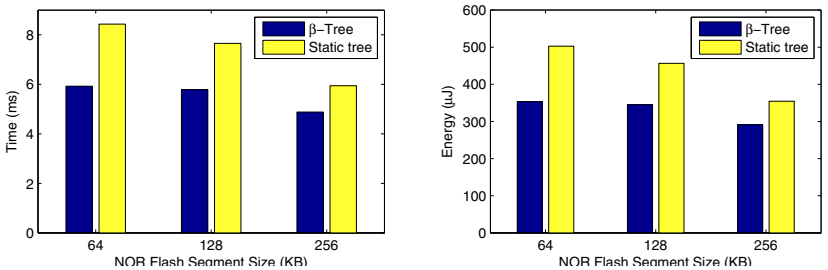
Fig. 3. Energy consumption of insertions

4.2 Time-Based Equality Queries

We also investigate the performance to process a time-equality query to find a record with a specific timestamp (i.e., $t_1 = t_2$). Fig. 4 shows that even though the query time window is quite large (i.e., over 2.5 million readings in 5 years), HybridStore is able to locate the record with about 5–6 page reads. Such a high performance can be achieved due to the following reasons. Firstly, the β-Tree improves the storage efficiency, resulting in fewer segments to store the same number of readings. Secondly, the skip list can locate the segment containing the required timestamp efficiently. Thirdly, the scale binary search can locate the page quickly because all readings are stored continuously in each segment, avoiding traversing through a block chain. Therefore, compared to a global index (e.g., MicroHash requires about 5.4 page reads to process such a query when the buffer size is 2.5KB, as shown in Fig. 14 in [7]), HybridStore has almost the same performance, while consuming less RAM. Compared to Antelope [17], HybridStore can achieve a much better performance if the same dataset is used.

4.3 Joint Queries: Time-Based Range and Value-based Equality

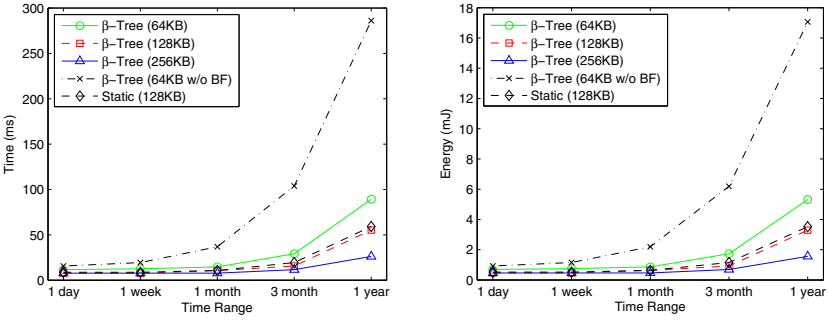
In this scenario, we study the impact of Bloom filter on joint time-based range and value-based equality queries. Fig. 5 shows the average performance per query to search nonexistent key values. We can observe that the in-segment Bloom filter can significantly improve the performance of value-based equality queries (more than 3 times improvement when the NOR segment size is 64KB and the time range is more than 3 months). In addition, the β-Tree can reduce the latency and energy consumption for queries involving large time window by about $4ms$ and $230\mu J$, respectively. Finally,



(a) Latency (b) Energy

Fig. 4. Performance of time-equality queries: HybridStore (β-Tree) v.s. Antelope [17]

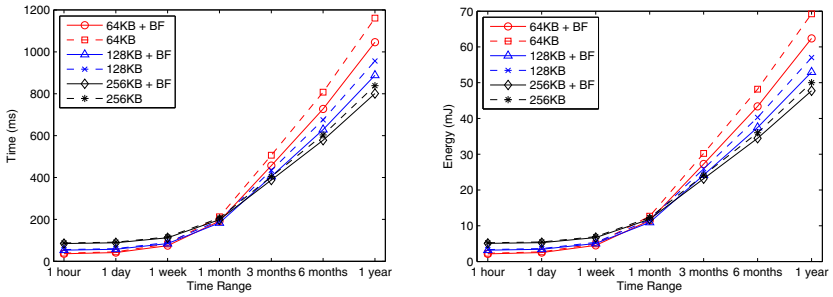
HybridStore is extremely efficient to check the existence of key values. When the NOR segment size is 256KB, HybridStore can decide the existence of a key value in over 0.5 million readings spanning one year time window in 26.18ms, consuming only 1.56mJ.



