Exploiting the Interplay between Memory and Flash Storage in Embedded Sensor Devices

Devesh Agrawal, Boduo Li, Zhao Cao, Deepak Ganesan, Yanlei Diao, Prashant Shenoy
University of Massachusetts, Amherst
{dagrawal,boduo,caozhao,dganesan,yanlei,shenoy}@cs.umass.edu

***Abstract*—Although memory is an important constraint in embedded sensor nodes, existing embedded applications and systems are typically designed to work under the memory constraints of a single platform and do not consider the interplay between memory and flash storage. In this paper, we present the design of a memory-adaptive flash-based embedded sensor system that allows an application to exploit the presence of flash and adapt to different amounts of RAM on the embedded device. We describe how such a system can be exploited by data-centric sensor applications. Our design involves several novel features: flash and memory-efficient storage and indexing, techniques for efficient storage reclamation, and intelligent buffer management to maximize write coalescing. Our results show that our system is highly energy-efficient under different workloads, and can be configured for embedded sensor platforms with memory constraints ranging from a few kilobytes to hundreds of kilobytes.**

I. Introduction

Embedded networked systems have become popular in recent years for applications such as habitat monitoring, environmental monitoring and surveillance, with deployments comprising battery-powered embedded nodes with wireless networking capabilities. Typically such embedded nodes, exemplified by the Mote and iMote2 platforms, consist of a low-power microprocessor, a low-power radio, a small amount of RAM and some flash storage. While significant research attention has been focused on optimizing embedded sensor applications for computation and communication, there has been less emphasis on the interplay between memory and flash-based storage. Presently an embedded sensor application must be optimized for the memory footprint of the embedded node; since the memory on such nodes can vary from a few kilobytes to a few megabytes, the same application needs to be designed differently to yield a kilobyte- or a megabyte footprint. This increases application design complexity, since it is non-trivial to design a particular application to run on a range of embedded devices with a range of memory constraints. At the same time, since flash memories are relative cheap and energy-efficient, most devices can be easily equipped with relatively large amount of flash storage[7], [14]. This trend opens up new opportunities for optimizing the interplay between memory and flash, in particular for trading abundant flash for scarce memory. Our work focuses on the design of a memory-adaptive flash-based sensor system that allows an application to exploit the presence of flash and adapt to different amounts of RAM on the embedded device.

This research was supported in part by NSF grant CNS-0626873.

Specifically, we present the design of a memory-adaptive, energy-efficient flash-based system that is tailored for data-centric embedded applications. We argue that the design of such a system requires an in-depth understanding of the interaction between memory constraints and flash constraints. As a storage medium, flash has fundamentally different read/write characteristics from other non-volatile media such as magnetic disks. In particular, flash writes are immutable and one-time—once written, a data page must be erased before it can be written again. Moreover, the unit of erase often spans multiple pages, further complicating storage management. Both flash and memory constraints fundamentally impact the energy consumption of a storage system: the idiosyncrasies of flash lead to a high update cost, and the constrained memory exacerbates the number of flash accesses.

In addition to being influenced by memory and flash characteristics, a storage system should also be informed by the data needs of embedded sensor applications. Unlike a general purpose system, a sensor-specific data storage system needs to be optimized for the particular characteristics and needs of sensor data, namely that archived data is generated as a *continuous stream* of observations, that may need to be *indexed* for efficiently answering queries. While data storage on flash is straightforward, design of indexes has to deal with the lack of in-place updates on flash, often necessitating new techniques. Further, the finite capacity of flash storage and the need to deploy embedded networked systems in remote areas for long periods without human intervention implies storage may need to be reclaimed, especially for high-data rate sensors.

The goal of our work is to design a flash-based storage layer that is optimized for the flash and memory constraints imposed by sensor devices, while efficiently meeting the archiving, indexing, and aging needs of sensor data. We show that flash and memory constraints interact in many intricate ways, and necessitate new design paradigms for a storage system.

