Microsearch: When Search Engines Meet Small Devices

Chiu C. Tan, Bo Sheng, Haodong Wang, and Qun Li

College of William and Mary, Williamsburg VA, USA,
{cct,shengbo,wanghd,liqun}@cs.wm.edu

Abstract. In this paper, we present Microsearch, a search system suitable for small devices used in ubiquitous computing environments. Akin to a desktop search engine, Microsearch indexes the information inside a small device, and accurately resolves user queries. Given the very limited hardware resources, conventional search engine designs and algorithms cannot be used. We adopt information retrieval techniques for query resolution, and propose a space efficient algorithm to perform top-k query on limited hardware resources. Finally, we present a theoretical model of Microsearch to better understand the tradeoffs in system design parameters. By implementing Microsearch on actual hardware for evaluation, we demonstrate the feasibility of scaling down information retrieval systems onto very small devices.

1 Introduction

Interacting with our physical environment is a key component in many pervasive computing applications [1, 6, 7, 24, 26, 22]. A typical system design usually involves a combination of simple beacons and a more powerful backend server. For example, a simple RF beacon can be embedded into a file binder and programmed to continuously emit a unique ID. Information regarding the documents found in the binder is stored in the backend sever. A user accesses this information by obtaining this ID and returning it with his query to the backend server. Since each ID is unique, the backend server can retrieve all the data associated with this particular binder and resolve the query. A similar process is executed when a user updates information about that binder.

Hardware improvements, which we will elaborate later, allow us to consider a different design paradigm which does not utilize a backend server. Instead of embedding a simple RF beacon into an object, we can embed a more powerful device. Information previously kept on a server will now be stored directly on this device. User queries will also be resolved by the object itself. This new paradigm reduces cost by eliminating the network of backend servers as well as long range wireless infrastructure needed for a user to communicate with the backend server. Short range protocols such as Bluetooth can be used for communication between a user and an object. Storing data on the object itself also simplifies ownership transfer. The physical act of handing over a binder

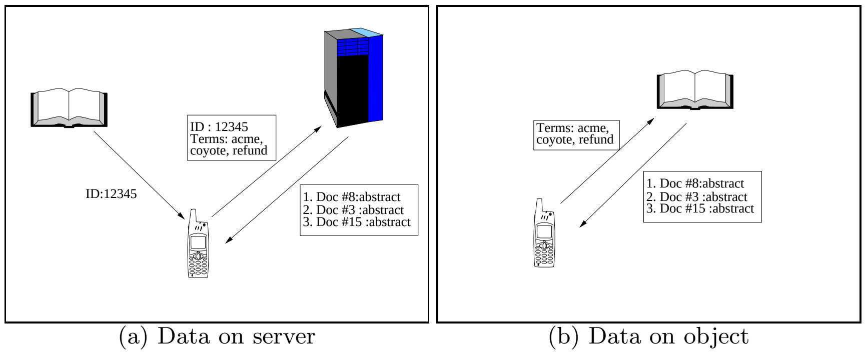


Fig. 1. (a)Typical design utilizing backend server. (b)Different paradigm without use of a server

implicitly completes ownership transfer since any data can only be obtained when the user has physical access to the object. Fig. 1 illustrates the two approaches.

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 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 [19] or PDA, the user can store notes into each binder. Microsearch will index the user input for future retrieval.

Microsearch is designed to run on resource constrained small devices capable of being embedded into everyday objects. One example of such hardware is manufactured by Intel [16] 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 [15], Spotlight [2] or Beagle [4], 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 [28] or MAX [31].

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. Unlike conventional disk, flash memory requires additional processing for I/O operations. Conventional flash file systems [30, 29] cannot be used for this purpose due to the limited hardware resources. This necessitates a different storage design.

We make the following contributions in this paper. (*a*) We provide a system architecture that effectively utilizes limited memory resources to store and index different inputs. (*b*) Our architecture incorporates information retrieval (IR) techniques to determine relevant answers to user queries. (*c*) 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. (*d*) A theoretical model of Microsearch is presented to better understand how to choose different system parameters. (*e*) 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 covers our search algorithms, and Section 5 presents the theoretical model of Microsearch. Section 6 contains our evaluation, and Section 7 concludes.

2 Related Work

Desktop search engines are a mainstream feature found in most modern 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 searches [10]. Information retrieval algorithms [18, 11–13] 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 [23], 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, where rules regarding structured inputs with specified attribute terms can be enforced, and users, e.g. doctors and nurses, are assumed to be well trained in the system. 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 differences between the Microsearch and PicoDBMS can be summed up as the differences between a search engine and a database.

Other embedded search systems can be found in sensor network literature [32, 20, 9]. Sensor networks are a collection of small, embedded devices usually deployed to collect environmental data such as temperature readings or soil humidity values. While sensor systems share a similar hardware platform as Microsearch, they are primarily concerned with indexing and processing numeric data. There appears to be no way for existing sensor search systems to index textual data. In addition, query processing in sensor networks typically returns a range query results or min/max values on collected data. Since the data is numeric, there is no concept of relevancy or ranking. Microsearch differs from sensor systems in that it handles textual in addition to numeric data, and uses IR algorithms to reply to queries.

3 System Architecture

We begin by 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 $\big\{(term_1, value_1), \cdots, (term_n, value_n)\big\}$. For a text based payload, the simplest method to determine the metadata value for a term is to count the number of occurrences of that term in the payload. Metadata values for non-text based payloads can be defined by the user.

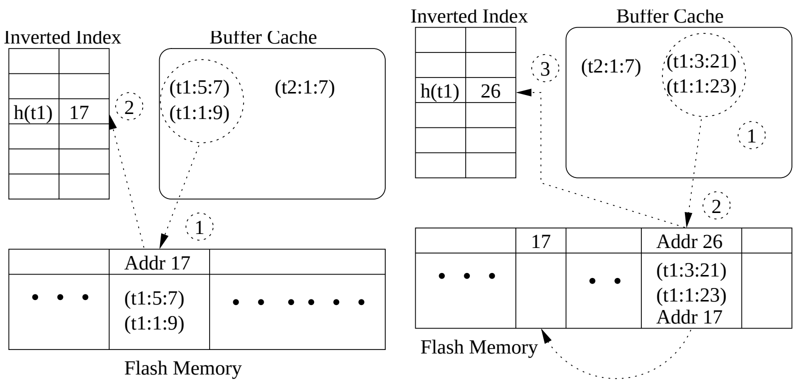
3.1 Microsearch Design

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 recover 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 cache now becomes a tuple , $(term, value, address)$, consisting of a metadata term, a metadata value, and payload address. For example, consider Microsearch writing a payload to flash memory location $addr_3$. All the metadata associate with this payload becomes, $\big\{(term_1, 3, addr_3), \cdots, (term_n, 2, addr_3)\big\}$.

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 [14]. 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 input files 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 metadata. Instead, metadata entries which have the same metadata term 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 $\{(term_1, 3, addr_3), (term_1, 8, addr_8)\}$.



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

Fig. 2. (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 recover archived information. A conventional inverted index has every slot on the inverted index 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 recover all the documents or webpages 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 selects the largest group of tuples, which all share the same metadata term, and applies a hash function to the metadata term to determine a 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. Fig. 2(a) illustrates this process.

Buffer eviction with initialized index slot: In the event that an inverted index slot has already been initialized, Microsearch will copy the address found in that slot onto the *first* metadata page of the group 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 recovered by traversing the links. We consider this *a chain* of metadata pages. Fig. 2(b) illustrates this process.

Data deletion: Once the flash is reaching full capacity, Microsearch simply erases the oldest data to make room. Deletion in flash memory occurs at a sector granularity, with each section usually being 64KB. A pointer is kept by Microsearch to indicate which is the next sector to erase. The deletion does not affect the working of Microsearch since payloads are always written to flash before metadata. Therefore we will not have "orphaned" payloads that exist in flash memory but cannot be retrieved. The next problem is that of entries in metadata pages pointing to invalid payload that have been deleted. This is solved using the deletion pointer. Since this pointer indicates the next sector to erase, the sector that lie before this pointer must have been just erased. Microsearch disregards entries in metadata pages that point to payloads in the sectors behind the delete pointer since they do not exist anymore.

4 Query Resolution

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.

In the basic algorithm, Microsearch first obtains a set of metadata entries which have metadata terms that 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 a simple information retrieval weighing calculation, the TF/IDF function, to determine how relevant each payload address is in satisfying the user's query. TF refers to the term frequency, and IDF refers to the inverse document frequency. 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. After determining the score of the each payload address, Microsearch 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.

4.1 Improving Performance

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. This information can only be obtained by reading in every metadata page from flash and checking the corresponding metadata terms. We cannot maintain a running DF score since each inverted index slot represents the metadata terms which hash to that slot. Without reading in the actual metadata page, we cannot determine what the actual metadata terms are.

Second, even 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 address 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 whether $addr_1$ or $addr_2$ contains y, we have to continue reading in the metadata pages for $hash(y)$ from flash.

4.2 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 written to flash memory is always written 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 $\{k, \{st_1, st_2, \ldots, st_t\}\}$, where st_i is the i^{th} search term. We denote the inverted index as $InvIndex$, and the latest metadata page to be 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 value 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

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

Due to the following lemma 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 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 ancestor in the same chain. An important property of metadata page chain is that if page i is an ancestor of page j, then $min(i) > max(j)$. Thus, $\min(page[i]) > p \geq$*cutoff*. It is a contradiction with the definition of *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 actually 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 calculations are used, this extra traversal can be avoided.

5 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.

With a larger inverted index, query performance is better. A larger inverted index will have fewer metadata terms hashing to each slot. As a result, the chain of metadata pages in flash memory which map to each inverted index slot is

Algorithm 1 Reply Top-k Query:

```
1: Input: $k, \{st_1, st_2, \ldots, st_t\}$
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($page[i]$)}
9:    for non-empty $page[i]$ and max($page[i]$) $\geq$ $cutoff$ do
10:      for every entry $e \in page[i]$ and $e \geq cutoff$ 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 $result$
```

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. The variables used for our model are found in Table 1.

D	# of documents
m	# of metadata per document
t	# of query terms
H	Size of main index
E	Size of metadata page
B	Size of buffer
f_s	Query frequency

Table 1. System Model Variables

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 DF 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.

Insert Performance: Insert performance is measured by the number of reads and writes operated during inserting D files. In our scheme, the number of reads is roughly the same as the number of writes. Microsearch only writes metadata pages to the flash in buffer eviction. Thus, the insert performance depends on the number of flushed entries during each eviction. Let x denote the number, i.e., on average, every eviction puts x index entries to the flash. After inserting all the files, $D \cdot m - B$ entries are written to the flash. Thus, we need $\frac{D \cdot m - B}{x}$ writes for them. Next, we give an analysis of deriving the value of x. According to our scheme, x is the most frequent hashed value when the buffer is full. Obviously, x is 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

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

In total, inserting D files requires $\frac{D \cdot m - B}{E(x)}$ number of writes and the same number of reads.

6 System Evaluation

We use the TelosB mote for our experiments. The TelosB mote 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 $65 \times 31 \times 6$ mm, and weighs 23 grams without the battery.

6.1 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 generating synthetic traces that approximate user behavior difficult. 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 workloads consider excessively large number of unique data.

Our evaluation consists of two workloads. The first is the *annotation workload* which represents a user storing many short pieces of information, similar to Post-it reminders or memos, onto a mote. The 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. Fig. 3 shows the metadata term distribution for this workload.

The second workload is the *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 contains 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.

6.2 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

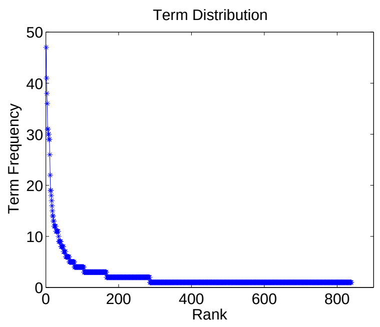


Fig. 3. Term distribution for annotation workload

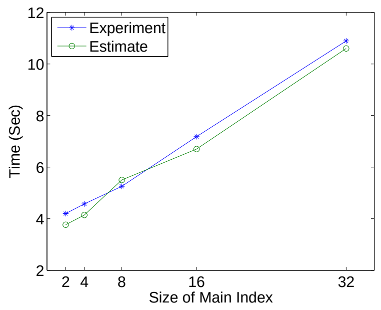


Fig. 4. Predicted and actual indexing performance

introduced earlier. The closer the predicted results match the actual results, the more accurate our theoretical model is.

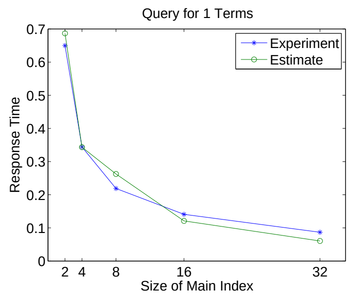
To prepare, we first generate a set of queries by randomly choosing terms from the 2059 harvested annotations. We then divided the set of queries into four groups, with the first group containing queries with one search term, the second group with queries containing two search terms and so on. Each group 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 [8, 3, 17].

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. Fig. 4 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.

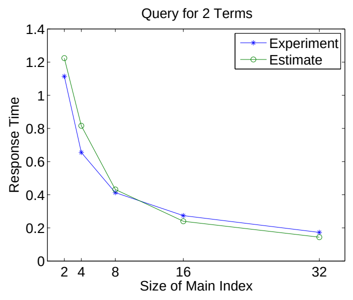
In Fig 5, we show the time taken for Microsearch to satisfy a user's query. As discussed in the theoretical model, we see that a larger inverted index processes queries faster than a smaller inverted index. The predicted query response time is also very close to the measured time. Overall, Microsearch is able to satisfy a user's query in less than two seconds, which we believe is a reasonable time. Fig 6 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.

6.3 Search Accuracy

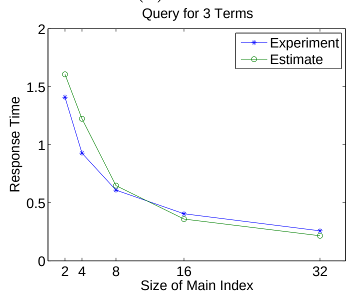
Shah and Croft [25] suggested using metrics from question answering (QA) research [27] to evaluate search algorithms for bandwidth or power constrained devices. QA is a branch of information retrieval that returns answers instead of relevant documents in response to a query. In QA research, the goal 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 (MRR).



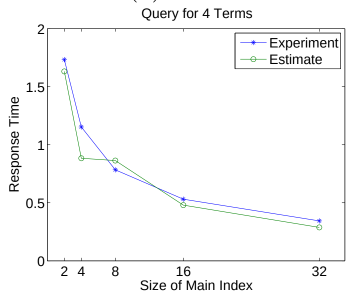
(a) 1 term



(b) 2 terms



(c) 3 terms



(d) 4 terms

Fig. 5. Predicted and actual query response time. Response time measured in seconds

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, consider the model answer to a query be the ranked list (A, B, C) and the 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 [5]. 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.

Our evaluation does not consider deliberately vague queries since it is difficult to objectively quantify what the answer *should* be. Instead, we generated queries which contain terms that are found in multiple documents, but these queries have

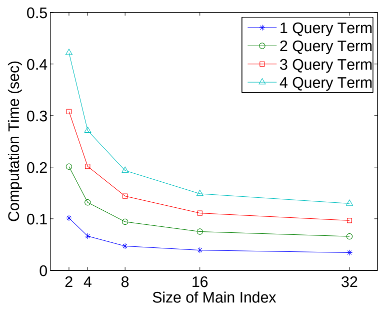


Fig. 6. Processing time overhead of search system processing

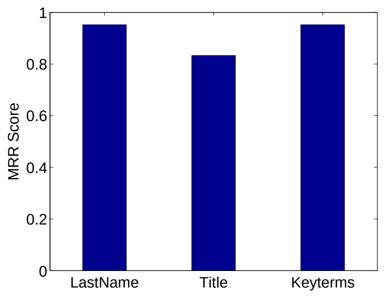


Fig. 7. Query accuracy (k=3)

a clear answer. An example of a query is "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. However, two other papers contain more occurrences of the term "storage". A good search system will be able to rank the most likely result ahead of a less likely one.

Fig. 7 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 $k = 3$.

6.4 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 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 $1KB$ 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 $76B$, and a buffer cache of $944B$. The alternative system takes an average of 6.5 ms to index the metadata in one file compared to the 20 ms for our scheme. Fig. 8 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

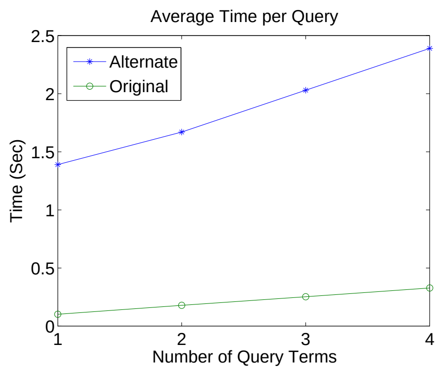


Fig. 8. Comparing alternative scheme with our scheme

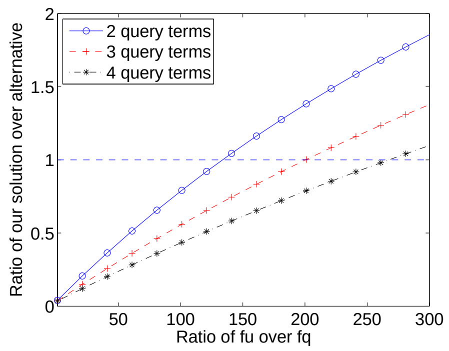


Fig. 9. Comparing power consumption of our scheme verses alternative scheme

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 be expressed as $E_1 = f_u \cdot W_i \cdot (P_w + P_r) + f_q \cdot R_q \cdot P_r$, where W_i is the number of pages written for the metadata in the worst case (when m terms are mapped to m different index entries), and R_q is the number of page read operations required for the query. From Section 5, we have $W_i = \frac{m}{\frac{B}{H}+1}$ and $R_q = \frac{2 \cdot D \cdot m \cdot t}{E \cdot H}$.

Similarly, the energy consumed by the alternative scheme can be expressed as $E_2 = f_u \cdot W_i' \cdot (P_w + P_r) + f_q \cdot R_q' \cdot P_r$, where $W_i' = \frac{m}{E}$, and $R_q' = \frac{2 \cdot D \cdot m \cdot t}{E}$. With the system parameters fixed at $D = 622, m = 3, E = 31$ and $H = 32$, we estimate the energy consumption for both schemes based on TelosB flash memory read and write energy performance presented in [21] (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 2, 3 and 4 query terms respectively. The estimation results are found in Fig. 9. The figure shows that for an average of 2 query terms, the alternative performs better when there are about 140 document insertions to a single query. For an average of 3 and 4 query terms, the alternative scheme performs better only when there are 200 and 260 document insertions to a single query. This suggests that the alternative scheme should be used only when the mote is used to store data and rarely if ever queried.

7 Conclusion and Future Work

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.

In future work we aim to implement security measures such as access control and data encryption into Microsearch and evaluate their performance. We also plan to incorporate Microsearch into our physical world search engine, Snoogle [28], for further evaluation.

Acknowledgments

We would like to thank the reviewers and our shepherd, Nigel Davies, for all their helpful comments in improving this paper.This project was supported by US National Science Foundation award CCF-0514985, CNS-0721443, and CNS-0747108.

References

1. G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: a mobile context-aware tour guide. *Wirel. Netw.*, 1997.
2. Apple. http://www.apple.com/macosx/features/spotlight/.
3. R. Baeza-Yates, G. Dupret, and H. Velasco. A study of mobile search queries in japan. In *WWW '06*.
4. Beagle. http://beagle-project.org/*main_page*.
5. J. Chen, A. Diekema, M. D. Taffet, N. J. McCracken, N. E. Ozgencil, O. Yilmazel, and E. D. Liddy. Question answering: CNLP at the TREC-10 question answering track. In *Text REtrieval Conference*, 2001.
6. K. Cheverst, N. Davies, K. Mitchell, and A. Friday. Experiences of developing and deploying a context-aware tourist guide: the guide project. In *MobiCom 2000*.
7. K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Developing a context-aware electronic tourist guide: some issues and experiences. In *CHI 2000*.
8. K. Church, B. Smyth, P. Cotter, and K. Bradley. Mobile information access: A study of emerging search behavior on the mobile internet. *ACM Trans. Web*, 2007.
9. H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys 2004*.
10. C. Faloutsos. Access methods for text. *ACM Comput. Surv.*, 17(1), 1985.
11. C. Faloutsos and D. W. Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, 1995.

12. W. B. Frakes and R. A. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
13. J. C. French, A. L. Powell, J. P. Callan, C. L. Viles, T. Emmitt, K. J. Prey, and Y. Mou. Comparing the performance of database selection algorithms. In *Research and Development in Information Retrieval*, 1999.
14. E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2), 2005.
15. Google. www.desktop.google.com.
16. Intel. www.intel.com/research/downloads/imote-ds-101.pdf.
17. M. Kamvar and S. Baluja. A large scale study of wireless search behavior: Google mobile search. In *CHI '06*.
18. M. Kobayashi and K. Takeda. Information retrieval on the web. *ACM Computing Surveys 2000*.
19. Logitec. www.logitech.com.
20. G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *SenSys 2006*.
21. G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low power data storage for sensor networks. In *IPSN 2006*.
22. J. Paradise and E. D. Mynatt. Audio note system.
23. P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. PicoDBMS: Scaling down database techniques for the smartcard. *VLDB 2001*.
24. J. Rekimoto, Y. Ayatsuka, and K. Hayashi. Augment-able reality: Situated communication through physical and digital spaces. In *ISWC 1998*.
25. C. Shah and W. B. Croft. Evaluating high accuracy retrieval techniques. In *SIGIR 2004*.
26. T. Starner, D. Kirsch, and S. Assefa. The locust swarm: An environmentally-powered, networkless location and messaging system. In *ISWC 1997*.
27. E. M. Voorhees. Overview of the TREC 2001 question answering track. In *Text REtrieval Conference*, 2001.
28. H. Wang, C. C. Tan, and Q. Li. Snoogle: A search engine for the physical world. In *Infocom 2008*.
29. D. Woodhouse. Jffs : The journalling flash file system.
30. Wookey. Yaffs: Yet another flash file system.
31. K.-K. Yap, V. Srinivasan, and M. Motani. Max: human-centric search of the physical world. In *SenSys 2005*.
32. D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST 2005*.