(a) Latency (b) Energy

Fig. 5. Impact of Bloom Filter on value-based equality queries for nonexistent keys

We also investigate the average performance per query to search existing key values, which is shown in Fig. 6. The key values vary in [40 °F, 60 °F]. We can observe that the in-segment Bloom filter can reduce the latency and energy consumption for queries involving large time window by 38–116ms and 3–7mJ, respectively. More importantly, HybridStore requires approximately only 826 page reads to get all readings with the given key value in one year time window when the NOR segment size is 256KB. Comparatively, MicroHash requires about 8700 page reads on average to search a given key value $\in$ [40 °F, 60 °F] in five years time window (inferred from Fig. 15 in [7]). Even if we assume that MicroHash can "intelligently" stop searching when a reading below the lower bound of the query time window is encountered, it still requires much more than 1740 page reads for one year time window, because many index pages and data pages are read unnecessarily.

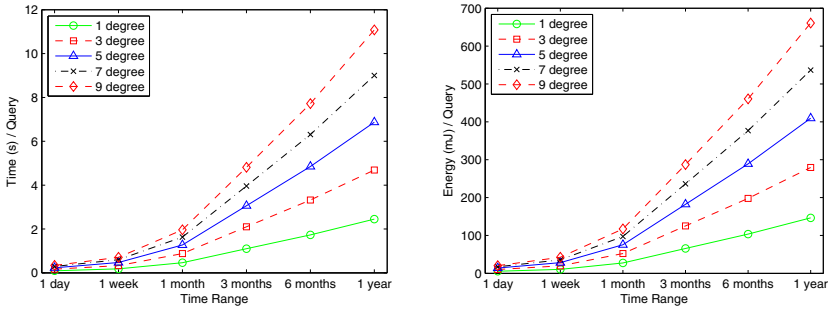


(a) Latency (b) Energy

Fig. 6. Impact of Bloom Filter on value-based equality queries for existing keys

4.4 Joint Queries: Both Time-Based and Value-based Ranges

In this scenario, we investigate the most common type of queries that involves both time windows and value ranges as selection predicates. Fig. 7 shows the average performance per query when the NOR segment size is 64KB. We can observe that HybridStore is extremely efficient to process such queries. When the value range is 1 °F and the time window is 1 month (typical queries, because readings in small time windows are more interesting), HybridStore can finish the query in 461.6ms, consumes only 27.5mJ and returns 2678 readings. For queries involving a large value range (e.g., 9 °F) and a long time window (e.g., 1 year), HybridStore can return 120,363 readings in 11.08s, consuming only 660.7mJ (92.04μs and 5.48μJ per reading on average). Compared to Antelope [17], since the NOR flash is much slower and less energy-efficient, Antelope will take about 20s to retrieve 50% readings from a table with only 50,000 tuples in a range query (shown in Fig. 8 in [17]).



(a) Total Latency per query (b) Total energy per query

Fig. 7. HybridStore performance per query of full queries

Another index scheme proposed in [2] can support range queries. It will consume about 40mJ on average to process a query with 5-degree range on about only 100,000

readings (about 13,000 readings are returned). Comparatively, HybridStore will consume 75.61mJ to return the same number of readings by processing the same query on more than 2.5 million readings. While our dataset size is 25 times larger than the dataset used in [2], HybridStore consumes only 89% more energy. Therefore, HybridStore is more energy efficient to support queries on large-scale datasets. Besides, the size of each reading in [2] is much smaller, which consists of only a timestamp and a temperature value (12B is enough, while our record size is 32B), resulting in much less data pages. Finally, their scheme requires much more RAM resource (close to 10KB, shown in Page 10 in [2]), because the per-partition B-tree index, interval table, and the *last page of each interval's list* must be maintained in RAM.

5 Conclusions

In this paper, we proposed HybridStore, an efficient data management system for low-end sensor platforms, which exploits both the on-board NOR flash and external NAND flash to store and query sensor data. Compared to existing works that can only support simple queries, HybridStore can process typical joint queries involving both time windows and key value ranges as selection predicates extremely efficiently, even on large-scale datasets. Our evaluation with a large-scale real-world dataset reveals that HybridStore can achieve remarkable performance at a small cost of constructing the index. Therefore, HybridStore provides a powerful new framework to realize *in situ* data storage in WSNs to improve both in-network processing and energy-efficiency.

Acknowledgements. We would like to thank our shepherd, Luca Mottola, and the anonymous reviewers for their insight and detailed feedback. This work is partially supported by DARPA and SRC through grant award 013641-001 of the FCRP, by the National Science Foundation (NSF) under grant award CNS-1035655, and by the National Institute of Standards and Technology (NIST) under grant award 70NANB11H148.

References

1. Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., Singh, S.: Lazy-adaptive tree: an optimized index structure for flash devices. In: ACM VLDB, pp. 361–372 (2009)
2. Agrawal, D., Li, B., Cao, Z., Ganesan, D., Diao, Y., Shenoy, P.: Exploiting the interplay between memory and flash storage in embedded sensor devices. In: 16th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications, pp. 227–236 (2010)
3. Atmel Inc.: AT45DB041B, `http://www.atmel.com/Images/doc3443.pdf`
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
5. Kang, D., Jung, D., Kang, J.U., Kim, J.S.: μ-tree: an ordered index structure for NAND flash memory. In: ACM EMSOFT, pp. 144–153 (2007)
6. Li, H., Liang, D., Xie, L., Zhang, G., Ramamritham, K.: TL-Tree: flash-optimized storage for time-series sensing data on sensor platforms. In: ACM SAC, pp. 1565–1572 (2012)
7. Lin, S., Zeinalipour-Yazti, D., Kalogeraki, V., Gunopulos, D.: Efficient indexing data structures for flash-based sensor devices. ACM Trans. on Storage 2(4), 468–503 (2006)

8. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an acquisitional query processing system for sensor networks. ACM TODS 30(1), 122–173 (2005)
9. Mathur, G., Desnoyers, P., Ganesan, D., Shenoy, P.: Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In: ACM SenSys, pp. 195–208 (2006)
10. Mathur, G., Desnoyers, P., Ganesan, D., Shenoy, P.: Ultra-low power data storage for sensor networks. In: ACM/IEEE IPSN, pp. 374–381 (2006)
11. Mottola, L.: Programming storage-centric sensor networks with squirrel. In: ACM IPSN, pp. 1–12 (2010)
12. Nath, S.: Energy efficient sensor data logging with amnesic flash storage. In: ACM/IEEE IPSN, pp. 157–168 (2009)
13. Nath, S., Gibbons, P.B.: Online maintenance of very large random samples on flash storage. In: ACM VLDB, pp. 970–983 (2008)
14. Nath, S., Kansal, A.: FlashDB: dynamic self-tuning database for NAND flash. In: ACM/IEEE IPSN, pp. 410–419 (2007)
15. Perla, E., Catháin, A.O., Carbajo, R.S., Huggard, M., Mc Goldrick, C.: PowerTOSSIMz: realistic energy modelling for wireless sensor network environments. In: Proc. of the 3rd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks, pp. 35–42 (2008)
16. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33(6), 668–676 (1990)
17. Tsiftes, N., Dunkels, A.: A database in every sensor. In: ACM SenSys, pp. 316–332 (2011)
18. Yin, S., Pucheral, P., Meng, X.: A sequential indexing scheme for flash-based embedded systems. In: ACM EDBT, pp. 588–599 (2009)