A. Relation to Previous Work

While many recent efforts have targeted flash memory storage management, both for sensor networks [6], [14], [20] and for other embedded systems [2], our work is fundamentally different in two ways. First, much of the existing work on flash storage systems has been tailored to work under the memory constraints of a specific target sensor platform. For example, FlashDB is designed for Windows mobile [16], whereas Capsule, MicroHash, and Elf are all designed for

Motes [6], [14], [20]. This makes these systems less portable to platforms that have less memory, and not as efficient on platforms that have more memory. In contrast, our work seeks to design a memory-adaptive storage system that works within the memory constraints of a device, and can also be easily adapted to use more (or less) memory. Second, existing work has only peripherally addressed the problem of deletion and garbage collection of data from flash. We tackle this problem in detail in this paper, and provide an in-depth understanding of how deletion impacts the cost of maintaining data on flash as well as the mechanisms to minimize this cost.

B. Research Contributions

In this paper, we present a novel flash storage and memory management system for data-centric embedded applications. Our work has four major contributions.

- **Storage and Index Layer Primitives and Optimizations:** Our system provides: (a) storage-layer primitives that are designed to mask the idiosyncrasies of flash, but at the same time enable efficient index construction, and (b) index-layer optimizations that enable design of flash and memory-efficient index structures.
- **Memory-Adaptive Data Structures:** In our embedded storage system, all the data structures used for bookkeeping are: (a) flash-based to minimize memory footprint, (b) energy-optimized by minimizing reads, writes and erases on flash, and (c) memory-aware to be able to adapt to sensor platforms with diverse memory constraints.
- **Efficient Storage Reclamation:** Our storage system is designed to support diverse reclamation strategies for indexes and data, while minimizing the energy overhead of aging by eliminating expensive movement of data.
- **Energy-Optimized Buffer Manager:** Since limited memory on sensor nodes needs to be shared by numerous flash-based components, we present an energy optimized buffer manager that allocates memory buffers across different flash-based storage components while considering the interactions between them.

We have implemented our storage and memory management system on the iMote2 platform running Linux and have conducted a detailed evaluation using different sensor workloads and memory constraints. Our results show that our system can scale across a wide spectrum of memory constraints from a few KB to 100s of KB, while optimizing energy consumption given these constraints. We also show that our buffer allocation strategy performs better than other allocation strategies like equal allocation and application rate-based allocation by upto 50% and 33.33% respectively, and adapts to different sensor workloads. We provide a number of optimizations that enable the construction of indexes tailored to flash and memory constraints, and evaluate them for B-trees, interval trees, and inverted indexes. We also provide the ability to tradeoff construction cost for querying cost, thereby adapting to the application workload. Finally, we provide two case studies and demonstrate that our system can be used to design an efficient image search engine, and an efficient temperature monitoring system.

II. Interaction of Memory and Flash Constraints

Why does memory matter for the design of a flash-based storage system for sensor platforms? To answer this question, we start with a brief background on flash constraints. We then take a simple example of a tree-based structure on flash, and discuss how different design tradeoffs are impacted by memory constraints. Finally, we articulate several design principles that result from these constraints.

A. Background

We focus on network embedded devices equipped with raw NAND flash memories in this paper since they are the most energy-efficient flash media [8]. While other types of flashes or flash devices are available (e.g. NOR flashes, SD cards), we leave the discussion on how to handle these types to our technical report [1] due to space constraints.

Although NAND flash is an energy-efficient non-volatile storage medium, it is fundamentally different from other devices such as disks due to its *no-overwrite* nature—once a data page is written to flash, it can not be updated or rewritten and must be erased before being written again. Thus, a *read-modify-write* cycle on traditional storage devices becomes a *read-modify-erase-write* operation on flash. The smallest unit that can be erased on flash, termed an *erase-block*, typically spans a few tens of pages. The mismatch between flash write and erase granularities makes a *read-modify-write-operation* prohibitively expensive since it requires copying all valid pages within the erase block, then erasing the block, and finally copying the valid pages and updated page back to the block. Therefore, it is preferable to write the updated data page to a *different* location, rather than erasing and rewriting the old block; thus updates result in a *read-modify-write elsewhere* operation, requiring a different set of storage optimizations from traditional devices that support *in-place updates*.

B. Index Case study

The no-overwrite nature of flash is well-suited for archiving streaming sensor data, since such data is written to flash in an append-only fashion (and is typically not updated after being archived). However, it is particularly ill-suited for maintaining mutable data structures such as indexes (trees, lists, tables, etc). This is because each index must be constantly rewritten to flash when it is updated.

Consider, for instance, a canonical multi-level tree-based index. Such a basic index can be used to build an array, B-tree, and other structures, and is fundamental to our storage system. Each internal tree node maintains pointers to all its children nodes (each child node is a data page on flash), and the leaf nodes of the tree contain keys. Since embedded sensor devices are memory-constrained, only a subset of the tree nodes can be maintained in memory at any given time. Memory constraints impact the cost of such a tree-based index in many ways:

Cascading Updates and FTLs: Consider the case when a leaf node in the tree is updated. A leaf node update involves reading the node from flash, updating it, and writing it to a new location (to avoid a costly erase operation). Since the physical location of the leaf node has changed, the corresponding pointer in its parent node must be updated to point to the new location. This triggers an update to the parent node, requiring it to also be written to a new location, and so on, all the way to the root node of the tree. Thus, a single update can potentially trigger up to k writes in a k-level tree.

In many existing flash systems, such "cascading updates" are avoided by employing a *flash translation layer (FTL)* that hides the complexity of such updates by exporting logical page numbers and internally changing the logical to physical mapping upon each update to a logical page. In this case, pointers to child nodes of the tree point to logical page numbers that do not change even though the physical page number changes after an update. However, the FTL is a very large data structure since it has a logical-physical mapping for each page (e.g. ranging from 128KB to 2MB depending on the size of flash and the allocation granularity [12]). This overhead exceeds the total memory on most sensor devices. Of course, one could maintain an FTL on flash, but since the FTL is itself a logical-to-physical index that is also maintained on flash, update of the FTL on flash incurs additional cost.

Thus, the first challenge is how to obtain the benefits of FTLs for storing mutable data on flash while not incurring the overhead of maintaining a large FTL on flash.

Caching: Given the "no-overwrite" constraint imposed by flash, judicious use of memory can vastly improve the energy-efficiency of a storage system, especially when there are multiple mutable flash-based data structures. In particular, the greater the amount of memory used to cache dirty pages of such a structure, the greater are the opportunities for write coalescing, *i.e.* updating the memory buffer rather than triggering out-of-place updates on flash. In practice, we have observed that, for some flash-based data structures, even a small increase in the amount of memory given to a flash-based index can have a large benefit in energy consumption (for example, about $2\times$ reduction when memory is increased from 1.5 KB to 2.5 KB). While caching is important for all mutable structures that use flash, different components in the system can benefit to different extents by using more caching. Thus, on low-memory embedded platforms, it is important to allocate memory to components that benefit most from it.

Thus, a second challenge is allocating memory for caching across various components to minimize overall energy use.

Storage Reclamation: When flash space runs low, the system needs to "garbage-collect" itself, *i.e.* to purge out the old invalidated copies of its nodes on the flash similar to segment cleaning in log-structured filesystems [18]. However, storage reclamation can be expensive since in each erase block, a few pages might be invalidated whereas others contain valid data. To free an erase block worth of data, valid data from the erase block needs to be copied out to a different location, before erasing the block, thereby incurring significant overhead.

Thus, a third challenge is optimizing the flash-based storage substrate to limit the overhead of out-of-place updates and consequent storage reclamation.

C. Design Principles

The flash and memory constraints articulated above yield the following design principles for our storage system:

Principle 1: *Support multiple storage allocation units and align them to erase block boundaries whenever possible.*
To minimize reclamation costs, the storage layer enables applications to specify the appropriate granularity of allocation/deallocation units, thereby providing a balance between energy-efficiency and application needs. For example, the sensor data can have a large allocation unit that is aligned to erase block boundaries to minimize reclamation cost, whereas indexes can use a smaller unit aligned to the node size.

Principle 2: *Implement smaller localized FTLs only when mutable data is stored.*
Although a flash translation layer provides a key benefit by hiding the changes to a physical page location every time a page is updated, maintaining an FTL for the entire flash imposes high overhead for memory constrained devices. To preserve the benefits of an FTL without incurring its high overheads, we maintain smaller FTLs for regions of flash where mutable data is stored (sensor data is immutable whereas indexes are mutable). Such "localized" FTLs are inexpensive to maintain on flash, or may fit completely in memory.

Principle 3: *Design all data structures in the system to be tunable memory-wise and apportion memory across data structures to minimize overall energy consumption.*
Scalability across memory constraints (1KB - 1MB) requires that different data structures used by the storage layer be tunable to memory-size by configuring the number of buffers that they use. In addition, there needs to be judicious apportioning of limited memory resources across different data structures that use flash memory in order to extract maximum savings via write coalescing—a particularly important issue for no-overwrite devices such as flash.

Principle 4: *Provide storage-layer primitives and index-layer optimizations to design flash and memory-optimized index structures.*
The design of indexes tailored to flash and memory constraints is a complex task. To facilitate such design, we provide storage-layer primitives that transparently optimize indexes that have not been designed for such constrained systems. We also provide specific index-layer techniques that can enable indexes to be tailored for flash and memory constraints.

III. System Architecture

Our storage system has three key components: the Storage Layer, the Index Layer, and the Buffer Manager (as depicted in Fig. 1) that provide the following services: (i) *storage allocation* across application needs, (ii) *index management* to facilitate indexing of data on flash, (iii) *storage reclamation* to handle deletions and reclamation of storage space, and (iv) *buffer management* to allocate memory buffers across all

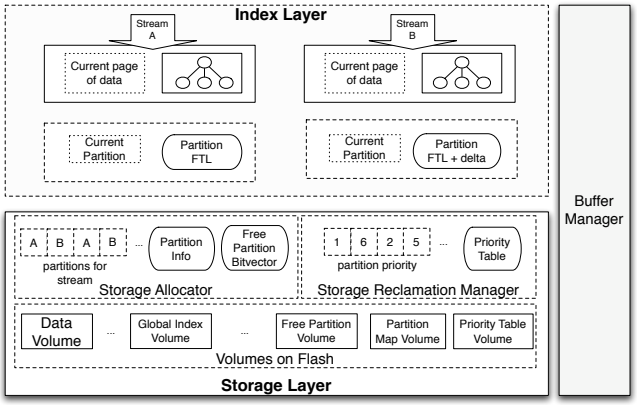


Fig. 1. System architecture.

active streams and key system-maintained data structures. Our system maintains several metadata structures on flash, which are summarized in Table I.

A. Storage Allocation

The storage allocator allocates storage space, upon request, to each active sensor stream. At the highest level, flash is divided into fixed size volumes, each of which can be allocated to different active streams. Volumes can be of two types: *user* volumes and *system* volumes. A user volume is used to store one or more sensor streams and/or indexes. A system volume is used to store internal system data structures, and is not directly accessible by applications.

Within a volume, storage is allocated at the granularity of a *partition*. While partitions can be of any size, the choice has a significant impact on overall performance when reclamation is needed. A small allocation unit that is significantly smaller than an erase block (e.g. a flash page or sub-page granularity) greatly increases reclamation costs. In each erase block, a few pages might be invalidated whereas others contain valid data. This will incur a high cost of copying valid data when an erase block worth of data needs to be freed, as explained above. In contrast, if the application is designed to use an erase block as the unit of allocation, storage reclamation is inexpensive since the invalidated erase blocks can be erased to make space. However, a large partition is inconvenient for index structures such as a B-tree, which use nodes that are much smaller than an erase block. In this case, a page or sub-page partition may be more convenient. Note that although storage is allocated and reclaimed at the granularity of a partition, it is different from the read/write granularity, which can be done at the granularity of an individual page or sub-page (if the flash allows multiple writes to a page). This helps reduce memory overheads (smaller I/Os imply smaller memory buffers).

A *free partition list* is maintained on flash for each volume, to track free partitions. In addition, storage reclamation is invoked to free space if no free space is available in the volume. All storage allocation requests are handled through an API exposed to upper layers. Each allocation request must indicate whether the partition/volume is mutable or not; if mutable, an FTL is created for the appropriate allocation unit.

Module	Structure	Description	Construction
Storage Allocation	Free partition list	track free partitions in volume	array on flash (impl. as an n-ary tree)
Index Management	FTL (agnostic)	FTL for requested volume/partition	in-memory table (small FTL) or array on flash (large FTL)
	FTL (delta-based)	FTL with deltas and consolidation	in-memory table if small or array on flash if large
Storage Reclamation	Invalidation table	stores whether partition is valid or not	array on flash
	Priority table	stores the priority level of partition	bitmap index
	Temporal log	stores time of partition creation	simple log on flash
	Partition map	stores owner of partition for callback	array on flash

TABLE I
SYSTEM METADATA STRUCTURES

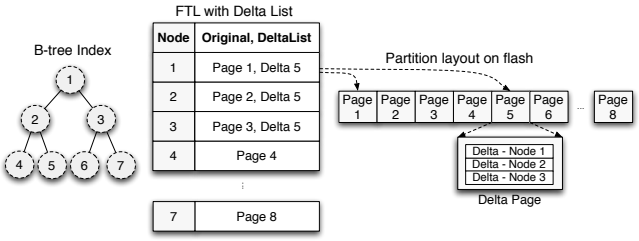


Fig. 2. Flash optimized index illustration.

In summary, aligning partitions to erase block boundaries should be adopted when possible, which is verified by our experiments. We will show how to efficiently manage indices in the Section III-B, where page-sized partitioning is desired.

B. Index Management

The index management module provides a variety of primitives, both at the storage and indexing layers, to facilitate the construction of indexes under flash and memory constraints.

1) Storage-Layer Optimizations: We support two approaches to allow a generic index structure to deal with the lack of in-place updates on flash.

Flash-agnostic structures: The first primitive enables indexes to update pages on flash in a manner oblivious to the no-overwrite constraint of flash. This is achieved through the use of a flash translation layer (implemented in software) which exports logical page numbers to upper layers and maintains a mapping of logical page numbers to their actual physical locations. While FTLs are commonly used in flash storage systems, a unique aspect in our system is that it employs FTLs only for partitions/volumes where it is requested.

Flash-optimized structures: FTLs simplify the design of upper layers by hiding flash idiosyncrasies, but they incur high energy cost since each update, regardless of its size, triggers an out-of-place page rewrite. This is especially wasteful when only a small fraction of page is modified (e.g., insert of a single key in a B-tree node), causing the entire page to be rewritten. To optimize for small updates, we support the notion of a *delta*. A delta is a self-describing structure that compactly represents an incremental update to a page. For instance, while

inserting keys to a B-tree node, the index layer can specify the key and inserted position as a delta. Fig. 2 illustrates a B-Tree implemented using a FTL with a delta list.

If a node has many updates, the list of deltas can be very long, resulting in significant overhead for reads. To balance the read and write costs, we use a consolidation scheme (similar in spirit to FlashDB [16]). The key idea is to keep growing the "delta chain" until the overhead incurred for reads exceeds the savings obtained from reducing the number of writes. At this point, the delta chain is merged with the original node, and a consolidated node is written to flash.

A flash-optimized index is usually more efficient than a flash-agnostic index except for query-heavy workloads.

2) Index-Layer Optimizations: Despite the storage layer optimizations, the cost of building an index on flash is often too expensive on low-memory platforms due to a large number of out-of-place updates. Next, we describe how indexes can often be re-designed to minimize out-of-place updates so that they are more efficient under memory and flash constraints. We describe our approach in the context of two data types — temperature and images. In the case of temperature trace, we assume that each reading is being indexed, and in the case of an image, features within each image are indexed.

For the temperature data type, we construct an interval B-tree, where the index entries in the leaves of a tree are modified from a single value to a *value interval*. Fig. 3(a) illustrates such a structure. The top layer is a tree search structure over the intervals. Intervals are stored in the leaf pages of the tree, as shown in layer (2). Each interval points to a list of pages that contain readings (more precisely, addresses of those readings on flash) belonging to this interval, as shown in layer (3). In our system, the list for each interval is implemented as a reverse linked list of pages, i.e., the most recent page is at the head of the list. The interval bins can be determined using a dynamic construction method similar to Microhash [20] or a linear regression method to find the connection between the width of intervals and the memory size.

This structure has two key benefits. First, the interval bins can be adapted such that the tree search structure can be maintained in memory, while only the tail of each list needs to be stored on flash. Since the tail is append-only, it does not result in out-of-place updates, unlike a generic B-tree. Second, interval bins are particularly effective when data is temporally correlated (e.g. weather readings), since consecutive readings often fall into the same interval, leading to good caching behavior in index construction.

The technique that we outlined above can be used for other index structures as well. We use a similar approach to design an inverted index that is used for image search. Image applications usually extract a subset of distinct features (identified by their feature IDs) from each image, index the features in each image, and later retrieve those images with a particular feature. An inverted index is a common index for image features, where each unique feature has a list of image IDs. Constructing a generic inverted index on flash is very costly, because there are many thousands of distinct features

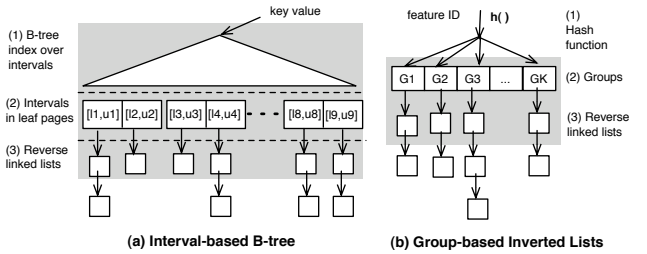


Fig. 3. Optimizing Index structures for flash.

in the index, resulting in numerous out-of-place updates of their Image ID lists. Our system adapts an inverted index by hashing each feature to a group and building an inverted list only for each group, as shown in Fig. 3(b).

3) Granularity of Index Construction: A key question in index management is: *what should be the granularity of index construction?* In particular, should a single global index be constructed over all the sensor data, or are there benefits of constructing smaller indexes over a contiguous subsets of the stream (e.g. over a partition or over a sequence of K values)? We discuss the tradeoffs in the context of the above-described interval tree over temperature data. Building an index for each partition may allow cheap index construction. Since a per-partition index deals with a smaller data set and often a smaller range of distinct values to index, it is possible to have a compact representation of the per-partition index so that it (almost) entirely fits in memory. However, the downside of per-partition indexing is that querying cost can be high since a query may have to scan across multiple per-partition indexes, instead of just looking up a single global index. Therefore, such an approach is more appropriate for applications where the query load is low or where queries only need to retrieve data from a few partitions. Details of these techniques and how the per-partition indexes compare with global indexes are described more in the case studies in Section V.

C. Storage Reclamation

Our system deletes partitions to reclaim storage space whenever the free space in the system falls below a pre-defined threshold. We support multiple mechanisms to reclaim partitions. The "default" approach is one where upper layers invalidate partitions that they no longer use, and the storage system reclaims these partitions when storage space is low. For example, when a B-tree is maintained on flash, this approach invalidates old copies of rewritten nodes. In addition to this, we provide two specialized mechanisms that are tailored to typical sensor data deletion needs:

Priority-based Aging: Here, each partition is assigned a "priority" that captures how important the data in the partition is to the user. (For example, a partition with event data might be considered more valuable than one without event data.) We assume that there are a small number of priority levels (e.g. 8 levels). For each of these priority levels, we maintain a bitmap index [17], where each partition is represented by a single bit that represents whether the partition is of the specified priority. For reclamation, the partitions of lowest priority are found by

scanning the bitvectors in the descending order of priorities. The bitmap index is highly packed, and hence requires very few operations to access.

Time-based Aging: Our system also supports temporal aging of sensor data. The age of the partition is its creation time, which causes the oldest partitions to be reclaimed first. This is implemented by a simple log of partition numbers on flash. Each time a partition is closed (or timestamped), we add its partition number to the tail of this log. During reclamation, we delete partitions from the head of the log.

When the automatic aging methods decide to delete a partition, the owner of the partition (e.g. a data stream) is notified of its impending deletion via an up-call; such notifications allow the owner to perform any book-keeping operations as needed. To efficiently determine the owner of each partition that is being deleted, we maintain a *partition map* on flash; the map contains metadata for each partition, including its owner. Upon deletion, the partition is inserted into the free partition list for subsequent allocation. The partition version number, also maintained in the *partition map*, is incremented to protect higher layer structures from stale references to a partition after it has been deleted.

All data structures in our system, with the exception of the temporal log, are implemented as *multi-level N-ary trees* (which allows for efficient implementation of an array or table on flash). The leaf pages constitute an array of entries and the non-leaf pages serve to guide the path to the appropriate leaf page containing the entry of interest. Each data structure is assigned a volume on flash (in the system area) and is designed to garbage collect itself. Each update to a tree page causes it to be written to a new location on flash and old invalid pages are periodically garbage collected by moving valid pages within the reclaimed erase blocks to new locations.

D. Buffer Management

We finally consider allocating the total system memory among different system components to minimize the total energy cost. The buffer manager provides a buffer cache to every component in our system that uses the flash[1]. The buffer cache lowers flash I/O cost for each component by increasing cache hits for flash read requests and yields write coalescing benefits, thereby reducing rewrite costs. The allocation of buffers across different components is based on the results of an a priori analysis using training data, as detailed below.

Since typical sensor platforms do not efficiently support dynamic memory allocation [13], our current design focusses on a priori allocation of memory among different components. Assume that we have M buffers of available memory, where M is governed by the hardware platform and the memory already used up by application and OS code pages. The buffer allocation problem can be formally stated as: How can we apportion M among each active stream and system structures to minimize the total energy costs of storage operations. In order to solve this, we proceed in two steps. We first determine the energy cost/benefit of allocating a certain amount of memory to each component. Then, we partition the total memory based on these individual costs so as to minimize overall system-wide energy consumption. Based on this analysis, we determine how many buffers to allocate to each component.

Energy-Memory Profiles: For each active stream or system structure, the first step is to quantify the total I/O energy costs incurred for different memory allocations. This relationship between different memory allocations and the corresponding I/O cost for each allocation is referred to as the *energy-memory* profile of that stream or data structure. We assume that such energy-memory profiles are derived offline for each data structure/stream and specified *a priori* to the buffer manager. The profile may be derived by offline profiling, where a trace of I/O requests is gathered, one for each possible memory allocation while keeping the workload and other parameters fixed. These empirical traces then yield the total I/O costs for different memory allocations.

Buffer allocation: Once the energy-memory profiles are obtained, the buffer allocation can be done as follows. The energy consumed by the storage system is the sum of costs of updating storage system metadata structures during partition allocation, as well as erase and data movement operations during storage reclamation. Note that storage reclamation is not required the first time that the flash is being filled. The energy consumed by the data management module has two parts: (a) writing data as well as indexes to flash, and (b) reading index pages from flash during both index construction (reads of prematurely evicted pages, if any) and querying. The cumulative energy cost includes the storage cost and the data management layer cost. Given the energy-memory profiles of each data structure, the overall buffer allocation problem simply involves minimizing this cost function, given the constraints of the minimum amount of memory that each data structure needs. This can be easily computed using a standard optimization package.

Inter-component interactions: The above buffer allocation may not be optimal since it assumes independence and ignores interactions between components. For example, if an application index structure is allocated less number of buffers, it may write more often to flash, since it evicts more dirty pages. This can result in reclamation being triggered more often since the volume/partition fills up sooner, which in turn can result in more accesses to system data structures maintained for reclamation, thereby increasing its energy cost. Such interactions often depend on a number of parameters including the type of index and its cache behavior, data/index layout on flash (per-partition vs global index), etc.

We observe two common types of interactions. First, for a per-partition index that is co-located with data, less memory for the index can result in a data+index partition filling up sooner, thereby triggering more storage allocation/reclamation. The second case is when a global index in a separate volume uses a large FTL that needs to be maintained on flash. Here, the number of buffers allocated to the index impacts its flash access pattern, and hence FTL cost. In both cases, we jointly

[1]We use an LRU eviction policy in our current implementation.

construct the energy-memory profiles for the interacting components, and use them to solve buffer allocation. For example, in the co-located partition case, we use the energy-memory profile of the index to determine the rate at which the index writes to flash as a function of memory, and use this trace to determine how it impacts storage allocation/reclamation costs.

IV. Implementation

We have implemented a prototype of our storage system on the iMote2 platform running Linux. Our system is written in C++, and consists of a few thousand lines of code that implement the storage allocator, indexing module, and the buffer manager. Currently, the iMote2 platform only supports SD cards with onboard hardware flash controllers and lacks support for raw NAND flash storage. Consequently we wrote a "pseudo driver" to emulate a raw NAND flash store comprising a 1Gb (128 MB) Toshiba TC58DVG02A1FT00 NAND flash chip. The emulated storage device has a page size 512 bytes, and an erase block size of 32 pages. This flash chip has been accurately measured and profiled in recent work [14], and we use the results from this measurement study for accurate emulation—a page read, write and a block erase consume 57.83 μJ, 73.79 μJ and 65.54 μJ respectively. Each buffer cache had a buffer size of 512 bytes, to match the page-write granularity of the underlying flash.

The index layer has an in-memory FTL and on-flash FTLs (non-delta based and delta-based). We have also implemented three types of flash-based indexes including a B-tree, an inverted index, and an interval tree. (Due to time constraints, the inverted index and the interval B-tree are implemented as a stand-alone emulator, and not integrated with the rest of our system.) For all the above indexes, we have implemented a per-partition as well as a global variant as discussed in Section III-B. In addition to these complex indexes, we also support simple data structures like logs and queues, in a manner similar to that described in [14].

V. Experimental Evaluation

In our evaluation, we used three types of data and index workloads for micro-benchmarking and use case studies: The **Uniform** stream has uniformly distributed keys. The **Temperature** stream contains temperature readings obtained every 15 minutes, since Jan. 1, 2007, from a local weather station. The **Image** stream contains 1985 images from the Oxford buildings dataset [21]. Each image has roughly 1000 features, and there are 1 million unique features across all images. We use this trace in our image storage and indexing case study (§ V-B).

A. Microbenchmarks

Our first set of experiments evaluates individual components of our storage system. Unless otherwise mentioned, we use the B-tree index and the Uniform trace in these microbenchmarks.

1. Impact of Partition Size. *How should partition sizes be chosen to minimize energy consumption?* We consider three partition allocation strategies: sub-page allocation, *i.e.* smaller than a page; equal to a page; and aligned to an erase block. For this experiment we wrote 256 MB of data over a 128 MB sized flash. We used priority based aging, where the priorities were randomly chosen by the application. Fig. 4 shows that for partition sizes less than a block, the cost of accessing the storage layer (SL) metadata increases greatly, primarily because of the larger size of the SL structures. In addition, for partitions smaller than an erase block, there is considerable data movement before an erase to move valid pages from erase blocks that are intended for deletion. In contrast, the block based allocation has zero data moving cost and a very low metadata maintenance overhead.

In summary, aligning partitions to erase block boundaries is highly efficient, and should be adopted when possible.

2. