Flash-Optimized Temporal Indexing for Time-Series Data Storage on Sensor Platforms

HUAN LI, DONG LIANG, LIHUI XIE, Beihang University
GONG ZHANG, Carnegie Mellon University
KRITHI RAMAMRITHAM, Indian Institute of Technology Bombay

While it is essential to exploit in-network processing in wireless sensor networks in order to save bandwidth and energy, we are constrained by the limited storage available in off-the-shelf sensor devices. NAND flash memory has great potential for extending storage capacity for sensor applications. Since each sensor platform is typically equipped with limited main memory and sensor data, as well as the fact that queries are temporal, existing flash index or file systems for general portable devices are not suitable for sensor networks. We propose Time-Log Tree (TL-Tree), a novel unbalanced and cascaded structure, that takes advantage of available flash capacity while making use of the time-series property as a primary feature for optimizing both memory and energy constraints. Extensive experiments show TL-Tree's ability to utilize both flash capacity and temporal locality to support sensor data processing. Compared to other schemes, it achieves much better access and energy savings for different kinds of random and temporal range queries. In addition, TL-Tree can also be easily extended to support value-based queries. We have developed a hardware board that includes a raw 128MB NAND flash chip on MicaZ mote. We have also implemented a flash driver and the TL-Tree to demonstrate the practicality of this idea.

Categories and Subject Descriptors: H.3.1 [**Content Analysis and Indexing**]: Indexing methods; D.4.2 [**Storage Management**]: Flash memory

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Sensor platform, NAND flash, time-series data storage, memory-aware index structure, energy efficiency

ACM Reference Format:
Huan Li, Dong Liang, Lihui Xie, Gong Zhang, and Krithi Ramamritham. 2014. Flash-optimized temporal indexing for time-series data storage on sensor platforms. ACM Trans. Sensor Netw. 10, 4, Article 62 (June 2014), 30 pages.
DOI: http://dx.doi.org/10.1145/2526687

1. INTRODUCTION

The availability of low-cost embedded systems has fostered the development of wireless sensor networks (WSNs). Networks of resource-constrained sensor devices are able to sense, disseminate, and process information from the physical environment in real

This work is supported by the National Nature Science Foundation of China, NSFC (61170293), the National High Technology Research and Development Program of China (863 Program: 2012AA0011203), State Key Laboratory of Software Development Environment (SKLSDE-2012ZX-03), and the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry.
Authors' addresses: H. Li (corresponding author), D. Liang, L. Xie, State Key Laboratory of Software Development Environment, School of Computer Science & Engineering, Beihang University, Beijing, China; email: lihuan2008@gmail.com; G. Zhang, Information Networking Institute, Carnegie Mellon University, Pittsburgh, PA; K. Ramamritham, Department of Computer Science & Engineering, Indian Institute of Technology Bombay, India.
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 copies are not made or distributed for profit or 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.
2014 Copyright held by the Owner/Author. Publication rights licensed to ACM. 1550-4859/2014/06-ART62 $15.00
DOI: http://dx.doi.org/10.1145/2526687

Table I. Comparison of NAND and NOR Flash by Specifications [Toshiba 2014]

	SLC NAND Flash (×8)	MLC NAND Flash (×8)	MLC NOR Flash (×16)
Density	512 Mbits - 4 Gbits	1Gbit to 16 Gbits	16 Mbits to 1Gbit
Read Speed	24 MB/s	18.6 MB/s	103 MB/s
Write Speed	8.0 MB/s	2.4 MB/s	0.47 MB/s
Erase Time	2.0 ms	2.0 ms	900 ms
Interface	I/O indirect access	I/O indirect access	Random access
Application	Program/Data mass storage	Program/Data mass storage	eXecuteInPlace

time. In such networked systems, a single embedded device is normally equipped with temperature, humidity, pressure, audio, and visual information collection modules, thus WSNs should be able to store, process, correlate, and fuse data originating from heterogeneous sources in the network. With the development of large-scale sensor networking applications, in-network data processing, querying, and aggregation can be employed to reduce data transmission overheads as well as energy consumption. Rather than sending a raw data stream to the basestation outside the network, a sensor node can save a lot of energy by keeping raw, compressed, or summary data in situ locally and then transmitting the processed results when requested [Mathur et al. 2006; Zeinalipour-Yazti et al. 2005], or sending raw data to a nearby clusterhead with more storage for further processing [Abbasi and Younis 2007; Fasolo et al. 2007; Li et al. 2013]. For instance, advanced in-network signal processing algorithms designed for volcanic earthquake detection [Song et al. 2009; Tan et al. 2009] exploit local storage.

However, many popular sensor platforms have very limited storage (e.g., RAM: 4KB–256KB; on-chip flash: 48KB–32MB) which prevents the effective use of the advanced sensor information processing techniques. Magnetic memory systems such as hard disks, on the other hand, are not suitable as secondary storage for sensor platforms due to node size and energy issues [Diao et al. 2007; Nath and Kansal 2007].

Two major forms of nonvolatile semiconductor memories, NAND flash and NOR flash, have emerged as the dominant varieties widely utilized in portable electronics devices in recent years. Table I illustrates various operating and performance characteristics and major differences between NAND and NOR [Toshiba 2014][1]. NOR flash is very similar to a random access memory device and can be easily accessed, while NAND involves a rather complicated I/O interface. Due to its high read performance and eXecute-In-Place (XIP) property, NOR flash is best used for code storage and execution, usually in small capacities. Compared to NOR flash, NAND flash offers faster erase and write times and up to ten times the write endurance; it requires a smaller chip area per cell, thus allowing greater storage densities and lower cost per bit. To this end, NAND flash is optimized for mass data storage. In addition, NAND flash consumes significantly less power for write-intensive sensor applications since energy is power times time. Detailed experimental results have also corroborated this conclusion: in the cases where the storage is designed to be an important part of the sensor networks, the storage device preferred is the cost-effective and energy-efficient NAND flash [Diao et al. 2007; Mathur et al. 2009; Nath and Kansal 2007].

NAND flash memory has unique read/write characteristics and some intrinsic limitations. In particular, a page is the smallest unit for write and read operation, and a block is the smallest unit for erasure at a time. Each block consists of a number of pages and a used page can be rewritten only after erasing the entire block to which the page belongs. Also, when a certain portion of memory is written and erased several

[1]SLC refers to Single-Level Cell, each storing one bit of data; MLC refers to MultiLevel Cell, each storing multiple bits of data.

times exceeding the program/erase cycle limit, that portion can be damaged and data integrity cannot be guaranteed. To minimize the number of block erases and maximize flash service life, normally, the flash memory controller is integrated with a software module called the flash translation layer (FTL) [Kwon et al. 2011] for performance and durability enhancement. State-of-the-art FTL algorithms have been proposed and developed both in industry and academia [Wu et al. 2007, 2010; Lee et al. 2008; Cho et al. 2009; Park et al. 2010; Gupta et al. 2011; Chiao and Chang 2011]. The general design functions and objectives of FTL are to: (1) emulate a normal block device interface, (2) hide the presence of erase operation/erase-before-write, and (3) address translation, garbage collection, and wear leveling. Hence, FTL algorithms are normally complicated that are mostly designed for solid-state drivers (SSDs), USB flash drives, GPS devices, Bluetooth products, and other mobile PCs.

Sensor network applications are highly write intensive, so the internal copying overhead will be huge if we have to copy the valid data to another block before erasing the entire block whenever an update happens. On the other hand, due to the very limited energy budget as well as CPU and memory constraints, these sophisticated FTL techniques are not well suited for wireless sensor platforms. How to design and achieve simple but effective sensor data organization and management systems for NAND flash as storage on sensor devices is the problem that we address in this article.

In sensor network applications, the data gathered by sensor nodes form a time series, namely, each element sampled at a given time. The sampling rate can be periodic in normal situations or bursty when there is a sudden event to track and act upon in real time. Hence, sensed data form tuples $< timestamp, value >$, where timestamps are unique. In addition, a large fraction of queries in sensor applications can be expected to access the time-series sensor data either in time sequence or within given time ranges [Gehrke and Madden 2004; Desnoyers et al. 2005; Nath 2009]. Temporal queries may also need aggregated data, for example, (max, min, avg) of some subset of the data sources at some desired frequency [Trigoni et al. 2005]. For instance, a query can be: "SELECT nodeid, light, temp FROM sensors SAMPLE PERIOD 1s for 10s" or "SELECT MAX(mag) FROM sensors WHERE mag > thresh SAMPLE PERIOD 64ms" or query the average value of temperatures every 5 minutes, etc. [Madden et al. 2005]. To efficiently support such queries, it is very desirable to store a large volume of sensor data in temporal order on local storage and provide a time-based index to achieve efficient access to temporal data.

In order to provide an order-preserving data structure for indexing temporal data, a straightforward solution is to use the well-known B^+-Tree-like structure to maintain the relationships between indexed values and thus allow natural access to ranges, as well as predecessor and successor operations on their key values [Desnoyers et al. 2005]. For instance, time-series sensor data can be organized by the order of timestamps on flash media, and then an index built using these timestamps as keys. This index structure is the same as the bulk-loading B^+-Tree structure [Ramakrishnan and Gehrke 2003]. However, in order to keep the balance property of B^+-Tree, the rightmost index node above the leaf level has to split every time it becomes full as a consequence of the ordered data records being appended, which in turn may cause the splitting of all nodes along the path to the root. Considering the flash write-once characteristic, such index updates will result in huge overheads for the operations on flash media. Other B^+-Tree-based systems such as MicroHash [Zeinalipour-Yazti et al. 2005], μ-Tree [Kang et al. 2007], FlashDB [Nath and Kansal 2007], and LA-Tree [Agrawal et al. 2009], though designed specifically for flash, are all directly indexing on value, not time.

On the other hand, building a log-based index on timestamps implies a large number of unique keys that will lead to a very large memory footprint in FlashDB-like structure

Table II. Comparison of TL-Tree with the State-of-the-Art

	Time-Series Storage	Memory Optimized	Energy Optimized	Index Data Structure	Specially for NAND	Abstraction
YAFFS	No	No	No	Log	Yes	File System
JFFS2	No	No	No	Log	No	File System
Capsule	No	Yes	Yes	Hybrid	No	Object
FlashLogger	No	Yes	Yes	Log	No	Object
FlashDB	No	No	Yes	B^{+}-Tree	Yes	Database
LA-Tree	No	Yes	Yes	B^{+}-Tree Variant	No	Value Index
μ-Tree	No	Yes	No	B^{+}-Tree Variant	Yes	Value Index
MicroHash	Yes	No	Yes	Hash-based	Yes	Value Index
TL-Tree	Yes	Yes	Yes	Tree	Yes	Time Index

[Nath 2009]. Though existing log-structured file systems such as JFFS2 [Woodhouse 2001], YAFFS2 [Aleph One 2005], and FlashLogger [Nath 2009] are good for data management on flash-based SSDs, it is very difficult to use any of them on memory-constrained sensor platforms due to the following reasons: (1) it increases query cost since it does not have tree-index support [Nath 2009]; (2) it consumes huge memory due to large RAM footprint [Mathur et al. 2006]; and (3) it limits the use of flash capacity because the in-memory log structure is not feasible for platforms with scarce memory.

In the face of memory and energy constraints, it is very hard to adopt current schemes for building a time-based index. A novel time-based index structure is required not only to support the storage of time-series sensor data in NAND flash, but also to provide energy- and memory-cognizant operations.

In this article, we propose Time-Log Tree (TL-Tree), a novel time-based index structure for time-series data storage on NAND flash of sensor platforms. In addition to the energy optimization that has been addressed in most existing schemes, TL-Tree is designed to solve the following challenging problems: (1) how to manage the time-series data and construct a time-based index specifically for NAND flash on memory-constrained sensor platforms; (2) how to effectively support both time-based and value-based queries. The performance goal of TL-Tree is to locate time-series sensor data as quickly as possible and concurrently minimize the energy consumption for the page-level file system on NAND flash. Table II presents a summary of comparison between TL-Tree and existing related work. Our contributions include the following.

—We present TL-Tree, which incorporates the following ingredients and properties:
  —*Simple time-log index*. In order to suit the temporal property of sensor data and the typical temporal queries in sensor networks, TL-Tree uses *time-log* to index the sensor data in the file system on NAND flash.
  —*Cascaded index structure*. In order to minimize the usage of main memory, TL-Tree constructs a cascaded index structure that only retains part of the index in main memory. In contrast to those approaches where the whole index structure is stored in main memory, this time-based cascaded design optimizes the usage of both limited memory and large flash storage.
  —*Unbalanced logical tree*. TL-Tree is designed to be an unbalanced tree with small fanout that is optimized for both access cost and energy cost of *lookup* and *insert* operations.
—In addition to effectively realizing both time query and time-range queries, we also propose an easy-to-use indexing structure based on TL-Tree to efficiently support value-based queries for different kinds of sensor applications.

—We analyze the storage bound problem resulting from the limited memory size of typical off-the-shelf sensor platforms and prove a relationship between the memory capacity and flash capacity, if the file size is designed to be an integer multiple of flash page size and log is adopted for indexing the files. We show that the upper bound of NAND flash capacity that can be sustained by a log-indexed structure can be further enhanced by the cascaded tree design, even under a stringent memory constraint.
—We provide an efficient online lookup algorithm to virtually construct the logical tree only at runtime when the query initiates a *lookup* operation. Since this logical tree is in fact not physically stored in the memory, we show that it can achieve considerable memory savings.
—We present a driver for the MicaZ platform configured with raw NAND flash. We also describe our implementation of TL-Tree to demonstrate the practical use of this idea.

Compared to standard B^+-Tree, TL-Tree has the following features: (1) it is an unbalanced tree; (2) tree nodes on flash are used to full capacity; and (3) no other physical storage is needed for the non-leaf tree indexing nodes. Since standard B^+-Tree is normally designed for traditional disk file systems and not for flash, we choose μ-Tree [Kang et al. 2007], which can be classified as a B^+-Tree-based index structure with enhancements designed specifically for NAND flash, to compare the performance in terms of time and energy efficiency. In order to have a fair comparison, we tailored the μ-Tree to time-series storage by: (1) adding a key buffer for the generated index keys, and (2) assigning an empty node to keep the newly arrived keys when the tree node becomes full. This improved tree is called $\mu(n)$-Tree. Experimental results demonstrate that TL-Tree can achieve much better performance for various sensor network applications that include both temporal and value queries. For instance, compared to μ-Tree and $\mu(n)$-Tree, TL-Tree consumes only about 30% of the energy with a 60% improvement in access time for random *lookup* under varied workloads.

The work is organized as follows. Section 2 presents the background and motivation. Section 3 describes the TL-Tree structure and the basic operations. Section 4 discusses the efficient data lookup operations, including the time-based search and value-based search. Section 5 is devoted to performance evaluation, while Section 6 briefly presents the implementation. Section 7 discusses the related work. Finally, Section 8 draws conclusions and presents future work.

2. BACKGROUND AND DESIGN CONSIDERATIONS

In this section, we first describe various characteristics and constraints of sensor platforms and flash memory. Then we discuss some design considerations that motivate our work.

2.1. Characteristics of Sensor Platforms and NAND Flash

2.1.1. Constraints of Sensor Platforms. Memory and storage are very precious resources in many of the start-of-the-art sensor platforms. For example, memory capacity of MicaZ is 4KB RAM, 128KB program flash, and 4KB EEPROM, and a 512KB measurement flash is used to store measurement readings, logs, etc.; TelosB platform is configured with 10KB RAM, 48KB flash, and 16KB EEPROM; the data storage of an advanced wireless sensor node platform, namely, iMote2 is: 256KB SRAM, 32MB SDRAM, and 32MB flash. So the system softwares and applications on sensor devices have to be memory cognizant and concise. A good example is that the core of the TinyOS operating system only takes less than 5KB main memory in use to work under a normal situation [Levis et al. 2005].

On the other hand, under the common assumption that computation is significantly less expensive than communication, in-network processing, including in-network querying, data aggregation, compression, and archival storage, among others, are promising schemes for reducing expensive communication in data-centric sensor network applications. However, all these methods place high demand on the local storage and the limited main memory and storage pose great challenges in the design of such in-network applications.

2.1.2. Properties of NAND Flash. Each NAND flash chip is partitioned into blocks and each block consists of a fixed number of pages. Write and read operations are done in pages, whereas a block is the smallest unit for erase operations. A key constraint of NAND is that writes are one time; once a page is written, it can be updated or rewritten only after erasing the whole block to which it belongs. Erase operations are relatively slow and expensive, and a flash block wears out after a limited number of erases. Since a block is normally several times (e.g., 32, 64, or 128) a page size, it is unacceptable if we have to erase the entire block whenever an update occurs. Due to this fundamental difference between the NAND flash and the traditional block-based storage devices, the traditional index system cannot be directly used on the raw NAND flash. (In the rest of this article, *NAND flash* or *flash* refers to the raw NAND flash.) The other important characteristic of NAND flash is that the energy costs of writing to and reading from a flash are significantly different, that is, 10–20 times greater for writes than reads [Chu et al. 2009]. At the same time, read is multiple times faster than write.

Although NAND imposes stringent constraints, they are the most energy-efficient storage solutions for sensor devices [Mathur et al. 2006, 2009]. A recent study has shown that the new-generation NAND flash is 100-fold more energy efficient than the serial NOR flash present on the Mica platform, and the energy costs of read/write operations are two orders of magnitude less expensive than communication costs [Mathur et al. 2009]. Compared to the limited available memory on sensor platforms, the distinguishing properties, such as large storage capacity, low cost, small size, and low power consumption, make NAND flash very attractive and suitable to be the secondary storage system for sensor networks to optimize in-network designs [Graefe 2009; Mathur et al. 2006, 2009]. When designing the flash storage system on sensor platforms, these properties should be considered in order to optimize the energy cost and the query latency.

2.2. Design Considerations

The characteristics of NAND flash and resource-constrained sensor platforms imply that the design of the flash file system and index structure should take several issues into consideration.

2.2.1. Temporal Data Storage and Index Structure. In wireless sensor devices, different sensors (such as temperature, humidity, pressure, and light, etc.) are used for capturing the environment changes. Normally, they are set to acquire readings periodically, for example, every 10 seconds, in normal situations or aperiodically when some events are detected and the sensing modules are triggered. In either case, the readings can be represented as a temporal data record: $d =< t, v_1, v_2, \ldots, v_n >$, where t denotes the timestamp and v_i refers to the value of the reading from a specific sensor. Here, we assume that the sensor device has n fixed sensors and at the moment when the sensor board is triggered, all sensors are called upon to obtain readings once. In addition to the time sequential observations, sensor data typically has constant size. For example, the temperature and humidity data are represented by 16 bits or 32 bits for each value stored in the database of sensor network applications [Song et al. 2009; Tan et al. 2009].

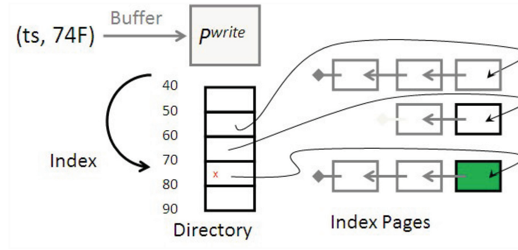


Fig. 1. Index structure of MicroHash [Zeinalipour-Yazti et al. 2005].

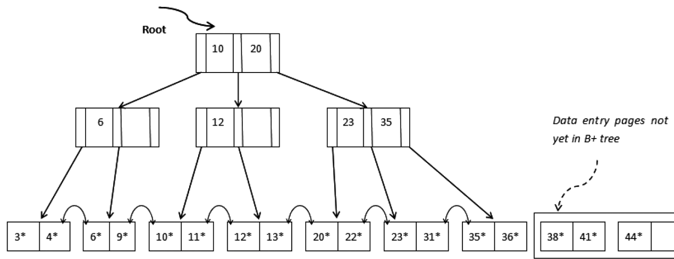


Fig. 2. An example of bulk loading of B^+-Tree.

As the stored-by-timestamp organization always yields completely full data pages on flash media, MicroHash [Zeinalipour-Yazti et al. 2005] proposed to store the data records on flash by the timestamps. In this work, although the sensor data is organized in timestamp order, the index in MicroHash is still ordered according to data value instead of timestamp. Based on a hash scheme, index records are created for each data record when the write cache is full and flushed out to flash, as illustrated in Figure 1. Obviously, such a value index cannot directly support temporal search. To facilitate temporal search, in MicroHash, the index page is designed to contain a fixed number of index records and an 8-byte timestamp of the last known data record. Since each data record needs an index, it is easy to fill up SRAM and when it gets full, index pages have to be forced out to flash by an LRU policy. In order to maintain this value index structure for temporal data storage, several issues need to be considered: (1) the overhead for the maintenance of the index structure such as the directory and repartition strategy; (2) extra CPU power and memory to run the LRU policy, and (3) the cost incurred when it has to scan all pages in flash storage for a query-by-time.

The other way to build the index for temporally ordered sensor data is to directly use timestamp as the key and create a tree-based index data structure, for instance, B^+-Tree. Since the data are organized in temporal sequence, this kind of index structure build-up process is the same as the third step of the bulk-loading algorithm for creating a B^+-Tree index on an existing collection of data records [Ramakrishnan and Gehrke 2003]. In order to keep the balance of the tree, the rightmost index node above the leaf level has to be split every time as the ordered data records are inserted that leads to a full index node. Figure 2 illustrates the index tree for bulk loading of a two-order B^+-Tree. As splits only occur on the rightmost path from the root to the leaf level, if we store this index structure on flash, as the tree grows, the previous non-leaf index nodes along that path will need to update the index pointer values (such as the page-id of a flash page), as shown in Figure 3. Thus, these updates will lead to large rewrite costs and garbage due to the specific write-once property of NAND flash. Besides, the frequent erase operations will shorten the flash lifetime.

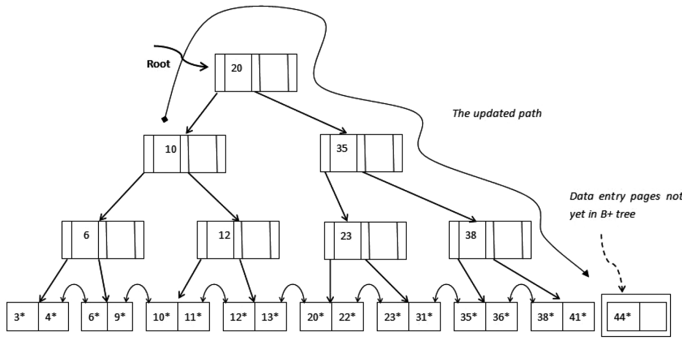


Fig. 3. Example of bulk-loading update of B^+-Tree.

Table III. Terminology

Block	Contains a few pages, is the smallest unit for erase operation.
Page	Is the smallest unit for write and read operation.
File	An abstraction to store, retrieve and update a set of sensor data records. It can be defined as by the page size of flash.
Log	Each file has a log that can be used for identifying that file and positioning the file location.

In summary, the flash storage system should cater to the needs of time-series sensor data, in order to best support time and time-based range queries, so that the expensive flash I/O operations are avoided or minimized. How to build an index that can alleviate the impact of the data pattern and, at the same time, effectively support both time-based and value-based queries is a problem we consider in this work.

2.2.2. Resource-Aware Design. In order to reduce the main memory footprint, the file system should maintain most of its index structure on flash [Nath 2009]. However, the NAND flash has a nonrandom read characteristic and longer access time [ATmel 2001; Samsung 2014; Toshiba 2003]. Therefore, it is better to place all or at least part of the index in the main memory in order to improve the performance. But, the memory capacity of typical sensor platforms is very limited, thus, for a flash with large storage capacity, keeping all the index in memory is impractical. Considering a log-structured indexing system on flash with log and file defined as in Table III, we have the following result.

THEOREM 2.1. *Given K bytes of main memory and M bytes of NAND flash memory, if the file size is L bytes where L is designed to be an integer multiple of the NAND flash page size due to the special read and write characteristics, for a log-based indexing structure for this file system, the largest number of logs that can be stored in the main memory is* $C = \lfloor \frac{4*K}{\log(M/L)} \rfloor$.

PROOF. Suppose each file needs a log, the number of logs to be stored in the main memory is : $Q = M/L$. Since each log must at least contain the critical information for identifying that file and an address to distinguish the specific location of that file in the whole address space, the smallest *log* can be defined as: $log =< key, address >$, where *key* refers to some unique data property and *address* denotes the location. According to information theory, if we visit Q logs with equal probability, we need $W = \log Q$ bits for the address space. Since each key in the log is unique, we have the same number of keys as the addresses. Thus, it at least needs the same number of bits for addressing keys. That means the total bits for one *log* is at least $2 * W$, so the maximum number of logs

that can be stored in a K-bytes main memory is $C = \lfloor (K * 8)/(2 * W) \rfloor = \lfloor 4 * K/W \rfloor = \lfloor 4 * K/\log(M/L) \rfloor$. □

For example, a sensor platform has 128KB main memory and 128MB flash, if the file size is 512 bytes, according to the preceding theorem, it can store at most 29127 *logs* ($C = \lfloor \frac{4*128K}{\log(128M/512)} \rfloor = \lfloor \frac{4*128*1024}{\log(128*1024*1024/512)} \rfloor = 29127$). On the other hand, for a 128MB NAND flash with 512 bytes/page [Toshiba 2003], if one page is to be a file, it will need 256K *logs* to represent the file information. Obviously, it is impossible to place all logs in the main memory, due to the limited available memory of sensor platforms.

Given a certain amount of main memory, how to design a feasible index structure to sustain a large flash storage is another key problem that will be addressed in this work.

2.2.3. Energy-Efficient Access. Efficient access to sensor data from a wireless sensor network has been the focus of systems such as TinyDB and Cougar [Yao and Gehrke 2002; Madden et al. 2005], where users input queries at the server in a simple, SQL-like language that describes the data they wish to collect and how they wish to combine, transform, and summarize it [Gehrke and Madden 2004]. As pointed out by Gehrke and Madden [2004], those queries are high-level statements of logical interests, thus the database system or file system on sensor nodes should be able to collect, store, and access satisfied data. Although query language design and query optimization are important problems, they are out of the scope of this work. In this article, we consider the same queries as defined in MicroHash [Zeinalipour-Yazti et al. 2005]. They are listed next.

Time-Based Equality Query $Q(t, t_1)$. Find sensor data records with timestamp $t = t_1$.

Time-Based Range Query $Q(t, t_1, t_2)$. Find sensor data records with timestamps in the range of $[t_1, t_2]$.

Value-Based Equality Query $Q(v, v_1)$. Find sensor data records with value $v = v_1$, for a specific sensor attribute.

Value-Based Range Query $Q(v, v_1, v_2)$. Find sensor data records with values in the range of $[v_1, v_2]$, for a specific sensor attribute.

Suppose sensor data records are stored on flash at the time when the sensor node senses the environment. In order to support efficient assess to the data of interest, we need to create index entries pointing to the respective data records on flash. As discussed before, sensor platforms have very limited energy budget. Therefore, the design of the index structure should not only be memory aware, but also be able to ensure simple and energy-efficient index operations, in order to achieve energy-efficient data access on flash.

3. OVERVIEW OF TL-TREE

In this section, we first discuss the logical and physical structure of TL-Tree. The basic operations such as *load-in*, *insert*, and *delete* will then be presented. Finally, we address the issues on how to extend the standard TL-Tree to support value -based indexing.

3.1. Basic Components of TL-Tree

3.1.1. Logical Structure of TL-Tree. Suppose a sensor device is installed with a raw flash that has $B = B_1, B_2, \ldots, B_m$ blocks, each of which contains a number of consecutive pages, that is, the smallest write/read unit. The platform is associated with a set of on-board sensors, such as temperature, humidity, etc. Every time when the readings are obtained, a data record is generated as $d = <t, v_1, v_2, \ldots, v_n>$, where t denotes the timestamp and v_i is the value observed by the ith sensor. Here, we assume that the readings from different sensors are acquired simultaneously with constant size at

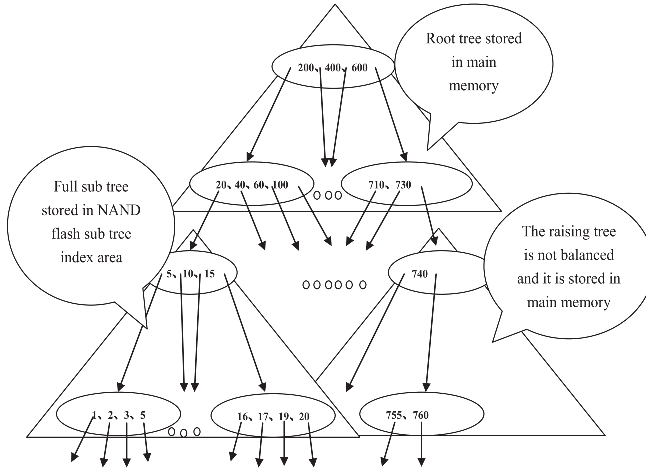


Fig. 4. The logical structure of TL-Tree.

the moment when the reading event is triggered. All these records will be organized in order by timestamps on the consecutive available pages of flash. Let one page be abstracted as a file, the data file size equals to the size of one flash page.

Let one file have the relevant log information. Since all sensor data are organized on flash by timestamps and timestamps are monotonically increasing as the time goes on, we set *log* to be a two-tuple: $log_i = <time, address>$, where *time* is the temporal tag associated with the last timestamp of the sensor data stored in the ith file and the *address* refers to the flash page address where the data belongs. If each block has n pages, then we will have $m * n$ logs. As analyzed in Section 2.2.2, it is impossible to place all logs in the main memory for directly indexing the data. Here, we will design a tree-based index structure for indexing the log information when the sensor readings are stored on flash. This structure is called temporal-log tree (TL-Tree).

According to the structure and where the index nodes are stored during the construction course, TL-Tree is composed of three parts, namely, root tree, raising tree, and subtrees, as shown in Figure 4, where *root tree* and *raising tree* are stored in the main memory and subtrees are stored on flash. The entries on the leaves of both raising tree and subtrees contain two fields: a temporal index key and the page address of data records on the flash. Each entry of the root tree also contains two fields: an index key and an address pointer to the page address of a subtree. So in general, all entries have the same structure as the log: $< key(time), address >$.

Entries are added to a TL-Tree in the following two ways: (1) the key index of the temporal log for each new page is added to the raising tree, if the raising tree is not full; (2) once the raising tree becomes full, it will be pushed out to flash and an index entry is added to the root tree. This process proceeds until the root tree is full.

Since a TL-Tree need not be balanced, no split is required as data records are generated and stored in temporal sequence in flash. For the previous example of the bulk loading of B^+-Tree, the TL-Tree for the same data records of Figure 2 is shown as in Figure 5. As new data records are added to the system, only the raising tree grows, as illustrated in Figure 6. If the raising tree is full, the root tree may need to grow also, as in Figure 7. But the growth of the root tree will not cause the overhead of splitting and changes of the relevant TL-Tree index nodes along the path, as in the bulk-loading B^+-Tree.

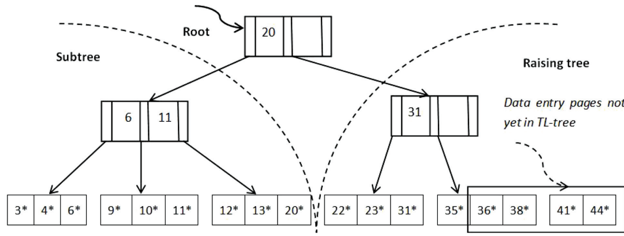


Fig. 5. The TL-Tree (compared to Figure 2).

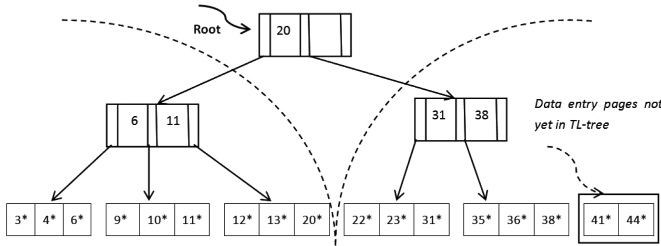


Fig. 6. TL-Tree update (compared to Figure 3).

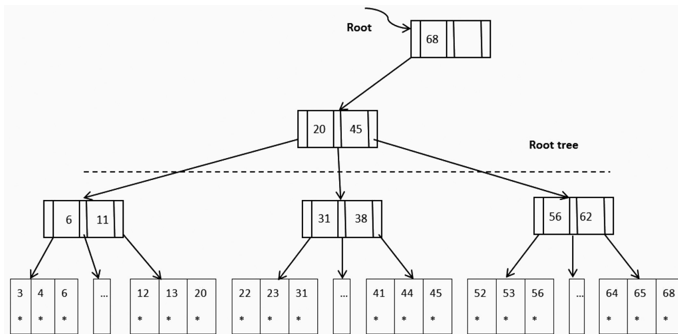


Fig. 7. TL-Tree update and root tree growth.

3.1.2. Physical Storage of TL-Tree. When the system is in operation, the main memory is divided into two areas that include a TL-Tree area and an area for other applications, as shown in Figure 8(a). In order to optimize the use of the memory in the TL-Tree area, only the time logs of the leaves of the root tree and the raising tree are stored in the main memory, as shown in Figure 8(a), while the time logs of other full subtrees are stored in the NAND flash. We denote the in-memory area where the time-log index of the root is stored as *root-tree index array* and the place the time-log index of the raising tree is stored as *raising-tree index array*.

Since the main memory is very limited, in order to avoid the overhead introduced by the index nodes, we design an efficient online algorithm (discussed in Section 4) to build the logical tree structure when needed during the runtime. Hence, there are no physical internal tree nodes stored in the memory.

Suppose the main memory size is C, the number of logs in the *root-tree array* is S_r and in the *raising-tree array* is S_s, then we have $S_r + S_s \leqslant C$. Since the total number of logs that the whole index tree may have is $S_w = S_r * S_s$, according to the *Geometric-Arithmetic Mean Inequality Theorem*, when $S_r = S_s$, S_w will have the biggest value.

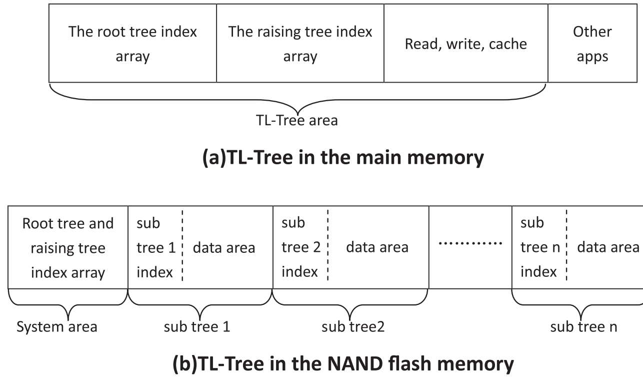


Fig. 8. The storage structure of TL-Tree.

In other words, when $S_r = S_s$, C has the minimum value. So if we want to minimize the space usage of the main memory, the number of root-tree logs should be equal to that of the raising tree. For example, for an 8G NAND flash with 4M pages, if we store one file per page, then we will have 4M logs. In this case, we can build a 2K/2K root/raising tree for indexing the 4M logs, which only needs 4K logs in-memory space. Since each log needs 6 bytes to denote the time and the address[2], the total memory used for indexing an 8G NAND flash is 24K bytes. This means that, using this cascaded time-log index tree, MicaZ is able to sustain an 8G NAND flash, which is enough for supporting most in-networking applications for sensor networks. Compared to other existing log-structure indexes [Lim and Park 2006; Aleph One 2001; Woodhouse 2001] that store the whole log information in the main memory, TL-Tree can effectively reduce the use of the main memory and at the same time support a large NAND flash capacity.

In NAND flash, the storage area is divided into the *system area* and the *subtree areas*, as shown in Figure 8(b). The *system area* stores the corresponding TL-Tree area information from the main memory when the system is turned off normally. So when the system is turned on, this area should be loaded first. For simplicity, the system area is designed to be one block of the NAND flash in our system. Each *subtree area* is constituted of a *subtree index area* and a *data area*. The subtree index area is used to store the subtree index logs when the corresponding raising tree is full. The data area is used to store the temporal sensor data.

3.2. Basic Operations of TL-Tree

In this section, we first describe the basic *load-in*, *insert*, and *delete* operations. The time-based and value-based *lookup* will be discussed in Section 4.

3.2.1. Fast Load-in. When the system is turned off, the in-memory index information will be stored back to the flash system area, so when the system is turned on, it can load these information to the main memory directly. Because it does not have to scan the flash pages like the other log-structured file systems [Lim and Park 2006; Aleph One 2005], this structure will save a lot of time and power.

If the system is turned off due to an abnormal situation such as an unexpected power failure, these in-memory index information cannot be written back to the flash.

[2] $\log_2^{4M} = 22$ bits are needed to address all 4M pages, and since MicaZ uses 8-bit CPU, 24 bits (3bytes) are needed for a page address, and the other 3bytes for the time of a file.

Table IV. The *Insert* Operation

Input: log $E < time, address >$ log number stored in raising tree array: $SubNum$ log number stored in root tree array: $RootNum$
1: Let $RaisingTreeArray[SubNum] = E$; 2: $SubNum = SubNum + 1$; 3: **if** $SubNum$ is bigger than log num. limit of one subtree 4: **then** 5: remove the $RaisingTreeArray$ to flash; 6: $SubNum = 0$; 7: $RootTreeArray[RootNum] = E$; 8: $RootNum = RootNum + 1$; 9: **if** $RootNum$ is bigger than log num. limit of root tree 10: **then** call Delete[*offset*].

In this case, the root tree can be reconstructed by scanning each *subtree index area* in subtree areas, and the raising-tree index can thus be reconstructed by scanning the corresponding subtree data area.

3.2.2. Temporal Sequence Insertion. Due to the specific write characteristic of NAND flash, a write cache is used to load the data before they are written to the flash. Here the write cache has to be at least twice the file size so as to improve the write efficiency. When the cache is full, the system will first move the sensor data, including the value and the time, from the cache to the data area of the flash, and then start the *insert* operation for the TL-Tree.

For the *insert* operation, a *log* is first generated and then inserted into the raising sub-tree array if the raising tree is not full. Once the raising tree is full, it will move the index information of this raising tree to the corresponding subtree index area of flash, and add one log to the root-tree index array. If the root tree is full, it will then call the *delete* operation that will be discussed next to delete the oldest subtree; otherwise, the insert operation is complete. The detailed steps of the *insert* operation are illustrated in Table IV.

3.2.3. Oldest Subtree Deletion. In order to simplify the operation and keep the logical structure, the root-tree index array is stored in a round-robin queue with an *offset* variable pointing to the location of the oldest key. Initially, $offset = 0$. When the flash is full, we need to first delete the oldest subtree area from the flash memory and then delete its corresponding index in the root tree, and set $offset = offset + 1$.

For instance, in Figure 9(a), the root-tree of a TL-Tree has $fanout = 4$ and $level = 2$, where $offset = 0$. Now, the flash memory is full and we need to delete the oldest subtree pointed to by the key 10. What we shall do is to delete key 10 in the root-tree array in the main memory and set $offset = offset + 1$. If at this time a new subtree is generated with index key of 85, then 85 will be inserted to the root tree at the physical location of the deleted key 10, as shown in Figure 9(d). But since $offset = 1$, from a logical point of view, the tree keeps the shape as before the deletion but shifts one location to the left, as shown in Figure 9(c).

In order to avoid out-of-place updates, the subtree size is designed to be an integral multiple of the block size of the flash, since the whole block is expected to be deleted whenever needed. This is a necessary and sufficient condition for out-of-place update avoidance of TL-Tree. Otherwise, for instance, if the subtree size is one-and-a-half times the block size, then every time when we erase the oldest subtree, we will have half a block of valid data that need to be moved. As we know, each block normally contains

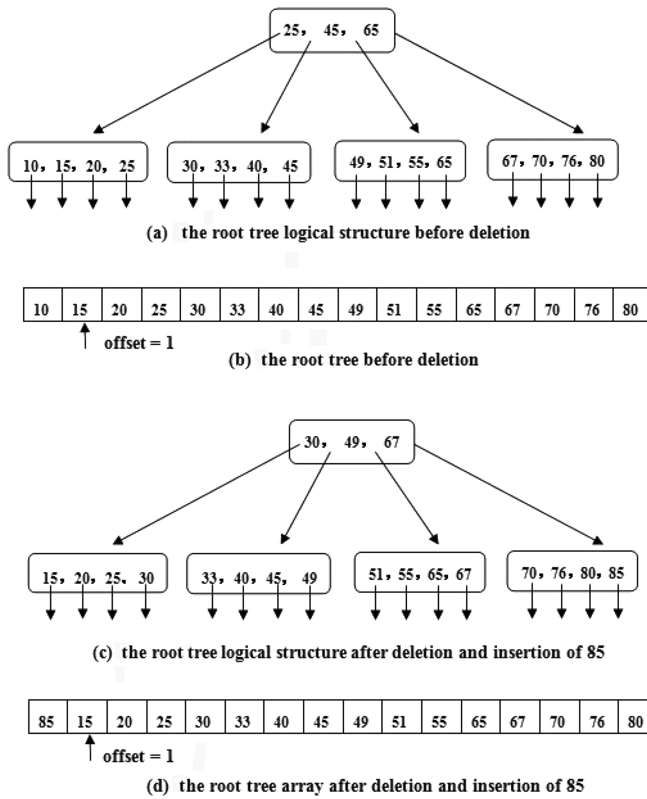


Fig. 9. A running example for deletion and insertion.

an even number (2^n) of pages and several pages are used for storing the subtree index[3]. Since the page is the smallest write/read unit, we should set the file size to be an integral multiple of a page. After taking out the pages for subtree indexing, we cannot always guarantee the full utilization of the pages on a block if the file size is some integral multiple of a page (for the previous example that has 32 pages a block and 4 pages for a subtree index, if the file size is to be 3 pages, in this case one page of this block will be left empty since 28 cannot be evenly divided by 3). To this end, we simply set the file size of TL-Tree to equal one page. Together with the subtree setting condition, it is easy to see that TL-Tree will never need to implement out-of-place updates and a garbage collection strategy; at the same time, it can fully make use of the flash capacity for data storage.

3.3. Value-Based Indexing Structure

In sensor applications, queries may arise that are not time based but value based, for instance, “SELECT nodeid, nestNo, light FROM sensors WHERE light > 400 EPOCH DURATION 1s”, or “SELECT AVG(sound) FROM sensors EPOCH DURATION 10s”, etc. [Madden et al. 2005]. It is easy to extend the basic TL-Tree index structure to accomplish value-based queries.

[3]How many pages are needed for the subtree index is dependent on the root/raising tree in the main memory, for example, for the 2K/2K example in Section 3.1.2, if one page is 512 bytes, then there are 4 pages used for a subtree index on a subtree area in flash.

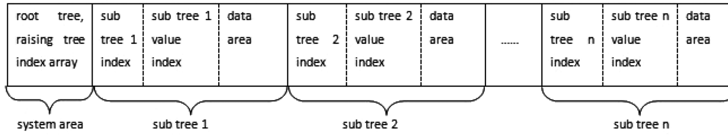


Fig. 10. The value-based subtree in flash.

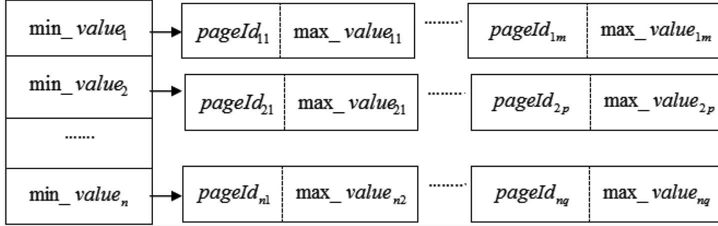


Fig. 11. The value index for a subtree.

We combine the temporal index and the value index by exploiting the observation that pure time-independent value-based queries are unrealistic in sensor applications; rather, a value-based query asks for data that matches a condition over the value in a particular time window. Suppose the flash size is M, one page size is P, and one subtree size is S. First, we assume that there is just one type of sensor data, such as, temperature data, stored on the flash; we will discuss the solution for multiple data types next. For each page, *max_value* (*min_value*) is used to record the maximum (minimum) value of all data stored in this page, and $pageId$ refers to the address of this page (from 1 to $\lceil M/P \rceil$). For each subtree, through the process of establishing the subtree index introduced in Section 3.1.1, at the same time we construct a value index by storing the Max/Min properties in the second page of the subtree while keeping the temporal index in the first page. Thus the new TL-Tree in flash memory would be changed to the structure shown in Figure 10.

The details of the value index structure are shown in Figure 11. Every element of this structure is composed of a tuple-$\langle key, value \rangle$, where *key* is the *min_value* of all the data in each page and the *keys* are ordered in ascending order. The *value* is a linked list that contains the $pageId$ and *max_value* of the relevant pages. In addition, this linked list is ordered by $pageId$ in ascending order. This means all pages that have the same *min_value* will be located within the same linked list.

4. EFFICIENT LOOKUP OPERATION

The purpose of *lookup* is to search for the sensor data that is queried by the applications. In order to improve the access performance for readings that possess temporal locality, a read cache is used in the main memory and the simple FIFO paging algorithm is adopted. Initially, the cache is empty and, once *lookup* obtains the location of the first wanted page in NAND flash, this whole page is moved to the read cache.

4.1. Search by Time

According to the place where the sensor data are stored, the possible locations for the queried sensor data or the corresponding index are *read cache*, *raising tree*, and *root tree*. The detailed steps for the retrieval of the data are illustrated in Table V.

If the data is found in the read cache, this operation returns the value; otherwise, it is required to compare the queried time T with the earliest time of the raising tree. If T is bigger, the *raising-tree index array* is searched. If the queried time log is found, it

Table V. The *Lookup* Operation

Input: the queried data time of T
1: Search the *read cache*;
2: **if** have T **then return;**
3: compare T with the earliest data time of raising tree;
4: **if** T is bigger **then** // T is in the raising tree
5: $address = TS[T, X, 0, subnum, RasingTreeArray]$;
6: move *address* page from flash to read cache;
7: **return**;
8: $subadd = TS[T, X, offset, rootnum, RootTreeArray]$;
9: Load the *subadd* pointed subtree to memory and call the search algorithm again to obtain data file address, and then move that page to read cache.

does not need to visit the flash any more. As discussed in the previous section, there does not exist a physical tree, so at runtime the operation will call the *TreeSearch algorithm* to search for the *raising-tree index array* in the main memory in order to obtain the flash page address that contains the queried sensor data. Here *TreeSearch* is an efficient algorithm that can imitate the logical index tree construction.

If T is smaller, which means the queried log is not in the rasing tree, we have to search the *root-tree index array* in the main memory to find the subtree index page address obtained by the *TreeSearch algorithm*. Upon the receipt of the address, since the whole tree index is stored in the flash, it can first move the root level of the subtree from flash to the main memory and then compare the time logs to obtain the page address in the next level of the tree. This process will proceed until reaching the leaf level of the subtree index to eventually get the page address of the sensor data.

The key point of the *TreeSearch* algorithm is to compare the queried time (index key value) with the keys in the virtual logical index tree, from the root node to the leaf-level nodes of that tree. The output is the corresponding flash page address. As shown next, this special property can save a lot of main memory, a highly limited resource on sensor platforms. The details are illustrated in Algorithm 1.

THEOREM 4.1. *Let us consider the root tree (or the raising tree, a loaded-to-memory subtree) of a TL-Tree, if the fanout number is X, compared to the method that stores the full index tree in main memory, the TreeSearch algorithm can reduce the space by about* $\frac{1}{X}100\%$.

PROOF. Suppose the tree has height of H, if we keep the whole index tree in main memory, the space utilized for storing the tree will be $\sum_{i=1}^{H} X^i = X * (1 - X^H)/(1 - X)$. Since TL-Tree only needs to store the logs of the leaf-level nodes, the space cost of TL-Tree will be X^H. Therefore, the total memory space will be reduced by $\frac{X*(1-X^H)/(1-X)-X^H}{X*(1-X^H)/(1-X)} = \frac{(X-X^H)/(1-X)}{X*(1-X^H)/(1-X)} = (\frac{1-X^{H-1}}{1-X^H})100\% \approx (\frac{1}{X})100\%$. □

When the fanout is small, the benefits are considerable. For instance, if the fanout is 4 and the height is 6, which is the root tree we use in the TL-Tree in the MicaZ for 128M NAND flash, the total memory for the index structure is reduced by about 25%.

Except for the index array $IndexTree$ itself, there are four arguments to the algorithm. The additional space for the data retrieval of the logical TL-Tree is six that is used to store the local variables of the algorithm. Therefore, the space complexity of *TreeSearch* is $O(1)$. Also, it is easy to see that the time complexity of this algorithm is $O(H * X)$, $H = \lfloor \log_X N \rfloor$, namely, the height of the tree.

Table VI. The *Value Lookup* Operation

Input: the value range and time range of the queried data (v_1,v_2,t_1,t_2)
1: start address $sAddr \Leftarrow search\ by\ timestamp\ t_1$;
2: end address $eAddr \Leftarrow search\ by\ timestamp\ t_2$;
3: start subtree $Id_1 \Leftarrow \lceil sAddr * P/S \rceil$;
4: end subtree $Id_2 \Leftarrow \lceil eAddr * P/S \rceil$;
5: **while** $Id_1 <= Id_2$ **do**
6: $ValueTreeSearch\ [index - value, sAddr, eAddr, v_1, v_2]$;
7: $Id_1 \Leftarrow Id_1 + 1$;
8: $sAddr \Leftarrow$ the first pageId in subtree(Id_1);
9: **end while**

ALGORITHM 1: *TreeSearch* Algorithm (TS)

Input: queried time T, fanout X, *offset*, number of elements in the array N, the array $I[\,]$.
Output: page address A in NAND flash.
$H = \lfloor \log_X N \rfloor$; $j = H$; $p = 0$; /* j: height; p: pointer to the node containing T
repeat
  $j = j - 1$; $i = 1$; /* i: current key comparison location
  repeat
    if $(T <= I[(offset + p * X^{(j+1)} + i * X^j - 1) mod N].time)$;
    then
      $p = i$; break; /* find the key
    end
    $i = i + 1$; /* not find the key
  until $i < X$ /* inside a node;
until $j > 1$ /* for non leaf nodes;
$k = 1$; /* k: current comparison location in the leaf node
repeat
  if $(T <= I[(offset + k + (p - 1) * X^j - 1) mod N].time)$;
  then
    $A = I[(offset + k + (p - 1) * X^j - 1) mod N].link$; /* find the key, return the address
    return A;
  end
  $k = k + 1$;
until $k < X$ /* if k is within a node;

4.2. Search by Value

We presume that the value-based queries are specified as follows: query the data x in the time range $[t_1, t_2]$ $(t_2 > t_1)$ whose value is greater than v_1 and less than v_2 $(v_2 > x > v_1)$. Detailed retrieval steps are shown in Table VI and Algorithm 2.

For time range $[t_1, t_2]$, we first obtain the corresponding page address, that is, $sAddr$ and $eAddr$, calculated by the *TreeSearch* algorithm. We hence query data in a reduced space from page with Id from $sAddr$ to $eAddr$, that is, from subtree $Id_1 = \lceil sAddr * P/S \rceil$ to subtree $Id_2 = \lceil eAddr * P/S \rceil$, instead of the whole flash. Then we can use the value index in subtrees from Id_1 to Id_2 to access the target data. We infer that if one page has the eligible data, its min_value must be less than v_2 and max_value must be greater than v_1. We traverse the linked list for each min_value key if it is less than v_2, to find a page that contains data that satisfy the condition. Once a page is found, it will be read into the cache and the lookup operation continues for eligible data in the following pages.

As we know, the sensed data can be of different types, such as temperature, humidity, motion, etc. Although in the preceding solution we only store/index a single

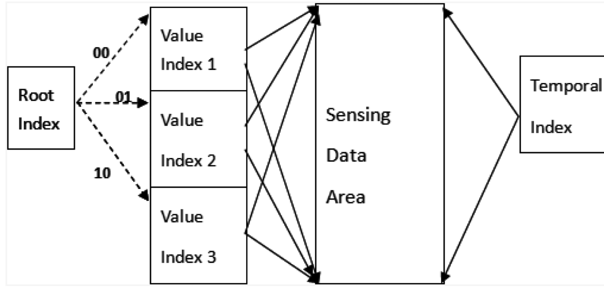


Fig. 12. The complete value and temporal index structure.

ALGORITHM 2: *ValueTreeSearch Algorithm*

Input: value-index of subtree, $sAddr$, $eAddr$, v_1, v_2 .
$i = 1$;
repeat
  if ($min_value_i <= v_2$);
  then
    $j = 0$;
    repeat
      if ($sAddr <= pageId_{ij} <= eAddr) \wedge (max_value_{ij} <= v_2$);
      then
        read the page with $pageId_{ij}$ to read cache;
      end
      $j = j + 1$;
    until $j < end_of_page_list$;
  end
  $i = i + 1$;
until $i < n$;

value, it is easy to extend this structure to support multiple-value indexing. For instance, we can store different sensed data in flash by the type as in sequence [*temperature*, *humidity*, *motion*] if they keep the same sensing period, or just by temporal order for different sampling intervals. Then we can build a simple virtual root index to point to the different types. That is, in this specific example, we only need 2 bits to indicate the temperature, humidity, and motion data, as shown in Figure 12.

The flash is thus divided into three parts: temporal index, value index, and sensed data part. The value index part uses type index area to first differentiate sensor data types, where each type index structure is the same as in Figure 11, but with different values. In this example, area one points to temperature, two points to humidity, and area three points to motion. Every part can index all the data, similar to the temporal index. Given a value-based query, the index system will first choose one index area in the value index part according to the query type and then proceed with the retrieval process as described before.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of TL-Tree in terms of energy efficiency and access latency. We build an emulator for a MicaZ/Iris sensor board configured with a customized 128MB raw NAND flash chip (512 bytes per page and 16K bytes block size) as in Mathur et al. [2009], the energy cost and read/write time adopted here for the main memory and flash are the same as in ATmel [2001], Mathur et al. [2009], and

Table VII. Operation Cost on Memory and Flash

Symbol	Definition	Value
T_m	main memory read time	45 $ns/byte$ [ATmel 2001]
T_r	main memory write time	2 $\mu s/byte$ [ATmel 2001]
T_n	access time of NAND flash	$20\mu s/page$ [Toshiba 2003]
T_p	NAND flash write time	$200\mu s/page$ [Toshiba 2003]
E_m	main memory read energy	$0.26\mu J/byte$ [Mathur et al. 2009]
E_r	main memory write energy	$4.3\mu J/byte$ [Mathur et al. 2009]
E_n	NAND flash read energy	$2.05\mu J/page$ [Mathur et al. 2009]
E_p	NAND flash write energy	$4.61\mu J/page$ [Mathur et al. 2009]

Toshiba [2003]. Table VII illustrates the symbols and relative cost values used for the evaluation.

The emulator is able to count the number of *lookup (read)* and *insert (write)* operations performed on main memory and NAND flash so that the energy and latency cost can be calculated accordingly. For simplicity, the subtree index is designed to be equal to one page size of the flash. Let M denote the NAND flash size, L be the data file size, according to the analysis in Theorem 2.1, each time-log size is $S = 2 * W = 2 * \log(M/L)$. If we denote P as the flash page size, then for each page, the total number of logs is $\lceil P/S \rceil$. Since the size of each subtree index is one page, each subtree index has $\lceil P/S \rceil$ logs. Therefore, the root tree has $\lceil M/(\lceil P/S \rceil * L) \rceil$ subtrees. For example, for the preceding system, if each file contains 512-byte data, each subtree will have 64 pages $(512/(2 * \log(128M/512)))$, and the root tree has 4K subtrees $(128M/(64 * 512))$. We adopt these settings for the experiments conducted in this section.

5.1. Energy Consumption Calculation

The target data for a query can be located in three places: (1) read cache, (2) raising tree, and (3) NAND flash, so we define five variables: *count_read_cache*, *count_write_cache*, *count_rasing*, *count_root*, $count_flash$, and *count_other* to describe the number of comparison times at each place. The value of each variable can be calculated as follows.

Case 1: Cache hit. *Cache hit* means the queried data is located in the read cache, thus *count_read_cache* = 2 since we only need to compare the data value to the minimum and maximum key value of the read cache. In this case, other variables are equal to zero. (Note here, we only consider the comparison to tell whether the data is in the read cache and do not count the energy consumption for further locating steps.)

Case 2: Raising-tree hit. The data found in the raising tree means we have confirmed it is not in the read cache. There are two possibilities: (1) if data is smaller than the first key in the cache, then *count_cache* = 1; (2) otherwise, we need to compare with the first and last keys in the cache, and *count_cache* = 2. The *count_rasing* equals the comparison times by the execution of the *TreeSearch* algorithm for the raising tree, $count_flash$ is 1 (locating the data address on flash pointed by the raising-tree leaf node) and *count_write_cache* is 512 (to read and load the corresponding page containing the target data from flash to read cache).

Case 3: On flash. If we figure out that the data is not in the read cache and raising tree, here the *count_read_cache* is 1 or 2 and the *count_raising* is 1 (for comparing to the smallest key of the raising tree), then we need to search the root tree area to find the subtree index that points to the target data page. In this process, *count_root* refers to the comparison times by *TreeSearch* algorithm performing over the root tree. If we find the relevant subtree, the subtree is read to search cache so that $count_flash$ is 1 (the energy is the same as for the read cache since both are in RAM) and the *count_write_cache* is 512. Next, we need to search the subtree to find the page address of the target data,

and *count_other* is for recording the comparison times by the *TreeSearch* algorithm. Finally, the target page is brought into the read cache; until now *count_flash* is 2 and *count_write_cache* is $512 * 2$.

In general, the aforesaid calculation for energy consumption of a query can be summarized as follows.

$EnergyCost = count_read_cache * size_of(data) * E_m + (count_rasing + count_root + count_other) * size_of(data_log) * E_m + count_flash * E_n + count_write_cache * E_r$

Let us look at the example shown in Figure 6, and the query is $Q(t, 36)$. We assume that the data timestamp in the read cache is from 10 to 13, then we need to compare twice to figure out that the queried data is not in cache. After comparing with 22, that is, the smallest key in the raising tree, it is confirmed that the key can be found in the raising tree. So, the *TreeSearch* algorithm is called to find the location of $Key = 36$ (it needs 4 comparison times in this example). Finally, the flash page is read and loaded to cache. The total energy consumption is $EnergyCost = 2 * 16byte * Em + 1 * 8byte * Em + 4 * 8byte * Em + 1 * En + 512 * 4.3uJ = (32 + 8 + 32) * 0.26 + 2.05uJ + 2201.6uJ = 2222.37uJ$

Similarly, the time cost for each query can be calculated using the following formula.

$Timecost = count_read_cache * size_of(data) * T_m + (count_rasing + count_root + count_other) * size_of(data_log) * T_m + count_flash * T_n + count_write_cache * T_r$.

5.2. Temporal Data and Query Settings

In this experiment, we generated two sets of synthetic data with different temporal patterns for data arrival intervals: one is with constant period to mimic the sensor data sampling rate under a normal situation, and the other uses a Poisson process to model the event-based sensing environment. The constant period is set to 1s. λ is set to 0.5 and 0.02 to distinguish two scenarios where the expected value of the interval between two arrivals is 2s and 50s, respectively.

We conducted tests with two different benchmark workloads: random-time lookup and time-range lookup, to evaluate the TL-Tree for the previous data arrival patterns. Each operation specifies a 6-byte integer key and the number of data items is 8MB. In the figures, each value is the average of 100 experimental results.

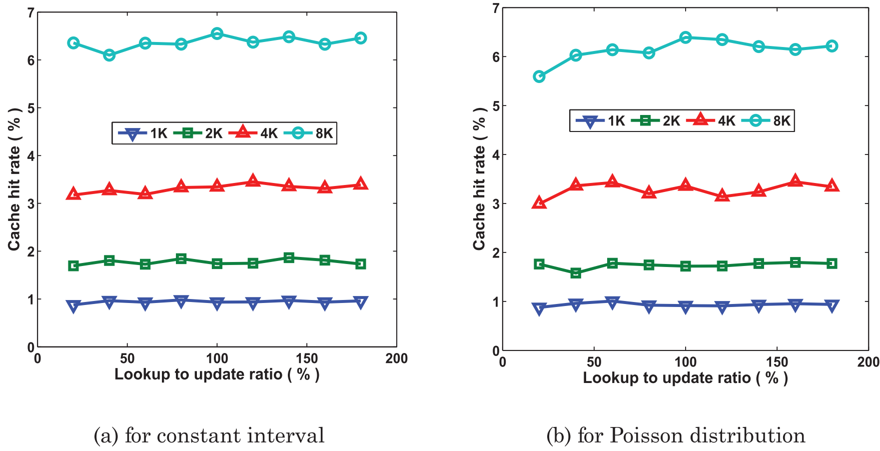
For a random lookup operation, like the experiments conducted in Agrawal et al. [2009], the workload consists of a sequence of a random mix of updates (*insert* operations) and lookups with a given Lookup-To-Update ratio (LTU) that ranges from 20% to 180%. The queried time points are randomly distributed in the temporal range of all sensor data records.

For a time-range lookup, the queries are generated to read all the records with timestamps within a time window $[t_i, t_i + length]$, where t_i is uniformly distributed in the index range. Since all sensor data are temporal, the *length* refers to the time range of the search and is varied from 2000 to 30000. Here, we test the ranges incremented with 100 in [2000, 10000], and then increase with 5000 in [10000, 30000].

5.3. Evaluating Cache Impact

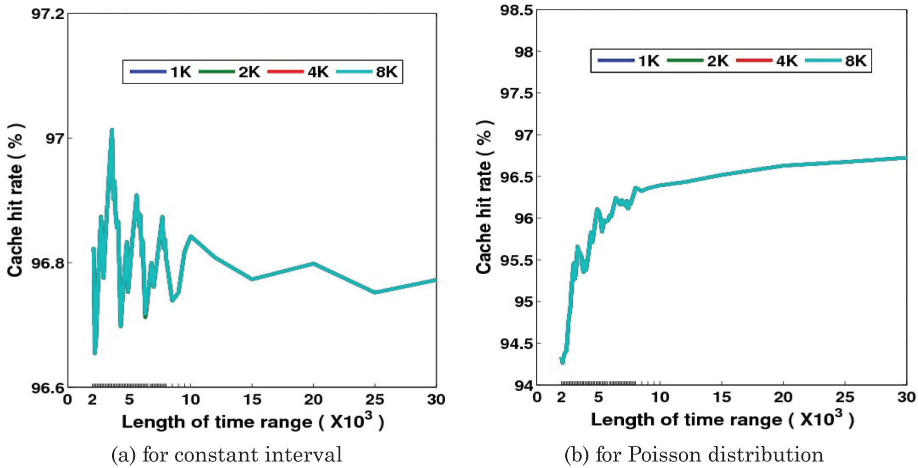
5.3.1. Cache Hit Rate. In TL-Tree, our design uses FIFO as the page replacement algorithm for the read cache since it is a low-overhead algorithm that requires little bookkeeping work ($O(1)$ additional amount of work per page replacement), and it is not a marking algorithm so that it does not need to pay an extra maintenance cost for marking the pages in cache. Although the paging strategy is out of the scope of this work, we still conducted experiments to show the cache-related properties in the context of the TL-Tree structure.

For a random query, Figure 13 illustrates the cache hit rate for a data sequence with constant arrival interval and Poisson distribution ($\lambda = 0.02$), respectively. We noticed



(a) for constant interval (b) for Poisson distribution

Fig. 13. Hit rate for random-time query.

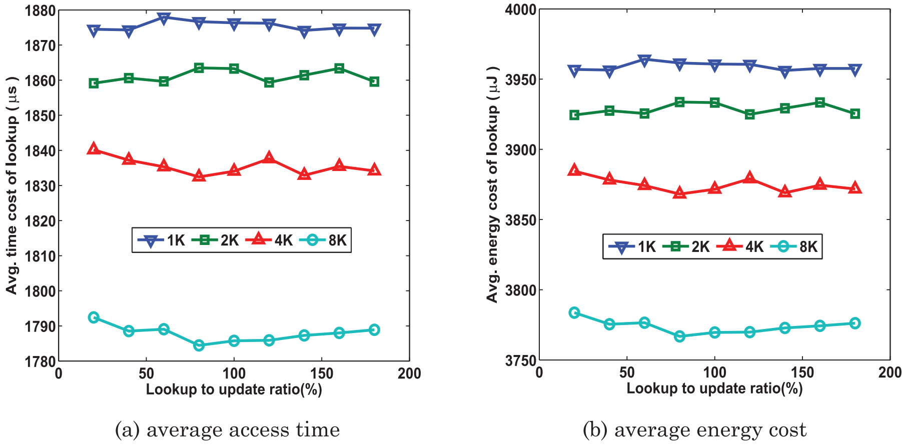


(a) for constant interval (b) for Poisson distribution

Fig. 14. Hit rate for temporal range query.

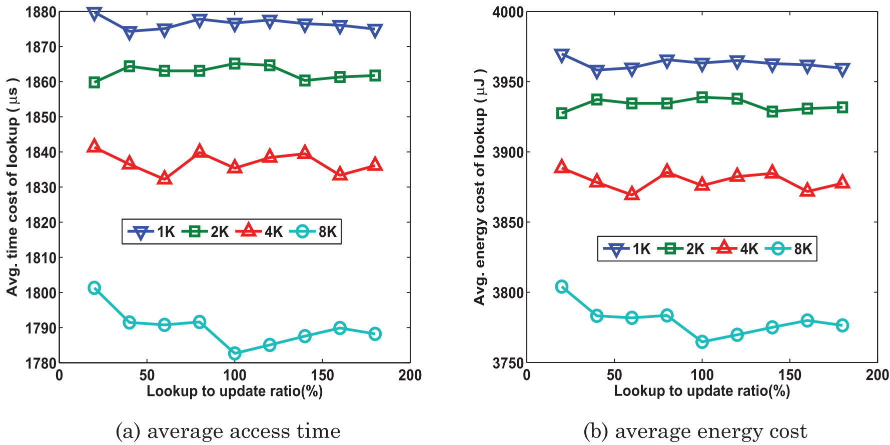
that the cache hit rate is low because in this benchmark the queried data are completely random and there is no connection between any two queries. Also, we found that cache miss traffic decreases when cache size increases.

The relationship between the read cache locality and the hit rate for the temporal range access is shown in Figure 14. We observed that the hit rate is very high for both sequences. In this benchmark each query looks for data records within a temporal range. For example, if we need to query data between [3000,5000] time range for a constant data interval model, we only need to find the page address that contains 3000, then read the sequential flash pages to the read cache until the maximum time in the page is bigger than 5000. We know one page contains 32 data (512 byte/page, data size = 16 byte), therefore, we need to read $(5000 - 3000)/32 \approx 63$ pages and only the first data in one page is needed to be read from flash, while the other 31 data values are already in the read cache. So the cache hit rate is about $(2000-63)/2000 \approx 96.85\%$. From Figure 14, it is not surprising to see that the cache hit rate is almost the same for different cache sizes, since: (i) the cache size is relatively too small, and thus



(a) average access time (b) average energy cost

Fig. 15. Random lookup on constant period.



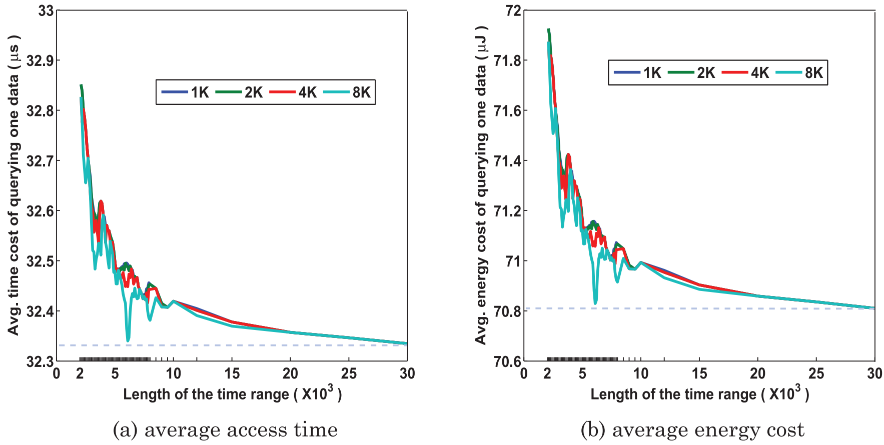
(a) average access time (b) average energy cost

Fig. 16. Random lookup on Poisson interval ($\lambda = 0.02$).

has little impact compared to other issues like data organization and query patterns; (ii) the temporal-ordered data organization and the temporal indexing structure of TL-Tree matches the time range access pattern, which eventually dominates the hit rate performance for range queries.

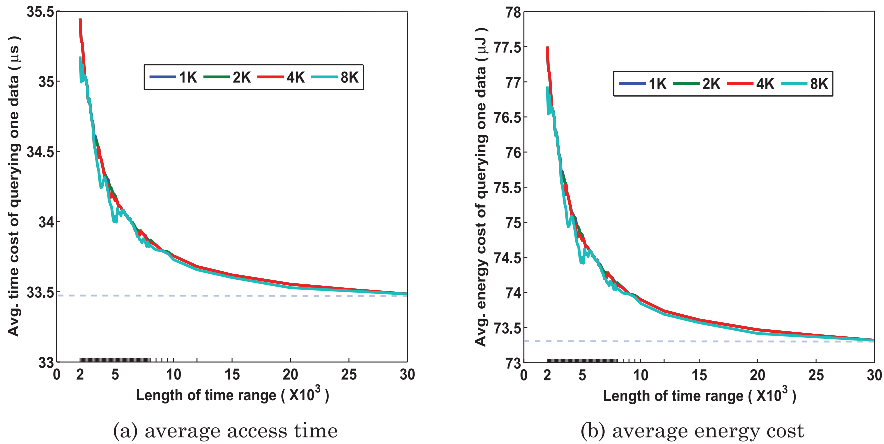
5.3.2. Random-Time Lookup. Figure 15 and Figure 16 present the average access and energy cost per lookup operation for different workloads with different cache sizes. Here, we only show the results a for Poisson distribution with $\lambda = 0.02$, since the results are very similar for $\lambda = 0.5$. We have the following observations.

First, for each benchmark workload, the impact of the read cache on the data query is almost the same whether the data arrival interval is constant or not. Second, for the same workload, TL-Tree benefits from the increased cache size. This is expected because queries are likely to find required data in the cache in direct proportion to the cache size. Third, for a given cache size, the cost is relatively stable. This is because the cache size is too small compared to the 128M NAND flash memory and also that the queries generated are completely independent.



(a) average access time (b) average energy cost

Fig. 17. Time-range lookup of TL-Tree (constant interval).



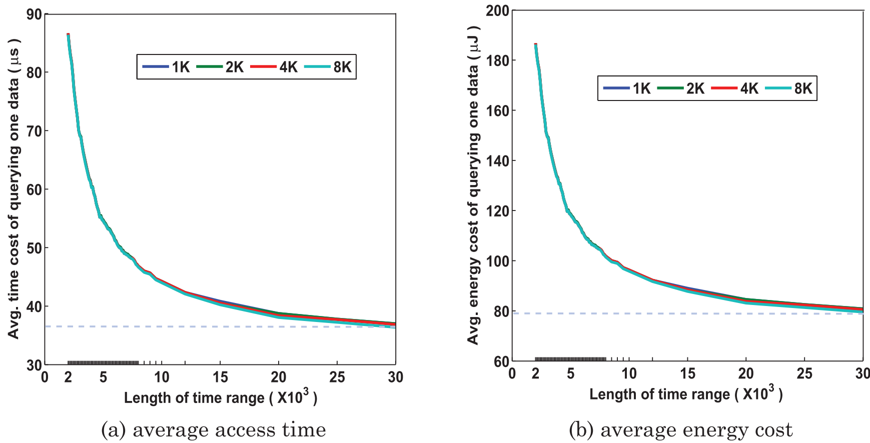
(a) average access time (b) average energy cost

Fig. 18. Time-range lookup of TL-Tree ($\lambda = 0.5$).

5.3.3. Time-Range Lookup. Figure 17, Figure 18, and Figure 19 illustrate the cache performance for the range queries. We first observed that, as the query range increases, TL-Tree performance improves dramatically. The benefit comes from the time-based index structure and the high cache hit for sensor data with high temporal locality. Also, for a specific data arrival pattern, different cache sizes have very limited impact on the performance. This is because only the first and the last data need to be allocated, and other in-between data can thus be loaded into the cache as needed. This holds for different cache sizes.

Another observation is the energy consumption and access cost for queries with Poisson arrival intervals at $\lambda = 0.5$ are vary close to those at the constant arrival pattern. Here, we noticed that the expected time interval is 2s for a Poisson distribution at $\lambda = 0.5$ and 1s for the constant period. The results further indicate that the performance of TL-Tree is stable and not sensitive to the data arrival distribution, because the number of data items obtained is the major impact factor for a query range.

Finally, when the expected time interval is relatively large, for example, when $\lambda =$ 0.02 for a Poisson distribution, it consumes more energy compared to small arrival



(a) average access time (b) average energy cost

Fig. 19. Time-range lookup of TL-Tree ($\lambda = 0.02$).

intervals if the query range is small. As the range increases, such as to 30000, the cost becomes low as in $\lambda = 0.5$. Since each page contains a fixed number of data items, for instance, 32 in this work, if the interval is constant, then for a given range query, say 2000, the total number of pages accessed is $2000/32 \approx 60$, where the energy consumption for each data lookup is about $60/2000 = 3/100$ (it assumes one energy unit for each page). If the expected interval is 50 ($\lambda = 0.02$) for a Poisson distribution, then only 2000/50 data items can be obtained, which occupy two pages. In this case, the average energy consumption for each item is 2/40, about twice 3/100.

5.4. Performance Comparison

In this section, we choose a flash-optimized index structure, namely, μ-Tree [Kang et al. 2007], to perform the comparison. μ-Tree attempts to improve upon the B^+-Tree by putting all nodes along the path from the root to the leaf together into a single flash memory page in order to minimize the number of flash write operations. Although this specific improvement of μ-Tree outperforms B^+-Tree for access latency, it is designed for generic flash devices, not the sensor platforms that store time-series sensor data. In order to have a fair comparison, we first enhance the μ-Tree to be more adaptive to the sensor network applications, and then compare the TL-Tree to it and the original μ-Tree.

5.4.1. $\mu(n)$-Tree. According to the balanced structure of μ-Tree, when the temporal-ordered index node is full, it will split it into two subindex nodes and each node is half full. If we use the generated time as index, all the new data will be inserted into the right subnode, while meanwhile, the left one remains half empty—a big waste of the index storing space. So, instead of splitting into subnodes when the index node becomes full, we propose to allocate a new empty node to keep the new arrival keys. In addition, each time when a new key is generated, μ-Tree has to allocate a new page to load the path from the root to this key because NAND flash does not support in-place update. This process will, however, not only lead to large useless space in old pages, but also lead to a lot of access operations on memory and NAND flash. In order to alleviate this impact, we propose to use a 256-bytes buffer for the newly generated index keys (μ-Tree is designed to allocate half of a flash page to the new keys in the leaf, so the key buffer should be designed to conform to the page size of the specific flash, here 512 bytes per page). The purpose of this key buffer is to save those new keys until

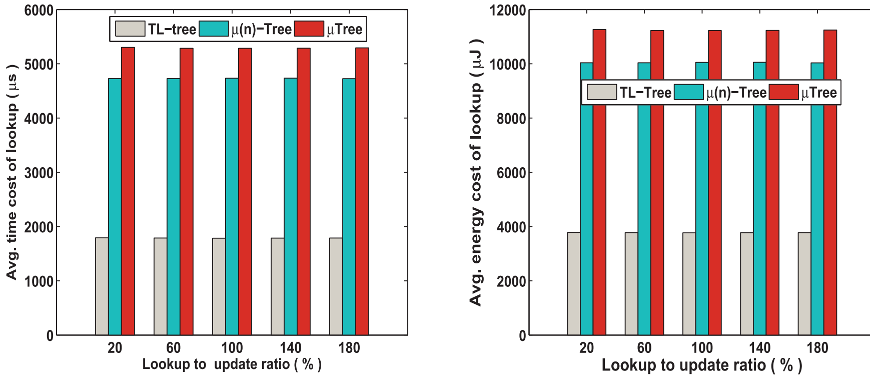


Fig. 20. Random lookup.

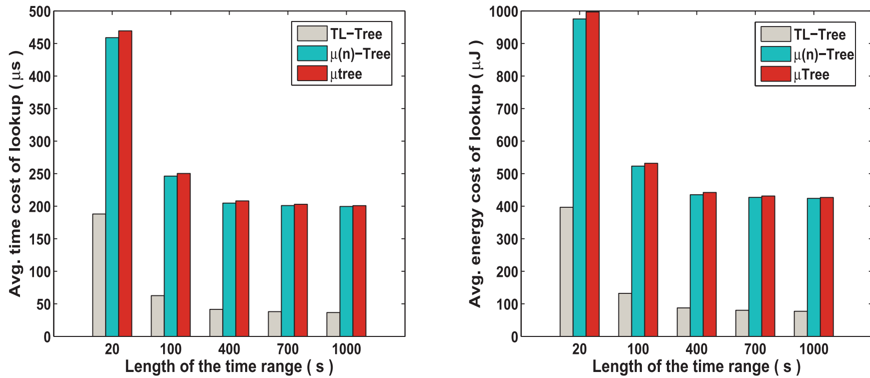


Fig. 21. Time-range lookup.

the cache is full, then all the keys will be written to the flash once. The benefit of this process is that it decreases the access times of flash memory.

In the following part, the aforesaid μ-Tree-tailored index structure is called $\mu(n)$-Tree (n implies the buffer size). The writing steps of $\mu(n)$-Tree include: (1) writing the data to the write cache; (2) when the write cache is full, generating an index node; (3) writing this index node to the key buffer; and (4) when the buffer is full, moving all the nodes from the buffer to the flash memory.

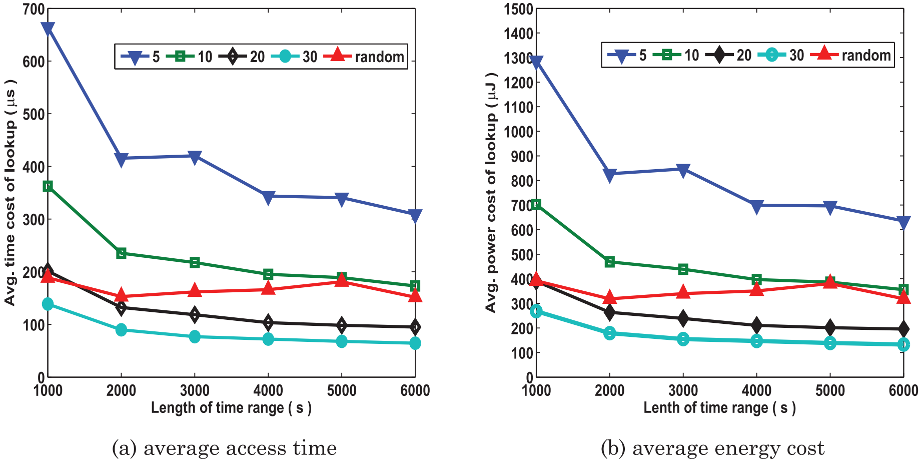
5.4.2. Comparison for Lookup *Operation.* In this section, we only present the results for the use of 8K read cache for the experiments. (The results with different cache sizes are similar and hence omitted.) The results are shown in Figure 20 and Figure 21.

We observe that, in general, TL-Tree has superior performance to $\mu(n)$-Tree and μ-Tree in terms of both access cost and energy cost. For example, under the 100% workload condition for the random lookup, the access time of TL-Tree takes only about 37% and 33% of $\mu(n)$-Tree and μ-Tree, respectively. And the energy cost benefits are similar. This is because both μ-Tree and $\mu(n)$-Tree are stored in flash memory. If the data is not found in cache, this kind of storage organization will result in many accesses to flash. Although $\mu(n)$-Tree is an improved tree, its logical structure is the same as that of μ-Tree.

For the case of time-range lookup operation also, the TL-Tree outperforms the other two trees. Because the data stored in the subtree area of TL-Tree is based on time

Table VIII. Insert Cost Comparison

Index Structure	Time(μs)	Energy(μJ)
TL-Tree	38.86	70.05
μ(n)-Tree	43.69	80.31
μ-Tree	300.55	617.48



(a) average access time (b) average energy cost

Fig. 22. Value-based lookup.

sequence, when looking up the data given a time range, we only need to judge whether this range is contained in a subtree area or not, then, we can read the sequential pages in the area.

5.4.3. Comparison for Insert *Operation.* Table VIII presents the *insert* cost of the three trees. We observe that both TL-Tree and $\mu(n)$-Tree have better performance than μ-Tree. What makes the write performance of $\mu(n)$-Tree superior to μ-Tree is the key buffer we have added in the main memory, since the need to move all index nodes from the buffer to the flash arises only when the key buffer is full. This operation is very similar to the process of the raising tree in TL-Tree, and thus they have close performance. However, due to the specific structure, TL-Tree has a bigger raising tree than the buffer assigned to $\mu(n)$-Tree, which leads to less accesses to the flash. Therefore, TL-Tree outperforms $\mu(n)$-Tree.

5.5. *Value-Based* Lookup Operation

We evaluate the value index structure introduced in Section 3.3 and the search operation presented in Section 4.2. The query searches for the temperature data between $x_1^o C$ and $x_2^o C$ within a time range of t_1 and t_2. The temperature values are randomly generated in the range of $[-20, 50]$, and the time range is set from 1000 to 6000 in increments of 1000. We did two different types of experiments: one is for a fixed-value range query where the range is set to $[5, 10, 20, 30]$, and the other is for a variable value range where they are randomly selected in $[0, 30]$.

The simulation results are shown in Figure 22. We observe the following: First, for each value range, as the time range increases, both the energy and time cost decrease. This is as in the previous discussion. Second, as the value range of the query increases, the performance becomes better. This is because if a query looks for data with a larger value range, it is more possible that the bulk of data are stored within a page or in continuous pages of a block. Finally, the costs for the random case are very low and

stable. These results further indicate the benefits of adapting TL-Tree to the sensor applications.

6. IMPLEMENTATION

Currently, the MicaZ platform lacks support for raw NAND flash storage [Agrawal et al. 2009]. Consequently we developed a driver for a raw NAND flash that comprises a 128MB Toshiba NAND chip (TC58DVG02A1FT00) for a storage system on the MicaZ platform. The driver provides read, write, and erase interfaces to control the NAND flash directly. Interested users can access the code at http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-1.x/contrib/beihang/. Above the driver, we have also implemented a prototype of a TL-Tree-based storage system on the MicaZ platform running TinyOS 1.x. Our system is written in nesC, and consists of a few hundred lines of code that implement the storage allocator and indexing module. More details can be found at http://nand.herokuapp.com/.

7. RELATED WORK

Energy and Memory Efficiency. Energy optimization is probably one of the most important objectives for the file system design of sensor platforms. Capsule [Mathur et al. 2006] achieves energy efficiency by optimizing the organization of storage objects to the type of access methods. Later, FlashDB [Nath and Kansal 2007] proposed a dynamic self-tuning database for NAND flash. It considers the specific characteristics of NAND flash and the workload, presenting a self-tuning B^{+}-Tree that can adapt to the dynamic behavior of the workload and the underlying device. FlashLogger [Nath 2009] is an energy-efficient sensor data logging system that uses lazy amnesic compression in a flash-efficient manner. Although these approaches achieve energy efficiency from different aspects, among them, only Capsule and FlashLogger claim to adopt a memory cognizant design. Capsule employs a hardware abstraction layer that hides the vagaries of flash memories from the application and uses a log-structured design along with write caching for memory efficiency. FlashLogger abstraction uses approximately 500 lines of nesC code and it has around 14KB ROM footprint and 1.5KB RAM footprint on a Moteiv Tmote Sky node. Unlike these efforts that tried to optimize the organization of the object or maintain compressed data on flash using a small memory footprint, our work takes the main memory constraint into the index design consideration. Through the optimization of the cascaded index structure, TL-Tree not only minimizes the in-memory usage, but also enhances the flash capacity.

Time-Series Concern. To the best of our knowledge, there have been few efforts with a specific design target for both time-series data management and temporal query support for NAND flash on sensor platforms. From the data organization perspective, according to the method of data arrangement, MicroHash [Zeinalipour-Yazti et al. 2005] is probably the closest storage structure to TL-Tree because it provides an append-only and time-based storage. However, it proposes a hash-based index structure and index is built on the value, not the time as in TL-Tree. This method needs to generate one index record for each data record. Also, for the time-range queries, it may have to search the whole flash memory—a big cost in both time and energy. ELF [Dai et al. 2004] is a log-structured flash file system especially for microsensor nodes. Its design considered the temporal sensing data and attempted to avoid excessive energy consumption. However, it is designed for the NOR flash only, so it cannot be directly used for the NAND flash memory due to the specific read/write characteristics.

Others. YAFFS [Aleph One 2001] seems to be the first log-structure file system designed especially for raw NAND flash. But it only supports the page size of 512-bytes NAND flash chips. Though YAFFS2 [Aleph One 2005] supports more NAND flash chips, the capacity is still very limited. CFFS [Lim and Park 2006] is a log-structured

file system that takes in a pseudo-hot-cold separation to reduce the garbage collection overheads. Although these log-structured systems have good performance for update and garbage collection, they are not designed for sensor platforms, so they do not optimize for memory and energy. In μ-Tree [Kang et al. 2007], the whole search read index is packed into one page, which can reduce the update pages when out-of-place updating happenes. But it needs at least 50% of the storage capacity to store the index information, which is a big waste for flash. LA-Tree [Agrawal et al. 2009] is a universal index system designed for SSD and other NAND flash. It uses cascaded update buffers that greatly reduce the update pages. However, due to the limited main memory and flash characteristics, where to put the buffer is a challenging problem.

8. CONCLUSIONS AND FUTURE WORK

In this article we propose the TL-Tree, a time-log, cascaded, fast load-in, unexpected power-off-tolerant and power-optimized tree index system for the raw NAND flash used for sensor platforms. TL-Tree consists of three components: root tree, subtrees, and raising tree, where each part can be viewed as an individual tree structure. The differences between TL-Tree and the standard B^+-Tree are many. First, according to the index construction of TL-Tree, TL-Tree is not a balanced tree, and the search path from the tree root node to a leaf can be of different lengths. Therefore, it violates the most important property, namely, the balance property of B^+-Tree. Also, there is no other physical storage needed for the non-leaf TL-Tree nodes because of the full subtree property. However, in traditional DBMS, the non-leaf B^+-Tree nodes are normally physically stored in another place such as main memory or cache rather than disk, to speed up the search since the nodes are not full and the deletion operation can lead to a different physical structure.

Compared with the state-of-the-art, TL-Tree is designed specifically for time-series sensor data stored in NAND flash on memory-constrained sensor platforms. With memory-cognizant structure design, TL-Tree cannot only sustain large flash storage capacity, but also achieve significant performance in terms of access and energy efficiency for both time- and value-based queries. Specifically, it offers the following benefits.

(1) *Read cache impact.* Although as the cache size increases, TL-Tree can obtain more benefit that enables it to perform similarly to the other indexing structures, the read cache has relatively little influence for the time-range lookup operation. The reason is that TL-Tree keeps the temporal sequence property in the indexing structure, so the performance is stable and not sensitive to different data arrival distributions. In addition, the cache hit rate does not vary much for different cache sizes. This further demonstrates the advantage of the temporal design of TL-Tree that dominates the performance.

(2) *Efficient lookup and insert operations.* Compared to flash-oriented B^+-Tree, TL-Tree has superior performance. For instance, under the 100% workload condition for the random lookup, the access time of TL-Tree takes only about 37% and 33% of $\mu(n)$-Tree and μ-Tree, respectively. It has similar benefits with respect to energy costs. For an insert operation, μ-Tree consumed 7 to 9 times the time and energy compared to TL-Tree.

(3) *Effective value-based retrieval.* The extended value-supporting TL-Tree has shown effective performance for the value-based queries, such as searches for the temperature data between $x_1^o C$ and $x_2^o C$ within a time range of t_1 and t_2. In addition to the superior results for varying time ranges, we also observed that, as the value range increases, both the time cost and energy cost decrease. This decrease is more significant for those queries with relatively smaller time windows.

Although TL-Tree is able to effectively support a large fraction of timestamp-based and value-based queries in sensor applications, we only considered sensor data with small size, such as temperature or humidity for wireless sensor networks. How to extend TL-Tree to index multimedia sensor data stored in flash is one of our important future works. Also, designing an embedded temporal database management system to support those queries that involve multiple-value types appears to be another promising and challenging topic to examine.

REFERENCES

A. A. Abbasi and M. Younis. 2007. A survey on clustering algorithms for wireless sensor networks. *Comput. Comm.* 30, 14–15, 2826–2841.

D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. 2009. Lazy-adaptive tree: An optimized index structure for flash devices. In *Proceedings of the 35th International Conference on Very Large Databases (VLDB'09)*. 361–372.

ALEPH ONE. 2001. Yaffs: Yet another flash file system. http://www.yaffs.net.

ALEPH ONE. 2005. Yaffs2 specification and development nodes. http://www.yaffs.net.

ATMEL. 2001. ATmel AT49f1024 datasheet. http://www.atmel.com.

M.-L. Chiao and D.-W. Chang. 2011. Rose: A novel flash translation layer for NAND flash memory based on hybrid address translation. *IEEE Trans. Comput.* 60, 6, 753–766.

H. Cho, D. Shin, and Y. I. Eom. 2009. Kast: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*.

Y.-S. Chu, J.-W. Hsieh, Y.-H. Chang, and T.-W. Kuo. 2009. A set-based mapping strategy for flash-memory reliability enhancement. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'09)*. 405–410.

H. Dai, M. Neufeld, and R. Han. 2004. Elf: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*. 176–187.

P. Desnoyers, D. Ganesan, and P. Shenoy. 2005. Tsar: A two tier storage architecture using interval skip graphs. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys'05)*. 39–50.

Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. 2007. Rethinking data management for storage centric sensor networks. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*. 410–419.

E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. 2007. In-network aggregation techniques for wireless sensor networks: A survey. *IEEE Wirel. Comm.* 14, 2, 784–789.

J. Gehrke and S. Madden. 2004. Query processing in sensor networks. *Pervas. Comput.* 3, 1, 46–55.

G. Graefe. 2009. The five-minute rule twenty years later, and how flash memory changes the rules. *Comm. ACM* 52, 7, 48–59.

A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. 2011. Leveraging value locality in optimizing nand flash-based SSDS. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*.

D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. 2007. μ-tree: An ordered index structure for NAND flash memory. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software (ACM/EMSOFT'07)*. 144–153.

S. J. Kwon, A. Ranjitkar, Y.-B. Ko, and T.-S. Chung. 2011. FTL algorithms for NAND-type flash memories. *Des. Autom. Embed. Syst.* 15, 3–4, 191–224.

S. Lee, D. Shin, Y.-J. Kim, and J. Kim. 2008. Last: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM Oper. Syst. Rev.* 42, 6, 36–42.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. 2005. Tinyos: An operating system for wireless sensor networks. In *Ambient Intelligence*, Springer, 115–148.

H. Li, Y. Liu, W. Chen, W. Jia, B. Li, and J. Xiong. 2013. Coca: Constructing optimal clustering architecture to maximize sensor network lifetime. *Comput. Comm.* 36, 3, 256–268.

S.-H. Lim and K.-H. Park. 2006. An efficient NAND flash file system for flash memory storage. *IEEE Trans. Comput.* 55, 7, 1–7.

S. Madden, J. Hellerstein, and W. Hong. 2005. Tinydb: In-network query processing in tinyos. http://telegraph.cs.berkeley.edu/tinydb/tinydb.pdf.

G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy. 2009. Ultra-low power data storage for sensor networks. *ACM Trans. Sensor Netw.* 5, 4, 33:1–33:34.

G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. 2006. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys'06)*. 195–208.

S. Nath. 2009. Energy efficient sensor data logging with amnesic flash storage. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'09)*. 157–168.

S. Nath and A. Kansal. 2007. Flashdb: Dynamic self-tuning database for NAND flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN'07)*. 410–419.

D. Park, B. Debnath, and D. Du. 2010. CFTL: A convertible flash translation layer adaptive to data access patterns. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10)*. ACM Press, New York, 365–366.

R. Ramakrishnan and J. Gehrke. 2003. Tree indexes. In *Database Management Systems,* McGraw-Hill Higher Education, 360–363.

SAMSUNG. 2014. M390S2858CT1 datasheet. http://www.samsung.com/global/system.

W. Song, R. Huang, M. Xu, A. Ma, B. Shirazi, and R. Lahusen. 2009. Air-dropped sensor network for real-time high-fidelity volcano monitoring. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services (MobiSys'09)*. 305–318.

R. Tan, G. Xing, J. Chen, W. Song, and R. Huang. 2009. Quality-driven volcanic earthquake detection using wireless sensor networks. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'09)*. 271–280.

TOSHIBA INC. 2014. Nand vs. nor flash memory technology overviews. http://umcs.maine.edu/~cmeadow/courses/cos335/Toshiba%20NAND_vs_NOR_Flash_Memory_Technology_Overviewt.pdf.

TOSHIBA INC. 2003. Toshiba tc58dvg02a1ft00 datasheet. http://www.toshiba.com/taec-Datasheet:TC58DVG02A1FT00.

N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. 2005. Multi-query optimization for sensor networks. In *Proceedings of the 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*. 307–321.

D. Woodhouse. 2001. Jffs: The journalling flash file system. In *Proceeding of the Ottawa Linux Symposium*.

C.-H. Wu, L.-P. Chang, and T.-W. Kuo. 2007. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.* 6, 3.

C.-H. Wu, H.-H. Lin, and T.-W. Kuo. 2010. An adaptive flash translation layer for high-performance storage systems. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 29, 6, 953–965.

Y. Yao and J. Gehrke. 2002. The cougar approach to in-network query processing in sensor network. *IEEE Trans. Comput.* 31, 3, 9–18.

D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. 2005. Microhash: An effcient index structure for flash-based sensor devices. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*. 31–44.

Received July 2012; revised August 2013; accepted August 2013