Microsearch: A Search Engine for Embedded Devices Used in Pervasive Computing

CHIU C. TAN, BO SHENG, HAODONG WANG, and QUN LI
College of William and Mary

In this paper, we present Microsearch, a search system suitable for embedded devices used in ubiquitous computing environments. Akin to a desktop search engine, Microsearch indexes the information inside a small device, and accurately resolves a user's queries. Given the limited hardware, conventional search engine design and algorithms cannot be used. We adopt information retrieval (IR) techniques for query resolution, and proposed a new space efficient top-k query resolution algorithm. A theoretical model of Microsearch is given to better understand the tradeoffs in design parameters. Evaluation is done via actual implementation on off-the-shelf hardware.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.3 [**Special Purpose and Application-based Systems**]: Real-Time and Embedded Systems

General Terms: Design, Security, Algorithms

Additional Key Words and Phrases: Embedded search engine, information retrieval, pervasive computing

1. INTRODUCTION

Pervasive computing allows users to interact with their physical environment just as they would a laptop. A tourist can just as easily interact directly with a signpost for directions as he would a navigate a website. Attendants in a conference can obtain minutes of the previous meeting by querying the conference desk instead of obtaining the data from the group wiki. Such applications all rely on small devices embedded into everyday objects and environment.

In this paper, we describe Microsearch, a search system designed for small embedded devices. We use the following example to illustrate how Microsearch can be used. Consider a collection of document binders. Each binder is embedded with a small device running Microsearch. Each device contains some information about the documents found in that binder. When a user wishes to find some documents, he can query a binder using some terms, i.e. "acme,coyote,refund", and Microsearch will return a ranked list of documents that might satisfy his query. Also included

Authors' address: Chiu C. Tan, Bo Sheng, Haodong Wang, and Qun Li, College of William and Mary, Department of Computer Science, Williamsburg, VA 23185; email:{cct,shengbo,wanghd,liqun}@cs.wm.edu.

This project was supported in part by US National Science Foundation grants CNS-0721443, CNS-0831904, and CAREER Award CNS-0747108.

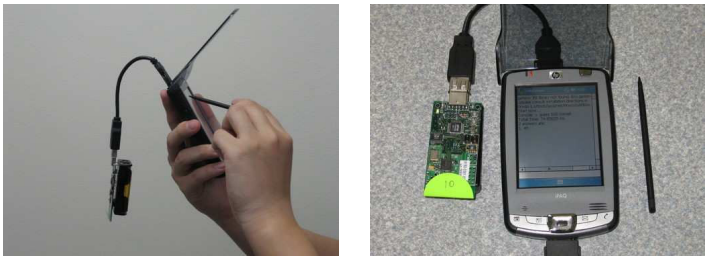


Fig. 1. Example of a PDA querying Microsearch

in the reply is a short abstract of each document to help him make his decision. Later, the user decides to add some notes to a document. Through input devices such as a digital pen [Logitec 2004] or PDA, the user can store notes into each binder. Figure 1 shows a PDA modified with a sensor mote that can be used on Microsearch. Microsearch will then index the user input for future retrieval.

Microsearch is designed to run on resource constrained small devices capable of being embedded into everyday objects. An example of a small device is manufactured by Intel [Nachman et al. 2005] which has a 12MHz CPU, 64KB of RAM, 512KB of flash memory, and wireless capabilities, all packaged in a 3x3 cm circuit board. Larger storage capacity can also be engineered to store more data. In this paper, we use the terms "mote" and "small device" interchangeably.

Similar to desktop search engines like Google Desktop [Google 2007], Spotlight [Apple 2007] or Beagle [Beagle 2007], Microsearch indexes information stored within a mote, and returns a ranked list of possible answers in response to a user's query. We envision that Microsearch can be an important component in physical world search engines like Snoogle [Wang et al. 2008] or MAX [Yap et al. 2005].

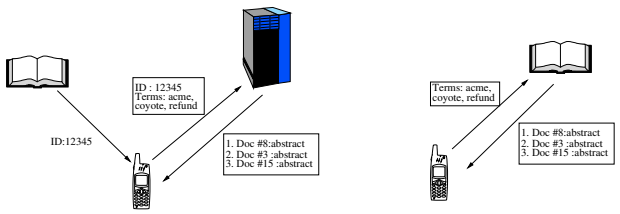
1.1 Background

Earlier pervasive systems [Abowd et al. 1997; Cheverst et al. 2000; Cheverst et al. 2000; Rekimoto et al. 1998; Starner et al. 1997] typically embed simple RF devices like RFID tags onto physical objects. Each tag contains a unique ID identifying that particular object. Data about physical objects are stored remotely on large servers, and are indexed by their respective IDs. To send or receive information regarding a physical object, a user first obtains the ID from the object, and then contacts the remote server with the ID. This design paradigm embeds very simple devices into physical objects, and relies on powerful servers for computation.

As embedded devices become more powerful, a different design paradigm which does not utilize a backend server can be used. Instead of embedding a simple RF beacon into an object, a more powerful device is embedded. Information previously kept on a server will now be stored directly on this device. User queries will also be resolved by the object itself. Figure 2 illustrates the two approaches.

There are several advantages in using a more powerful device to store and retrieve data, rather than relying on a server.

— Data accessibility. Storing the data on a more powerful embedded device instead of a remote server allows the user to obtain data directly from the object



(a) Data on server (b) Data on object

Fig. 2. (a) Typical design utilizing backend server. (b) Different paradigm without use of a server

using protocols like Bluetooth. In areas without long range wireless networks, a simple RF device will only give the user an ID and no other useful information.

— Simple deployment. An object embedded with a powerful device can be used without additional configuration, whereas a simple RF device still needs to be configured with a remote server before it can be deployed.

— Intuitive ownership transfer. Storing data directly on the device allows the use of more intuitive security protections for ownership transfer. When user A hands over the physical object to user B, user A no longer can access the stored data since he no longer has access to the object itself. When using a simple RF device, when user A hands over the physical object to user B, the associated data still resides on user A's servers, allowing user A access even though he no longer posses the physical object.

Despite the advantages, there are two main drawbacks for using a more powerful device. The first is **cost**. Simple RF devices like RFID tags are inexpensive, each tag costing several cents, compared to tens of dollars for a more powerful device. Since RF devices are so much cheaper, several of such devices can be attached to the same object to improve reliability against damaged tag. The relative higher cost rules out deploying multiple more powerful devices on the same object. The second drawback is **maintenance**. The more powerful device will require periodic maintenance such as replacing the batteries, whereas a simple RFID tag once attached are essentially maintenance free.

Given the strengths and weaknesses, no approach is suitable for all applications. Pervasive applications involving multiple owners and objects operating mostly in an outdoor environment are likely to benefit from using a more powerful device over a weaker RF device. The simple ownership transfer property makes it easier to manage data when the object has to move between different owners. Also, the lack of wireless connectivity makes communication with remote servers unreliable. Examples of such applications include tracking packing crates which may want to record contents and notes as the crate moves from one location to another. By storing the data directly on the crate, the data is available only to the new owners holding the crate.

1.2 Our Contributions

The challenge of designing Microsearch lies in engineering a complete solution that can run efficiently on a resource constrained platform. Desktop search systems

typically require large amounts of RAM to perform indexing. Similarly, query resolution algorithms usually store intermediate results in memory while resolving a query. With just kilobytes of RAM to spare, it is impossible to port existing solutions directly onto motes. In addition, mote hardware uses flash memory for persistent storage. While conventional flash file systems [Company 2008; Woodhouse 2001] have been designed, they require more memory than is available on a mote. This necessitates a different system design.

We make the following contributions in this paper. (1) We provide a system architecture that effectively utilizes limited memory resources to store and index different inputs. (2) Our architecture incorporates information retrieval (IR) techniques to determine relevant answers to user queries. (3) Since conventional IR techniques are designed for more powerful systems in mind, we introduce a space saving algorithm to perform IR calculations with limited amounts of memory. Our algorithm can return the top-k relevant answers in response to a user query. (4) A theoretical model of Microsearch is presented to better understand how to choose different system parameters. (5) Finally, we implement Microsearch on an actual hardware platform for evaluation.

The rest of this paper is as follows: Section 2 contains related work, and Section 3 describes the Microsearch system design. Section 4 details the security protections, Section 5 covers our search algorithms, and Section 6 presents the theoretical model of Microsearch. Section 7 contains our evaluation. and Section 8 concludes.

2. RELATED WORK

Desktop search engines are a mainstream feature found in most modem operating systems. In general, these search engines collect metadata from every file, and store the metadata into an inverted index, a typical data structure used to support keyword search [Faloutsos 1985]. Information retrieval algorithms are then used to determine the best answer to a query. Our work draws from the basic principals of IR to rank query results.

A counterpart to Microsearch is PicoDBMS [Pucheral et al. 2001], a scaled down database for a smart card. PicoDBMS allows data stored inside the smart card to be queried using SQL-like semantics. The main design difference between our work and PicoDBMS is that PicoDBMS uses a database design. Their approach works well in a specific domain like storing health care information, which can enforce structured inputs with specified attribute terms, and assume well trained personal. Microsearch on the other hand uses a search engine design which allows for unstructured inputs without enforcing pre-specified attributes, and a natural language query interface. The relationship between the two can thus be summed up as the differences between a search engine and a database.

We proposed an embedded search system in [Wang et al. 2008] which allows one to search the physical environment, but focused on integrating a hierarchy of sensors that can cover a large area rather than on how an individual embedded device manages data. Our later work [Tan et al. 2008] considered the problem of building an information management system on a single sensor. However, [Tan et al. 2008] does not provide any security solution to protect the data. Furthermore, in this paper, we improved on the theoretical model found in [Tan et al. 2008], and

evaluated its accuracy.

Low level flash storage systems on the sensor platform have only recently gained interest among researchers. Earlier sensor storage research treated the low level storage as a simple circular log structure. Efficient Log-Structured File System (ELF) [Dai et al. 2004] was the first paper that introduced a file system especially tailored for sensors, providing common file system primitives like append, delete and rename. Another file system is Transactional Flash File System (TFFS) [Gal and Toledo 2005b] which deals with NOR flash. Both research are different from ours in that they provide a sensor *file system* and not a sensor *search system*. A search system emphasizes good indexing and query response time while a file system does not.

Closer to our work is MicroHash [Zeinalipour-Yazti et al. 2005] which focuses on efficient indexing of numeric data using the sensor flash storage. It creates an index for every type of data monitored by the sensor, for example, temperature or humidity. Since the data indexed by MicroHash is generated by the sensor itself, the index size can be predetermined from the sensor hardware specifications. For example, if the sensor hardware supports temperature monitoring between 10 and 50 degrees at 1 degree granularity, the index with 40 entries can hold all possible data generated. An adaptive algorithm is applied to repartition the index to improve performance. Our research differs from MicroHash in two main ways. First, we allow indexing of arbitrary type of terms, not just numeric ones, and second, we adopt information retrieval algorithms to reply to queries.

Systems like Journaling Flash File System [Woodhouse 2001], Yet Another Flash File System [Company 2008] are designed primarily for larger devices, making them unsuitable for the sensor platform. We refer to [Gal and Toledo 2005a] for more details. One interesting exception is Capsule [Mathur et al. 2006a], which provides object primitives like a stack or index for other sensor applications. These object primitives are designed to work on sensor platform. Unfortunately Capsule's index primitives require the indexable data set to be known before hand, making it unsuitable for indexing the generic metadata. There is also no retrieval algorithm for ranking query results.

File system search is a mainstream feature in most modem operating systems. Since most desktop search systems are commercial offerings, detailed system design is unavailable. However, most search systems share some common functionalities. Metadata for every file is collected and stored in an index. This index data structure in its simplest form resembles an inverted table [Frakes and Baeza-Yates 1992], where given a term, it returns the location of the file containing that term. Information retrieval algorithms [Kobayashi and Takeda 2000; Faloutsos and Oard 1995; Frakes and Baeza-Yates 1992; French et al. 1999] are used to determine the best answer to a query.

3. SYSTEM ARCHITECTURE

We begin with describing the inputs to Microsearch. We assume that a user uploads information to Microsearch via a wireless connection through a suitable interface like a PDA. Microsearch requires every user input to consist of two segments, a *payload*, and a *metadata*. The payload is the actual information the user wishes

other people to download. The metadata is a description of the payload data, and is used to determine whether a payload is relevant to a user's query. Both the payload and metadata are user generated.

The metadata is essentially a list of terms describing the corresponding payload. Microsearch requires each term, known as a *metadata term*, to be accompanied by a numeric value, known as a *metadata value*, indicating how important *that* term is in describing the payload. A metadata using n metadata terms to describe a payload can be represented as $\{(term_1, value_1), \cdots, (term_n, value_n)\}$. For a text based payload, the simplest method to determine the metadata value for a term is to count the number of times that term appears in the payload. Metadata values for non-text based payloads can be defined by the user.

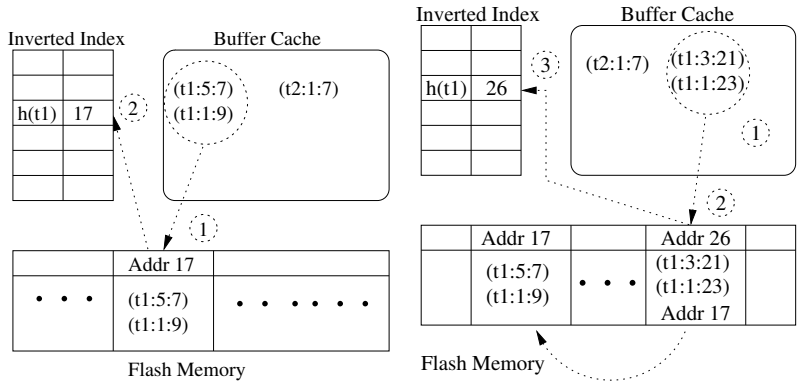
3.1 Microsearch Design

Microsearch maintains two data structures in RAM: a buffer cache, and an inverted index. The buffer cache is used to temporarily store and organize data before writing to flash to improve overall performance. The inverted index is used to track and retrieve the stored data. In general, when receiving an input file, Microsearch stores the payload into flash memory, and the metadata into the buffer cache. This continues as more inputs are sent to Microsearch until the buffer cache is full. Selected metadata entries are then organized and flushed to flash memory to free up space in the buffer cache, and the inverted index is updated.

Receiving an input: Upon receiving an input file, Microsearch first stores the metadata into RAM, and then writes the payload directly to flash memory. The starting address of the payload in flash is returned and added to each metadata entry for that payload. With this payload address, Microsearch can recover the entire payload if needed. Each metadata entry in the buffer space now becomes a tuple, $(term, value, address)$, consisting of a metadata term , a metadata value, and a payload address. For example, consider Microsearch writing a payload to flash memory location $addr_3$. All metadata associated with this payload becomes, $\{(term_1, 3, addr_3), \cdots, (term_n, 2, addr_3)\}$.

As mentioned earlier, flash memory is used as permanent storage for user inputs. Microsearch writes data to flash memory using a log structure style write which treats the entire flash memory as a circular log, always appending new data to the head of the log. A pointer indicating the next available location in flash memory is kept by Microsearch. Log-style writes have been found to be suitable for flash memory [Gal and Toledo 2005a]. Since writes are performed on a page granularity, Microsearch will always attempt to buffer the data into at least a single page before writing to flash.

Buffer cache organization: As more payloads are sent to the buffer cache, the buffer cache becomes a collection of metadata entries which describe the different input files stored in the mote. There is no longer the concept of a set of entries belonging to a particular payload. Instead, metadata entries which have the same metadata term are grouped together. For instance, two different input files may share some common metadata terms. Inside the buffer cache, the tuples with the same metadata terms are grouped together. For instance, two payloads stored in address $addr_3$ and $addr_8$ may share the same term $term_1$. Thus, inside the buffer cache, they will be grouped as $\{(term_1, 2, addr_3), (term_1, 5, addr_8)\}$.



(a) With uninitialized index slot (b) With initialized index slot

Fig. 3. (a) Buffer eviction with uninitialized index slot: 1) Flushes tuples from buffer cache, 2) Copies address of metadata page, $addr_{17}$, into inverted index. (b) Buffer eviction with initialized index slot: 1) Copies previous metadata page address from inverted index. 2) Flushes tuples from buffer cache. 3) Copies new address, $addr_{26}$, into inverted index.

Inverted index: An inverted index is commonly used in search engine systems to retrieve the archived information. A conventional inverted index has every slot correspond to a different term. Each slot stores a pointer to a list of documents or web pages containing that term. By matching a given query term with the inverted index, one can retrieve all the documents or web pages containing that term.

Microsearch uses a modified inverted index which differs from a conventional design in two ways. First, Microsearch uses a hash function to map multiple metadata terms to a certain slot in the inverted index. This results in a smaller inverted index which uses less RAM but is slightly inaccurate. We discuss how Microsearch resolves this inaccuracy in the next section. Second, Microsearch has each slot in the inverted index store the flash address of a page in flash memory containing a group of metadata terms which hash to the same slot. This flash page is known as a *metadata page*. An inverted index slot which already has metadata terms hashed to it is considered *initialized*.

Buffer eviction with uninitialized index slot: When the buffer cache reaches full capacity, tuples will have to be evicted to free up space for new entries. Microsearch hashes the tuples and selects the largest group with the same hashed result, and looks up the corresponding slot on the inverted index. If no metadata term has been hashed to that slot before, that slot is considered uninitialized. Microsearch organizes the group of tuples in the order of their arrival into the buffer cache, and writes the metadata pages into flash memory. If the group of tuples spans multiple flash pages, each metadata page contains the flash memory address of the next page. The address of the *last* metadata page containing the tuples is returned to the inverted index. The inverted index stores this address into the uninitialized slot. The slot is now considered initialized. Figure 3(a) illustrates this process.

Buffer eviction with initialized index slot: In the event that an inverted index slot has already been initialized, Microsearch first reads the page indicated by the address found in the index. Microsearch tries to add all the new tuples to that page. If there is not enough space, a new metapage is created and the address from the index will be copied onto the *first* metadata page of tuples. The group of tuples are written to flash memory as before, and the address of the last metadata page is returned and stored in the inverted index. The inverted index thus will always have the address of the latest metadata page written into flash memory. Since each metadata page in flash memory contains the address location of the preceeding page, every metadata page can be retrieved by traversing the links. We consider this *a chain* of metadata pages. Figure 3(b) illustrates this process.

Data deletion: Deletion in flash memory occurs at a sector granularity. A sector consists of many pages. Each page is 256B and a section is typically 64KB. A delete pointer is kept by Microsearch to indicate which is the next sector to erase. Once the flash is reaching full capacity, Microsearch frees up storage space by deleting the sector indicated by the pointer. Both payload pages and metadata pages in that sector are deleted.

Deleting a sector may cause a metadata page to point to a payload page that has already been deleted. We can use the delete pointer to determine what address have already been deleted. Microsearch ignores these payloads when returning data to the user. Deleting a sector may also cause a metadata page to point to another metadata page which no longer exists. Microsearch uses the delete pointer to determine if a metadata page has been deleted, and considers the chain of metadata pages to have terminated.

4. SECURE MICROSEARCH

Security is not a part of conventional search engines, but is an important consideration for embedded search engines. This is because while the servers used to run a conventional search engine can be kept in a secure location, small devices running Microsearch are deployed on physical objects that are easily misplaced or stolen.

4.1 Threats

As mentioned earlier, one advantage of storing data on the object instead of a server is simple ownership transfer. Once an object is handed off to a new owner, the previous owner automatically loses access to the stored data since he no longer has physical possession. However, this does not make Microsearch secure against an adversary that is in the vicinity of the user's object.

(1) **Privacy attack:** The adversary can query the user's object to obtain some private information.
(2) **Storage DoS attack:** The adversary can repeatedly send fake information to the user's object to deplete the storage space.
(3) **Query spoofing attack:** The adversary can use a malicious device to fool the user into querying the adversary's device instead of his own, and thus returning incorrect information to the user.
(4) **Storage spoofing attack:** The adversary tries to induce the user to store data onto the adversary's malicious device instead of his own. This effectively

"deletes" the user's object since no data is ever stored.

4.2 Straw Man Protocols

Given the high cost of updating embedded device software once deployed, additional consideration in the design phase will be useful. Here we consider several protocols which appear to provide adequate security but in actuality contain vulnerabilities.

Blanket encryption: One apparent solution is for the user to first create a secret key, and encrypt *all* the data before sending it to his object running Microsearch. Since Microsearch's indexing does not distinguish between ciphertext and plain text, no additional modifications are necessary. Since the adversary does not know the user's secret key, querying Microsearch only yields ciphertext which do not reveal the user's private information. This defends against the privacy attack. Blanket encryption also defends against the query spoofing attack. When the user receives any reply, he will decrypt using his secret key. Without knowing the secret key, the adversary cannot generate the correct ciphertext to respond to the user's query. The adversary will be detected when the user cannot decrypt a response.

However, blanket encryption does not defend against a storage DoS attack. The adversary can still store a lot of fake information for Microsearch to index, and thus deplete the storage space. The user is also vulnerable to the storage spoofing attack since he does not know whether his data has been stored on his own device.

Simple user only authentication: Another solution is for the user to first generate a public and private key pair, PK and SK. He then stores PK into his object running Microsearch. When the user wishes to store data or query Microsearch, he executes the following protocol,

$$\begin{aligned}
\text{User} \rightarrow \text{Microsearch} &: \text{Request} && (1)\\
\text{User} \leftarrow \text{Microsearch} &: PK\{n\} && (2)\\
\text{User} &: SK\{PK\{n\}\} = n' && (3)\\
\text{User} \rightarrow \text{Microsearch} &: n' && (4)\\
\text{Microsearch} &: \text{If n' = n, continue,} && (5)\\
&\quad \text{else terminate session}
\end{aligned}$$

where n is a random number generated by Microsearch, $PK\{n\}$ is encrypting n with the public key PK, and $SK\{PK\{n\}\}$ is applying the secret key SK to a bundle encrypted with PK. Note that Microsearch will generate a new n for each new request.

We see in step (2) that Microsearch encrypts a random number n with the public key. The value of n is obtained by applying the corresponding SK which is only known to the user. Thus, only the user can return the correct value of n' to Microsearch in step (4) which will match the original n. Without knowing the correct n, Microsearch will no longer process the user's request.

This protocol defends against the privacy attack and the storage DoS attack. The adversary does not know the secret key SK, and thus will not be able to return the correct n in step (4), leading Microsearch to terminate the session. This way, the adversary cannot query data nor add fake data to the object.

However, the simple user only authentication protocol is still vulnerable to attacks 3 and 4. The adversary will follow the same steps, but in step (5) will not check if $n' = n$. Instead, the adversary will always continue to process the user's request. Since the user knows the SK and expects to be authenticated, he will continue accessing the adversary's device as it were his own.

Hybrid solution: An apparent alternative is to combine the two straw man solutions together by running the simple user only authentication protocol *and* encrypting everything. However, this does not defend against the storage spoofing attack since the adversary's device can accept the user's encrypted data.

Simple mutual authentication: The problem with the simple user only authentication is that the user never verifies if the object processing his request belongs to him. A straightforward approach seems to be for the user to authenticate the device as well. The user first creates two public and private key pairs, one for himself, PK_u and SK_u, and the other for his object running Microsearch, PK_o and SK_o. The user stores PK_u and SK_o in the object. He then interacts with his object as follows

$$\begin{aligned}
\text{User} \rightarrow \text{Microsearch} &: \text{Request}, PK_o\{n\} && (1)\\
\text{Microsearch} &: SK_o\{PK_o\{n\}\} = n' && (2)\\
\text{User} \leftarrow \text{Microsearch} &: PK_u\{n'\} && (3)\\
\text{User} &: SK_u\{PK_u\{n'\}\} = n'' && (4)\\
\text{User} &: \text{If } n'' = n && (5)\\
\text{User} \rightarrow \text{Microsearch} &: n'' && (6)\\
\text{User} &: \text{Else terminate session.} && (7)\\
\text{Microsearch} &: \text{If } n'' = n', \text{ process request} && (8)\\
&\quad \text{else terminate session.}
\end{aligned}$$

This protocol appears to allow both the user and his object to authenticate each other. If the object belongs to the adversary, it will not know SK_o, and cannot obtain n' which is equal to n. As a result, when the user decrypts the adversary's reply in step (4), the user will observe $n'' \neq n$, and conclude that the object does not belong to him. Now considering the adversary trying to query the user's object. Since the adversary does not know SK_u, the adversary cannot send the correct n'' such that $n'' = n'$, resulting in the user's object declining to process the request. This protocol improves on the simple user only authentication, and appears to defend against all attacks listed above.

However, an adversary can launch an effective privacy and storage DoS attack by ignoring the objects reply in step (5). Since the adversary selects the initial random number n, he can always return the same value in step (6). The object will then verify that $n'' = n'$, and process the request.

4.3 Single User Protocol

The problem with the simple mutual authentication is the repeated use of the same random number. To be secure, two random numbers must be used, one generated by the user, and the other by the object. Assuming that we have only one user, we use the same setup as the simple mutual authentication, but executes the following

protocol.

$$\begin{aligned}
\text{User} \rightarrow \text{Microsearch} &: \text{Request}, PK_o\{n_1\} && (1)\\
\text{Microsearch} &: SK_o\{PK_o\{n_1\}\} = n_1' && (2)\\
\text{User} \leftarrow \text{Microsearch} &: n_1', PK_u\{n_2\} && (3)\\
\text{User} &: \text{If } n_1' = n_1 && (4)\\
\text{User} &: SK_u\{PK_u\{n_2\}\} = n_2' && (5)\\
\text{User} \rightarrow \text{Microsearch} &: n_2' && (6)\\
\text{User} &: \text{Else terminate session} && (7)\\
&\text{If } n_2' = n_2\text{, continues} && (8)\\
&\text{else terminate session}
\end{aligned}$$

where n_1 is a random number chosen by the user, and n_2 is the random number chosen by the object.

When the user sends $PK_o\{n_1\}$ to the object, only his object knows SK_o to decrypt and return the correct n_1. Thus at step (4) if $n_1' = n_1$, the user knows that the object belongs to him. Similarly, the object authenticates the user by sending $PK_u\{n_2\}$ which can only be decrypted by SK_u which is only known to the user. Thus in step (8), only the user can return the correct $n_2' = n_2$, at which point the object can trust that the request was not issued by an adversary.

We see that this protocol defends against both the privacy attack and storage DoS attack since the adversary cannot return the correct n_2', causing the object to terminate the session. Both query spoofing attack and storage spoofing attack are also foiled since the adversary's device cannot return the correct n_1' to the user, causing the user to terminate the session.

4.4 Multiple users

We can extend the single user protocol above to accommodate k users by storing $PK_1, \cdots, PK_k$ in the object. The user that sends a request will indicate which public key to use, and the rest of the protocol can be executed. However, the problem is that storing a large number of keys will take up limited storage space in the embedded device.

We assume that the owner of the object creates a master public and private key pair, PK_{mas} and SK_{mas}, as well as a key pair for the object, PK_o and SK_o. He stores SK_o and PK_{mas} into the object. Each valid user, u, will be issued his own public and private keys, PK_u and SK_u, and a certificate *cert* where

$$cert = SK_{mas}\{PK_u\}$$

Now, when the user wants to access the object, he will use the single user protocol, but include his *cert* and public key PK_u in the request. The object can apply the master public key to verity that PK_u is valid,

$$PK_{mas}\{cert\} = PK_{mas}\{SK_{mas}\{PK_u\}\} = PK_u.$$

Since only the object owner knows SK_{mas}, verifying PK_u through $PK_{mas}\{cert\}$ indicates that this public key is authorized by the owner. The rest of the interaction

Table I. RSA and ECC encryption parameters

Parameter	RSA	ECC
Secret key size	128 B	20 B
Public key size	128 B	40 B
Certificate size	128 B	40 B
Ciphertext overhead	128 B	60 B
Encryption time	21 s	2.8 s
Decryption time	21 s	1.4 s

remains the same as the single user protocol.

4.5 Performance Details

The challenge of implementing security on an embedded device is hardware limitations. With limited computation and battery power, conventional security protocols cannot be directly applied. The protocols given earlier rely on public key cryptography, which can be implemented using different cryptographic primitives such as RSA, ElGamah, or Elliptic Curve Cryptography (ECC).

A security protocol for Microsearch will inevitably be application and deployment specific. Since different types of users will have different requirements, we show in Table I several important parameters such as the size of a public key, and the time needed to encrypt some data [Wang et al. 2007; Wang et al. 2006]. The values are derived from an embedded device with 8MHz processor and 10KB RAM. The table can serve as a guideline for designers in estimating the cost of their protocols. A more detailed comparison of the costs is found in [Wang et al. 2008].

The ciphertext overhead in Table I is the additional number of bytes that result from encrypting the data. In other words, encrypting a 16 byte data using RSA will result in a 144 byte ciphertext. Furthermore, in practical security implementations, we typically do not use public keys to encrypt data. Instead, we use a symmetric key to encrypt the data and then use the public key to encrypt the symmetric key. A typically symmetric key is 16 bytes in size.

5. QUERY RESOLUTION

Query resolution describes the process of returning an accurate answer to a user's query. A user queries a mote by sending a list of search terms and parameter k which specifies the top-k rankings he is interested in. The user receives an ordered list of k possible payload data as an answer. We begin by first introducing a basic query resolution algorithm. The actual space saving algorithm used by Microsearch is presented later.

5.1 Information Retrieval Basics

Microsearch uses a simple information retrieval weighing calculation, the TF/IDF function, to determine how relevant each payload address is in satisfying the user's query. Under the TF/IDF function, the weight of each metadata term of a payload is determined by the product of $TF \cdot IDF$, where TF is the metadata value of the metadata term, and IDF is $\log(\frac{N}{DF})$, where N is the total number of payloads stored within the mote, and DF is the number of payloads which share the same metadata term. The relevancy of a payload, or the score of the payload, is the

combined weights of the metadata terms matching the search terms.

For example, let Microsearch contain a total of 5 payloads. One of the payload $p1$ have tuples $(term_1, 3), (term_2, 2)$, and another payload $p2$ have a tuple $(term_1, 6)$. The remaining payloads do not contain terms $term_1$, $term_2$. A user issues a query $(term_1, term_2)$. Clearly, the ideal answer should be $p1$ since it is the only payload that contains both terms. Using TF/IDF, the weight of $p1$ will be

$$3 \cdot \log \frac{5}{2} + 2 \cdot \log \frac{5}{1} = 5.96,$$

and weight for $p2$ will be

$$6 \cdot \log \frac{5}{2} = 5.49.$$

From the calculation, we see that $p1$ has a larger final score than $p2$ despite $p2$ having a larger term score 6.

5.2 Basic Algorithm

In the basic algorithm, Microsearch first obtains a set of metadata entries with metadata terms which match the search terms. Remember that a metadata entry is of the form $(term, value, address)$. With this chosen set of metadata entries, Microsearch then ranks the payload addresses in order of their relevancy, and uses the top ranking addresses to retrieve the payloads to return to the user. Since each payload has a unique flash memory address, this address is used as an identifier for a payload.

To obtain the set of metadata entries, Microsearch first scans all the metadata entries in the buffer cache for metadata terms matching the search terms. Matching entries are then copied to a separated section of RAM. Next, Microsearch uses the inverted index to find matching metadata entries in flash. Microsearch first applies the hash function to each search term to determine the corresponding slot in the inverted index. These slots contain the addresses of the metadata pages in flash memory. Each metadata page contains metadata terms which hash to the same slot. Note that the metadata terms found in the same page do not necessarily have to be the same. They only need to hash to the same slot. Microsearch then retrieves each metadata page one at a time until all metadata pages are read. For each metadata page read, Microsearch compares the actual metadata terms to the search terms, and copies the matching ones to RAM.

At this point, Microsearch has a list of all metadata entries which match the search terms. Microsearch uses the TF/IDF function to determine the score for each payload address, and orders them from the highest score to the lowest. Microsearch then uses the top k payload addresses to obtain the actual payloads from flash to return to the user.

5.3 Performance Improvements

The basic algorithm first selects all the metadata entries which match the search terms, and then proceeds to eliminate low scoring payload address. This approach requires a large section of RAM to be set aside. A better solution is to eliminate low scoring payload addresses as they are encountered.

There are two difficulties in deriving a better solution. First, Microsearch relies on TF/IDF calculations to determine the relevancy of each payload address. Calculating the IDF requires knowledge of DF, the number of payloads in flash which share the same metadata term. The DF score can only be obtained by reading in every metadata page from flash and checking the corresponding metadata terms. We cannot maintain a DF score for each inverted index slot since there could be multiple metadata terms that hash to the same slot.

Second, even if we use only TF score without IDF, a simple elimination scheme does not work. Consider the example when a user queries Microsearch with two search terms x and y, with $k = 1$. For simplicity, we assume that the buffer cache is empty, and x, y hash to different slots in the inverted index, i.e. $hash(x) \neq hash(y)$. We have 10 metadata pages each in flash memory matching $hash(x)$ and $hash(y)$. Now after reading in the first metadata page for x, we obtain 2 metadata entries with x. This means there are two potential payload addresses which can satisfy the user's query. Let us denote these two addresses as $addr_1$ and $addr_2$. The first metadata page for y does not contain either $addr_1$ or $addr_2$. At this point, even though the user specifies the top-1 answer, we cannot eliminate $addr_1$ or $addr_2$ because we cannot determine whether either payload address actually contains the term y. The reason is that Microsearch does not guarantee that metadata from the same payload are evicted from the buffer cache at the same time. To be sure with $addr_1$ or $addr_2$ contain y, we have to continue reading in the metadata pages for $hash(y)$ from flash.

5.4 Space Efficient Algorithm

To derive a space efficient algorithm, Microsearch exploits the sequential write behavior of log file system. This sequential behavior ensures that data is always written to the flash memory in a forward order. This means that if payload $p1$ is sent to the mote before payload $p2$, then the flash address of $p1$ will be smaller than that of $p2$.

To describe the space efficient algorithm, we first define some notations. We let t be the number of search terms and a user query is $\{k, \{st_1, st_2, \ldots, st_t\}\}$. We denote the inverted index as $InvIndex$, and the latest metadata page written to flash memory as the head metadata page. For example, $InvIndex[hash(st_i)]$ returns the address of the head metadata page for st_i. We represent this value as $head[i]$.

We allocate a memory space $page[i]$ for each query term st_i, which is sufficient to load one metadata page from flash memory. We first check the buffer and load the metadata entries whose metadata term is st_i to $page[i]$. If st_i is not found in the buffer, we load $head[i]$ to $page[i]$. Let $\min(page[i])$ and $\max(page[i])$ denote the smallest and largest payload addresses in $page[i]$ respectively. We define a *cutoff* value as the maximum value among $\min(page[i])$, i.e.

$$cutoff = \max\{\ \min(page[i]), \forall i \in [1, t]\ \}.$$

Due to the following lemma 5.1, we have all necessary information to calculate the IR scores for the loaded index entries, whose payload address is greater than or equal to *cutoff*. The entire algorithm is found in Algorithm 1.

LEMMA 5.1. *For any index entry whose payload address* $\geq$ cutoff*, if its term field is included in the query terms, it must have been loaded into memory.*

PROOF. It can be proved by contradiction. Assume there exists such an index entry whose term is one of the search terms st_i, and payload address is $p \geq cutoff$. In addition, the metadata page it belongs to has not been loaded yet. It means that the contents in $page[i]$ are from some preceding metadata page in the same chain, i.e., the loaded metedata page is closer to the chain head. For example, in Figure 3(b), the metedata page at $addr_{26}$ precedes the page at $addr_{17}$ in the same chain. According to our protocol, if metedata page a precedes metadata page b, page b must be flushed into flash earlier than page a. Thus, any payload address in a must be larger than any payload address in b. Based on our hypothesis, therefore, we can derive that any payload address in $page[i]$ must be larger than p, thus

$$\min(page[i]) > p \geq cutoff.$$

It is a contradiction with the definition of *cutoff*, which implies $\forall i \in [1, t], \min(page[i]) \leq cutoff$. □

A k-length array $result[k]$ is used to store the intermediate results which are the candidates of final reply. Every time we get a new IR score, this array will be updated to keep the current top-k results. The processed index entries will be eliminated from memory. When $page[i]$ is empty, we load the next metadata page in the chain from flash memory and repeat this process. Based on the definition of *cutoff*, there must be at least one $page[i]$ becoming empty after each iteration. The algorithm terminates when $\forall i, page[i] = \phi$ and every chain reaches its tail. In this design, instead of loading every metadata page, we load at most one page for each query term. Thus, the memory space needed is at most $O(E \cdot t)$, where E is the size of a metadata page.

Note that in practice we traverse each index chain twice, the first time to obtain the DF for the term, and the second time to execute the actual query algorithm. This is done to match the DF definition in the simple TF/IDF scoring algorithm adopted for this paper. If alternative scoring algorithms that do not require this form of IDF calculation are used, this extra traversal can be avoided.

6. THEORETICAL MODEL

A key parameter in designing Microsearch is the size of the inverted index. We first present the intuition behind the choice of inverted index size, followed by the theoretical model.

With a smaller inverted index, uploading information into Microsearch is faster. When the buffer cache is full, Microsearch evicts data from the buffer cache into flash memory. Microsearch groups all the metadata terms which hash to the same inverted index slot together for eviction. Recall that writing data to flash memory occurs on a page granularity. In other words, the cost of writing a page into flash memory is the same even in situations where there are not enough metadata terms hashing to the same inverted slot to make up a flash page. A smaller inverted index results in more metadata terms hashing to the same inverted index slot. This increases the probability of more entries being flushed out of the buffer cache each time.

Algorithm 1 Reply Top-k Query:

```
1: Input: k, {st_1, st_2, ..., st_t}
2: Output: k-length array result
3: head[i] = InvIndex[hash(st_i)]
4: Scan buffer and each relevant metadata page chain to accumulate the document
   frequency (df[i])
5: Load relevant index entries in buffer to the buffer page page[i]
6: If page[i] is empty, load Flash(head[i]) and move head[i] to the next page
7: while there exists a non-empty page[i] do
8:    cutoff=max{min(page[i]), ∀i ∈ [1, t]}
9:    for non-empty page[i] and max(page[i]) ≥ cutoff do
10:      for every entry e ∈ page[i] and e ≥cutoff do
11:         score = calScore(e)
12:         if score>minimum score in result then
13:            replace the entry with the minimum score in result by {e, score}
14:         for j = 1 to t do
15:            remove e from page[j]
16:   for i = 1 to t do
17:      if page[i] is empty then
18:         load Flash(head[i]) to page[i]
19:         move head[i] to the next metadata page
20: return result
```

Table II. System Model Variables

# of documents	D
# of metadata per document	m
# of query terms	t
Size of main index (# of slots)	H
Size of metadata page (# of metadata terms)	E
# of actual metadata terms per metadata page	E'
Size of buffer (# of metedata terms)	B
# of terms flushed per eviction	x

With a larger inverted index, query performance may be improved because each index slot will have fewer metadata terms hashing to it. As a result, the chain of metadata pages in flash memory which map to each inverted index slot becomes shorter. When replying to a query, Microsearch has to read in the entire chain of metadata pages. A shorter chain of metadata pages means that fewer pages are needed to be read from flash memory, and thus speeding up query performance. Next, we first analyze a important parameter, the number of metadata terms in each metedata page. Then we derive query performance and insert performance of Microsearch. Table II lists some variables in our model.

Analysis of E' and x: Although each metadata page can hold E metadata terms, the actual number of terms in each metadata page (E') might be less than E. It depends on the number of metadata terms Microsearch flushes to the flash when the buffer is full. Recall our eviction process described in Section 3, the largest group of tuples that hash to the same index slot will be evicted to the flash. Let x

denote the number of metadata terms in this largest group, i.e., every eviction could put x terms to the flash. According to our protocol, if $x > E$, Microsearch will write multiple metadata pages in the flash, where the last page contains (x mod E) terms and the other pages are full with E terms. Thus, in this case, $E' = \frac{x}{\lceil \frac{x}{E} \rceil}$. Otherwise, if $x \leq E$, Microsearch will write at most one new metadata page each eviction. Recall our eviction process will first attempt to pad the last written metadata page. As a result, every metadata page will contain $\lfloor \frac{E}{x} \rfloor \cdot x$ terms. In summary,

$$E' = \begin{cases} x/\lceil \frac{x}{E} \rceil & \text{if } x > E; \\ \lfloor \frac{E}{x} \rfloor \cdot x & \text{if } x \leq E. \end{cases}$$

Next, we give an analysis of deriving the value of x. Based on its definition, x is obviously at least $\lceil \frac{B}{H} \rceil$. For one hashed value h_i, the probability that p entries in the buffer map to h_i is

$$\binom{B}{p} \left(\frac{1}{H}\right)^p (1 - \frac{1}{H})^{(B-p)}.$$

Thus, the probability that at least p entries map to h_i is

$$q = \sum_{j \geq p} \binom{B}{j} \left(\frac{1}{H}\right)^j (1 - \frac{1}{H})^{(B-j)}.$$

The probability that $x \geq p$ is $P(x \geq p) = 1 - (1 - q)^H$. Thus,

$$P(x = p) = P(x \geq p) - P(x \geq p + 1).$$

Therefore, the expected value of x is

$$x = \sum_{i \geq \lceil \frac{B}{H} \rceil}^{B} P(x = i) \cdot i.$$

The following Figure 4 illustrates an example of x and E'.

Query Performance: Assume there are D number of files stored in the flash memory and each of them is described by m terms on average. Totally, we need store $D \cdot m$ index entries in the flash, which occupy $\frac{D \cdot m}{E'}$ metadata pages. Considering a fair hashing, the average length of metadata page chain is $\frac{D \cdot m}{E' \cdot H}$. When Microsearch processes a query for one term, based on the hash value of the term, it has to go through one of the metadata page chain twice. One round for collecting the value of document frequency and the other for finding the top-k answers. Expectedly, Microsearch will need to read $\frac{2 \cdot D \cdot m}{E' \cdot H}$ metadata pages from the flash. For a query for t terms, Microsearch has to access t distinct metadata page chains, when $t \ll H$. Thus, it takes at most $\frac{2 \cdot t \cdot D \cdot m}{E' \cdot H}$ page reads to reply. The following Figure 5 illustrates an example of query performance.

Insert Performance: Insert performance is measured by the number of reads and writes operated during inserting D files. Microsearch only reads once in each buffer eviction. As we mentioned, each eviction flushes x metadata terms. Since in total there are $D \cdot m$ metadata terms, the number of reads for insertion is $\frac{D \cdot m}{x}$. The number of writes must be no less than the number of reads. If $x \leq E$, each

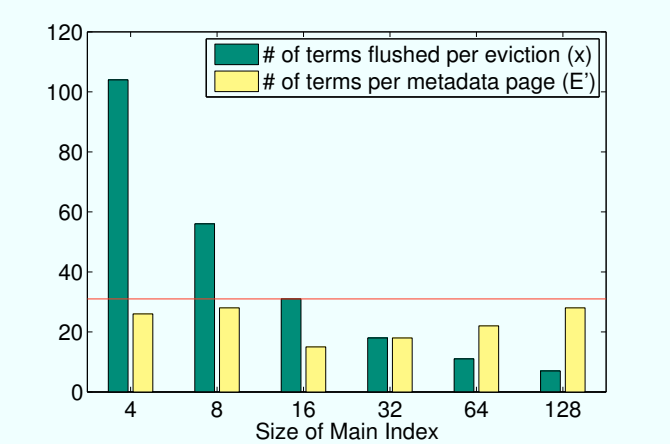


Fig. 4. Value of x and E' v.s. Index Size (H): We allocate $5K$ bytes to buffer cache. Thus, buffer size B is set to $B = 640$. The flat line illustrates the value of $E = 31$.

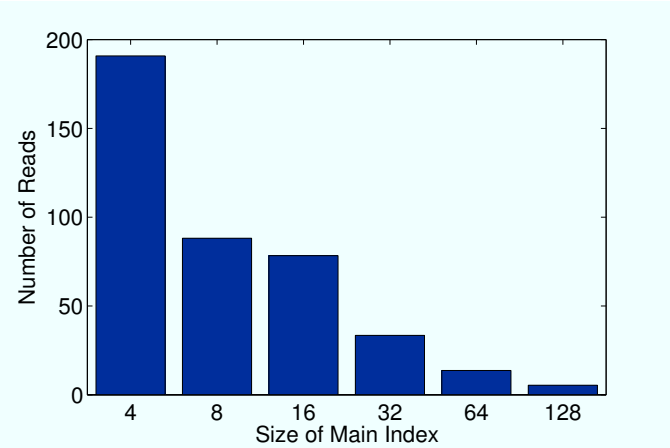


Fig. 5. Query Performance v.s. Index Size (H): We set $D = 1000, m = 10, t = 1, E = 31, B = 640$ ($5K$-byte buffer).

eviction only write on metadata page, thus the number of writes is the same as the number of reads. If $x > E$, however, Microsearch will write multiple pages in each eviction process. Since each metadata page contains E' terms, the number of writes is $\lceil \frac{D \cdot m}{E'} \rceil$. The following Figure 6 illustrates an example of query performance.

7. SYSTEM EVALUATION

7.1 Hardware and Implementation

We used the TelosB mote for our experiments. TelosB features a 8MHz processor, 10KB RAM, 48KB ROM and 1MB of flash memory. An IEEE 802.15.4 standard radio is used for wireless communication. The entire package is slightly larger, measuring 65x31x6 mm, and weighs 23 grams without the battery. We implement

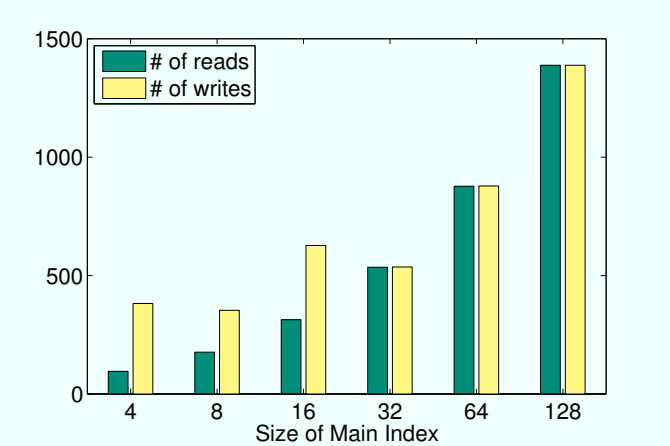


Fig. 6. Insert Performance v.s. Index Size (H): We set $D = 1000, m = 10, E = 31, B = 640$ ($5K$-byte buffer).

Microsearch using NesC in TinyOS environment and a user interface using Java. The core program takes around 1700 lines of NesC codes and the interface takes around 800 lines of Java codes.

From Table II, the parameters related to implementation include the size of main index (H), the number of metedata terms in each metadata page (E), and the size of buffer cache (B). In our implementation, we set $H = 32$, $E = 31$, and $B = 372$. Each entry in H is three bytes. Each metadata entry, $(term, value, address)$, is eight bytes. We use the first five bytes to store the term, three bits to store the value field, and the rest for the address field. The total amount of space allocated to the buffer and index is 3K memory.

7.2 Generating Workload Data

A difficulty in evaluating a search system lies in determining an appropriate workload. An ideal workload should consists of traces derived from real world applications. However, since Microsearch-like applications do not yet exist, we cannot collect such traces for evaluation. This also makes it difficult to generate synthetic traces that approximate user behavior. We generated our workload by observing related real world applications.

We envision that most objects such as a wedding photograph album or a document binder will embed a mote running Microsearch. Since each object has its own mote, each mote does not necessarily have to contain a large amount of unique data. For instance, a large bookshelf may contain hundreds of document binders, with a combined total of thousands of documents. However, each binder may contain only a dozen documents. Since each binder embeds a mote, each mote only needs to index the contents of its own binder. Consequently, none of our workload considers excessive large number of unique pieces of data. Our evaluation considers the following two workloads.

Annotation workload: This workload represents a user storing many short pieces of information, similar to Post-it reminders or memos, onto a mote. The

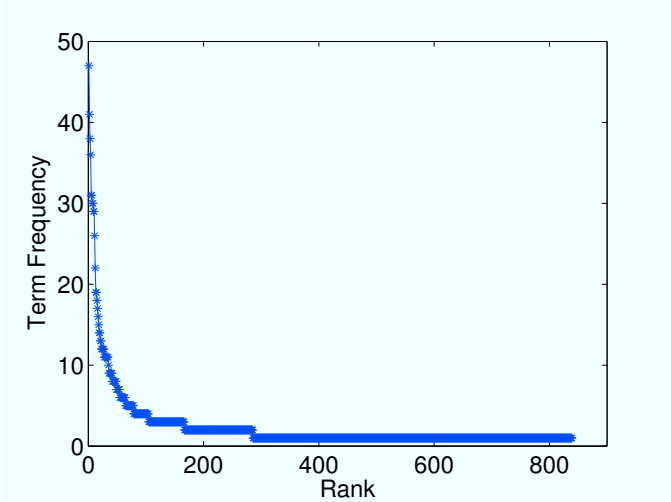


Fig. 7. Term distribution for annotation workload

metadata a user would associate with these type of applications is usually very short. We want a real world application where many users provided annotations, since this closely resembles the metadata we desire. One such application is the annotation of online photographs. We extracted 622 photographs and their accompanying annotations from the website `www.pbase.com`. This created a set of 2059 metadata terms, an average of 3.3 metadata terms per photograph. We consider each photograph as a unique input, and each photograph's annotation as the corresponding metadata terms. The metadata value of each term is set to 1. Figure 7 shows the metadata term distribution for this workload. Recall in Table II, the parameters related to workload are $D = 622$ and $m = 3.3$.

Doc workload: This workload represents a mote used for tracking purposes, such as keeping track of the documents inside a binder. We assume that the binder contains academic publications, and the accompanying mote contained the abstracts of all the papers. A user can query Microsearch just like querying *Google Scholar* to determine if a particular paper is inside the binder. To create the doc workload, we extracted 21 papers from the conference proceedings of Sensys 2005, and derived an average of 50 metadata terms for each paper. The metadata terms include author names, paper title, keywords. Metadata values are based on the number of times each term appeared in the paper abstract. Recall in Table II, the parameters related to workload are $D = 21$ and $m = 50$.

7.3 System Performance

We use the annotation workload to evaluate system performance. The objective is to determine the performance of the two main Microsearch components: indexing the data sent by a user, and replying a user query. Time is the main metric used. In addition, for every evaluation, we present both the actual measured performance, and the predicted performance derived from our theoretical model introduced earlier. The closer the predicted results match the actual results, the more accurate

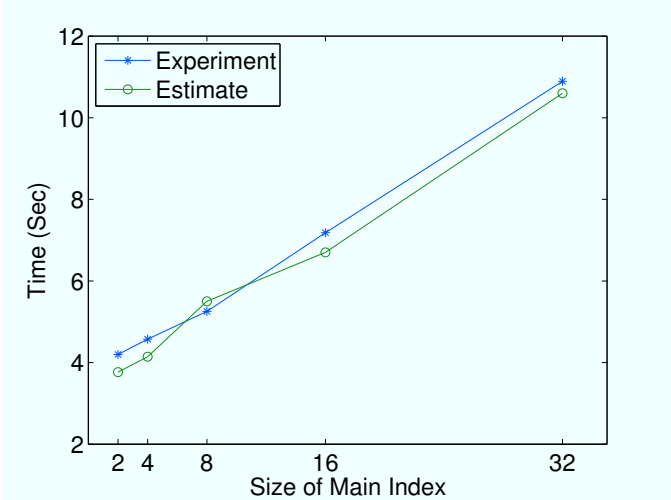


Fig. 8. Predicted and actual insert performance

our theoretical model is.

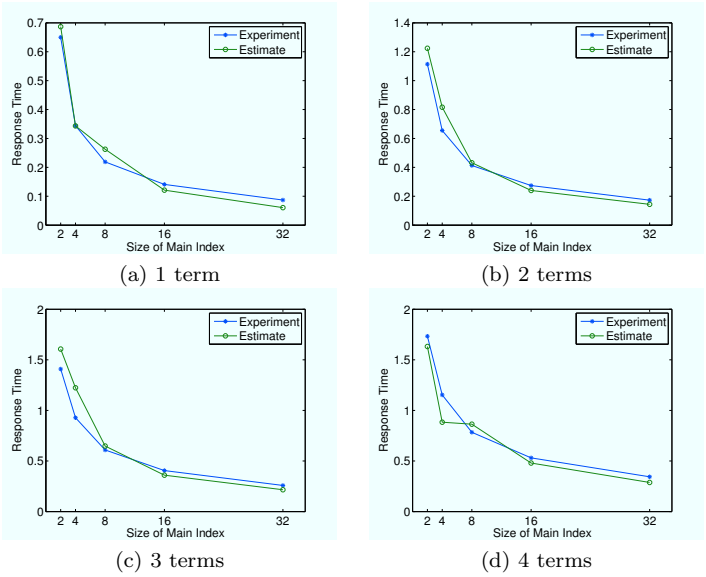
To prepare, we first generate a set of queries by randomly choosing terms from the 2059 harvested annotations. We then divided the set of queries into four groups, with the first group containing queries with one search term, the second group with queries containing two search terms and so on. Each group has a total of 100 queries. We limit the number of search terms to at most four terms, since studies conducted on mobile search conclude that most searches consists of between 2 and 3 terms [Church et al. 2007; Baeza-Yates et al. 2007; Kamvar and Baluja 2006].

We then inserted the 622 metadata files with a total of 2059 metadata terms into Microsearch. This is equivalent to inserting 622 short messages into the mote. Figure 8 shows the time taken to insert all the terms into Microsearch. We see uploading information is faster given a smaller inverted index. This is consistent with the intuition given in the prior section.

In Fig 9, we show the time taken for Microsearch to answer a user's query. As discussed in the theoretical model, we see that a larger inverted index processes queries faster than a smaller inverted index. The predicted query response time is also very close to the measured time. Overall, Microsearch is able to answer a user's query in less than two seconds. Fig 10 shows the actual overhead of Microsearch minus the time taken to read from flash memory. We see that the additional time taken to rank the query answers is less than 0.5 seconds.

7.4 Search Accuracy

The precision verses recall metric is commonly used to evaluate search systems. Precision is defined as the number of relevant items retrieved divided by the total number of retrieved items. Recall is the number of relevant items retrieved divided by the total number of relevant items in the collection. A better search system will consistently returns a higher precision for any given recall rate. However, the precision verses recall metric does not measure how well the search system *ranks*



(a) 1 term (b) 2 terms

(c) 3 terms (d) 4 terms

Fig. 9. Predicted and actual query response time measured in seconds

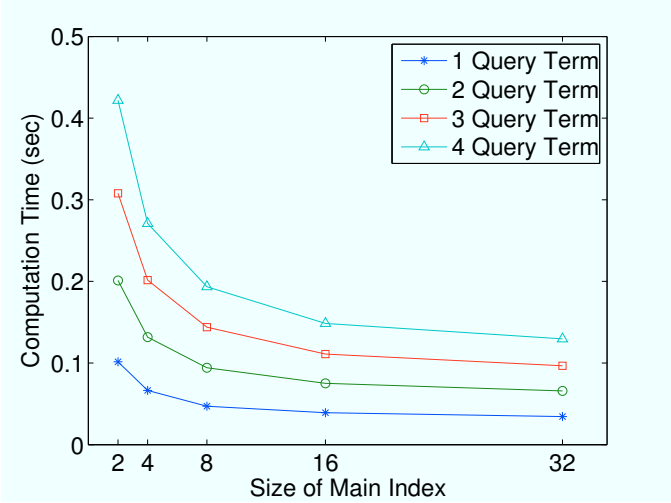


Fig. 10. Processing time overhead of search system processing

the results.

Shah and Croft [Shah and Croft 2004] suggested using metrics from question answering (QA) research [Voorhees 2001] to evaluate search algorithms for bandwidth or power constrained devices. QA is a branch of information retrieval that returns answers instead of relevant documents in response to a query. In QA research, the goal is to return a single or a very small group of answers in response to a query, not all relevant documents. The main evaluation in QA is the mean reciprocal rank

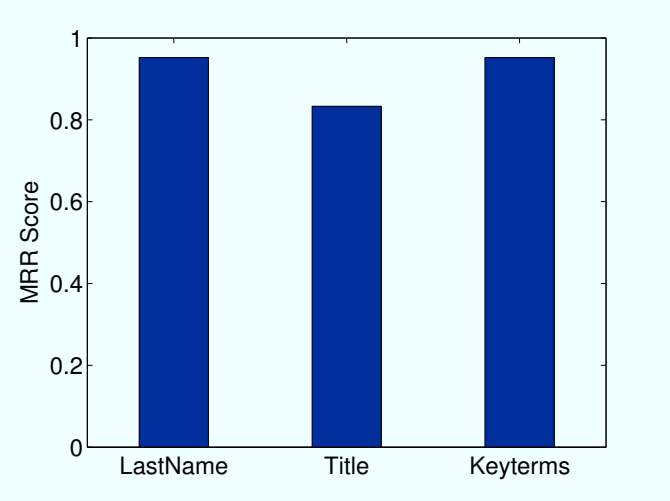


Fig. 11. Query accuracy (k=3)

(MRR). MRR is the calculated as

$$MRR = \frac{1}{\text{rank of first correct response}}.$$

The first correct response is the top ranked document in the model answer. For example, if the model answer to a query is the ranked list (A, B, C) and IR system returns the list of (C, B, A). The first correct answer should be A and the returned answer is 2 spots off. The MRR for this question is thus $\frac{1}{3} = 0.33$. We evaluate the performance of our search system by modifying the guidelines for QA track at TREQ-10 [Chen et al. 2001]. We consider only the top 3 answers in calculating MRR. If the model answer does not appear within the top three ranks, it has a score of 0.

We use the doc workload to evaluate the accuracy of Microsearch. We first determine a set of queries based on the 21 publications, and their corresponding answers by hand. These questions are divided into three groups, $LastName$, $Title$ and $KeyTerms$. The queries for the first two categories are terms from the last names and paper titles of the conference proceedings. The queries for the last category are a mixture of terms from last names, titles and abstract keywords.

Figure 11 shows the results of our search system for the three categories. For each category, we plot the MRR for the different categories over the average of 21 questions. From the figure, our system returns a MRR of 0.95 for both $LastName$ and $KeyTerms$. The MRR for $Title$ is lower at 0.83, because some of the paper titles contained very common words like "Packet Combining In Sensor Networks". In all cases, we see that on average Microsearch will return the correct answer when the user specifies $k = 3$.

7.5 Experiment Limitations

We stress that the results shown in Figure 11 do not suggest that Microsearch will yield similar accuracy results in a real deployment. The reason for the very high accuracy results is that we have deliberately avoid using vague queries since it is difficult to objectively quantify what the answer *should* be. Instead, we first generated a set of queries which contain terms that are found in multiple documents, and then manually determine what the correct answers to those queries. In all instances, the answers we select are unambiguous. For instance, given a query "underwater sensor storage". There is only one paper containing the term "underwater", and three papers containing the term "storage". Almost all papers contained the word "sensor". The correct answer is should be the only paper on underwater sensors despite the two other papers containing more occurrences of the term "storage".

Microsearch uses TF/IDF calculation to resolve queries, a conventional weighing algorithm widely used in information retrieval research. Our contributions are the space saving algorithm (Algorithm 1) which uses less memory space to compute TF/IDF, and that the results in Figure 11 can only be interpreted as showing the correctness of applying our space saving algorithm technique in determining TF/IDF, and not the accuracy of Microsearch.

7.6 Model Accuracy

As we expect higher level applications to be built above our low level search system, the predictability of our system is an important performance metric. Since sensors, as embedded devices, have more stringent resource limitations, it is important when developing applications to be able to accurately budget the sensor resource. A low level component that is gives unpredictable performance will adversely impact sensor application design and deployment. To evaluate our performance, we give a set of requirements and derive the expected performance based on our model. Then, we modify our prototype based on these requirements and compare the experimental results against the expected results. The closer the match, the better the predictability of our system. In all cases, we limit the available RAM for main index and buffer size to $3KB$. Table III shows our requirements and recommendations. For a given targeted query response time and expected query term length, the model provides a recommendation of size of index. The buffer cache size is then $3KB$ subtracted from the index size. An x indicates our search system cannot meet the targeted query response time given the hardware requirements. Figure 12 shows the results. We see that the experimental results are sightly higher than the targeted query response time when the number of query terms is expected to be larger. The difference is less than 0.05 seconds.

7.7 Alternative Design

An alternative system design is to not use an inverted index at all. The incoming metadata is buffered and flushed to flash when there are enough entries to make up a full metadata page. Each metadata page will contain a pointer to the previous metadata page in flash. A single entry kept in memory remembers the latest metadata page's location in flash. When querying, Microsearch accesses every metadata page in flash before replying since every metadata page could contain a payload

Table III. Recommended size of main index (H) for different query response requirements ($B \times 8 + H \times 3 = 3K$ bytes).

Groups	1 term	2 terms	3 terms	4 terms
$1(0.1s)$	16	156	x	x
$2(0.2s)$	8	16	68	156
$3(0.3s)$	6	11	16	46
$4(0.4s)$	4	8	12	16
$5(0.5s)$	4	7	10	13

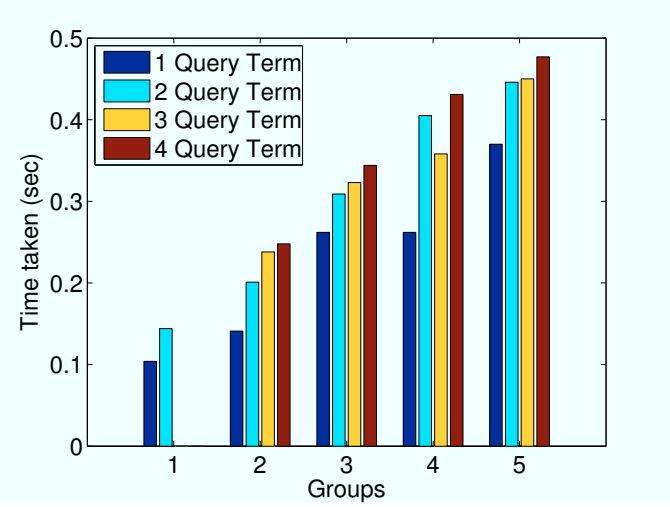


Fig. 12. Actual query response time.

matching the query terms. The intuition is that such a scheme will have a better indexing performance at the expense of worse query performance.

To evaluate, we used a $3KB$ memory limit. The alternative design will allocate all as much space as possible to the buffer cache, and have just one main index entry. Microsearch uses a balanced approach, using an inverted index size of $H = 32$ (96 bytes), and a buffer cache of $B = 372$ (2976 bytes). The alternative system takes an average of 6.5 ms to insert the metadata in one file compared to the 17.5 ms for our scheme. Figure 13 shows the difference in query response time for different number of query terms. Next, we compare the energy consumption between our scheme and the alternative scheme. Since both schemes have to do the same amount of writing for the payload data given the same document set, our comparison only measures the energy consumption of metadata input and query. Let P_w and P_r be the energy consumption for writing and reading one page data in flash memory respectively. Given the input insertion frequency f_u and user query frequency f_q, the energy consumption is determined by the amount of metadata writing during the input insertion period and the amount of metadata reading during the query period. For the simplicity, we ignore the energy consumption of CPU processing because that part is much smaller compared with the flash memory read and write operations. On a per unit time basis, the energy consumption of our scheme can

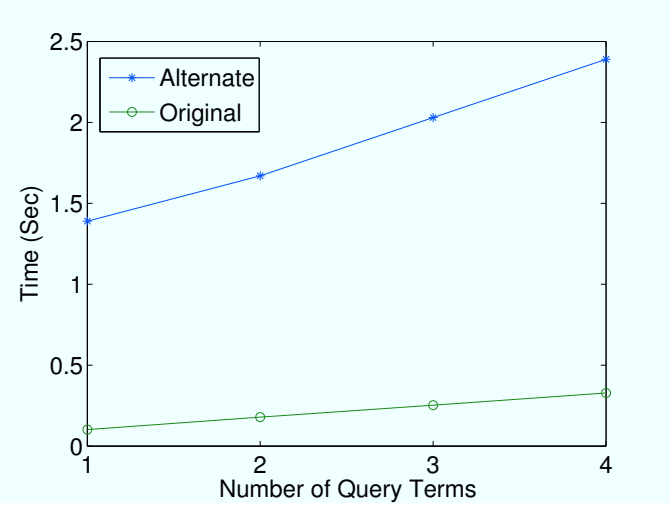


Fig. 13. Comparing alternative scheme with our scheme

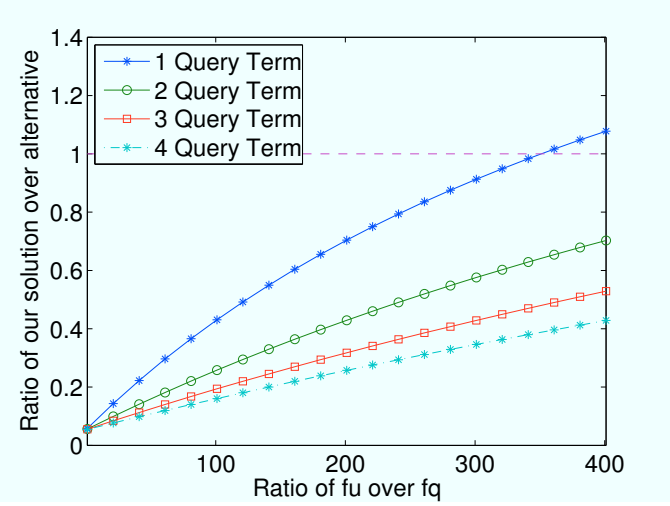


Fig. 14. Comparing power consumption of our scheme verses alternative scheme

be expressed as $E_1 = f_u \cdot (W_i \cdot P_w + R_i \cdot P_r) + f_q \cdot R_q \cdot P_r$, where W_i and R_i are the numbers of write and read operations for insertion, and R_q is the number of reads required for the query. From Section 5, we have $W_i = \lceil \frac{D \cdot m}{E'} \rceil$, $R_i = \frac{D \cdot m}{x}$, and $R_q = \frac{2 \cdot D \cdot m \cdot t}{E' \cdot H}$.

For the alternative scheme, we consider a more efficient insertion process without reading the last flushed metadata pages. Instead, new metadata pages will be directly written to the flash. Therefore, the energy consumed by the alternative scheme can be expressed as $E_2 = f_u \cdot W_i' \cdot P_w + f_q \cdot R_q' \cdot P_r$, where $W_i' = \lceil \frac{D \cdot m}{E} \rceil$ and $R_q' = \frac{2 \cdot D \cdot m \cdot t}{E}$. With the system parameters fixed at $D = 622, m = 3.3$ and

$H = 32$, we estimate the energy consumption for both schemes based on TelosB flash memory read and write energy performance presented in [Mathur et al. 2006b] (i.e., $P_w = 0.127 \times 256 = 32.5\mu J$, $P_r = 0.056 \times 256 = 14.3\mu J$).

To compare our scheme with the alternative, we find the ratio of $\frac{E_1}{E_2}$. Values less than 1 favor our solution while values larger than 1 favor the alternative. To simplify the results, we divide both E_1 and E_2 by f_q, which does not affect the ratio. As a result, $\frac{E_1}{E_2}$ becomes a function of $\frac{f_u}{f_q}$. We plot the energy ratio graph with 1, 2, 3 and 4 query terms respectively. The estimation results are found in Figure 14. The figure shows that for an average of 1 query term, the alternative performs better when there are about 350 document insertions to a single query. For other cases, our scheme is always superior to the alternative solution. This suggests that the alternative scheme should be used only when the mote is used to store data and rarely if ever queried.

8. CONCLUSION

In this paper, we present a search system for small devices. Our architecture can index an arbitrary number of textual metatdata efficiently. A space saving algorithm is used in conjunction with IR scoring to return the top-k answers to the user. Our experimental results show that Microsearch is able to resolve a user query of up to four terms in less than two seconds, and provide a high level of accuracy.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments, as well as Dr Joerg Henkel and Martin Buchty for their kind assistance.

REFERENCES

Abowd, G. D., Atkeson, C. G., Hong, J., Long, S., Kooper, R., and Pinkerton, M. 1997. Cyberguide: a mobile context-aware tour guide. *Wirel. Netw. 3,* 5, 421–433.

Apple. 2007. http://www.apple.com/macosx/features/spotlight/.

Baeza-Yates, R., Dupret, G., and Velasco, J. 2007. A study of mobile search queries in Japan. In *Query Log Analysis: Social and Technological Challenges, at WWW 2007*.

Beagle. 2007. http://beagle-project.org/*main_page*.

Chen, J., Diekema, A., Taffet, M. D., McCracken, N. J., Ozgencil, N. E., Yilmazel, O., and Liddy, E. D. 2001. Question answering: CNLP at the TREC-10 question answering track. In *Text REtrieval Conference*.

Cheverst, K., Davies, N., Mitchell, K., and Friday, A. 2000. Experiences of developing and deploying a context-aware tourist guide: the guide project. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM, New York, NY, USA, 20–31.

Cheverst, K., Davies, N., Mitchell, K., Friday, A., and Efstratiou, C. 2000. Developing a context-aware electronic tourist guide: some issues and experiences. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, New York, NY, USA, 17–24.

Church, K., Smyth, B., Cotter, P., and Bradley, K. 2007. Mobile information access: A study of emerging search behavior on the mobile internet. *ACM Trans. Web 1,* 1, 4.

Company, A. 2008. Yaffs: Yet another flash file system. In *http:/www.yaffs.net/*.

Dai, H., Neufeld, M., and Han, R. 2004. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, New York, NY, USA, 176–187.

Faloutsos, C. 1985. Access methods for text. *ACM Comput. Surv. 17,* 1.

Faloutsos, C. and Oard, D. W. 1995. A survey of information retrieval and filtering methods. Tech. Rep. CS-TR-3514, University of Maryland at College Park.

Frakes, W. B. and Baeza-Yates, R. A., Eds. 1992. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall.

French, J. C., Powell, A. L., Callan, J. P., Viles, C. L., Emmitt, T., Prey, K. J., and Mou, Y. 1999. Comparing the performance of database selection algorithms. In *Research and Development in Information Retrieval*.

Gal, E. and Toledo, S. 2005a. Algorithms and data structures for flash memories. *ACM Comput. Surv. 37,* 2.

Gal, E. and Toledo, S. 2005b. A transactional flash file system for microcontrollers. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 7–7.

Google. 2007. www.desktop.google.com.

Kamvar, M. and Baluja, S. 2006. A large scale study of wireless search behavior: Google mobile search. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, New York, NY, USA, 701–709.

Kobayashi, M. and Takeda, K. 2000. Information retrieval on the web. *ACM Comput. Surv. 32,* 2, 144–173.

Logitec. 2004. www.logitech.com.

Mathur, G., Desnoyers, P., Ganesan, D., and Shenoy, P. 2006a. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, New York, NY, USA, 195–208.

Mathur, G., Desnoyers, P., Ganesan, D., and Shenoy, P. 2006b. Ultra-low power data storage for sensor networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*. ACM, New York, NY, USA, 374–381.

Nachman, L., Kling, R., Adler, R., Huang, J., and Hummel, V. 2005. The intel®mote platform: a bluetooth-based sensor network for industrial monitoring. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*. IEEE Press, Piscataway, NJ, USA, 61.

Pucheral, P., Bouganim, L., Valduriez, P., and Bobineau, C. 2001. Picodbms: Scaling down database techniques for the smartcard. *The VLDB Journal 10,* 2-3, 120–132.

Rekimoto, J., Ayatsuka, Y., and Hayashi, K. 1998. Augment-able reality: situated communication through physical and digital spaces. *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, 68–75.

Shah, C. and Croft, W. B. 2004. Evaluating high accuracy retrieval techniques. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, New York, NY, USA, 2–9.

Starner, T., Kirsch, D., and Assefa, S. 1997. The locust swarm: an environmentally-powered, networkless location and messaging system. *Wearable Computers, 1997. Digest of Papers., First International Symposium on*, 169–170.

Tan, C. C., Sheng, B., Wang, H., and Li, Q. 2008. MicroSearch: When search engines meet small devices. In *Pervasive*. Sydney, Australia, 93–110.

Voorhees, E. M. 2001. Overview of the trec 2001 question answering track. In *In Proceedings of the 10th Text REtrieval Conference (TREC*. 42–51.

Wang, H., Sheng, B., and Li, Q. 2006. Elliptic curve cryptography based access control in sensor networks. *International Journal on Sensor Networks*.

Wang, H., Sheng, B., Tan, C. C., and Li, Q. 2007. WM-ECC: an Elliptic Curve Cryptography Suite on Sensor Motes. Tech. Rep. WM-CS-2007-11, College of William and Mary, Computer Science, Williamsburg, VA.

Wang, H., Sheng, B., Tan, C. C., and Li, Q. 2008. Comparing symmetric-key and public-key based security schemes in sensor networks: A case study of user access control. In *ICDCS '08:*

Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems. IEEE Computer Society, Washington, DC, USA, 11–18.

Wang, H., Tan, C. C., and Li, Q. 2008. Snoogle: A search engine for physical world. In *IEEE Infocom*. Phoenix, AZ, 1382–1390.

Woodhouse, D. 2001. Jffs : The journalling flash file system. In *Proceedings Ottawa Linux Symposium*.

Yap, K.-K., Srinivasan, V., and Motani, M. 2005. Max: human-centric search of the physical world. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. ACM, New York, NY, USA, 166–179.

Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. A. 2005. Microhash: an efficient index structure for fash-based sensor devices. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA, USA, 3–3.

Received May 2008; revised February 2009; accepted March 2009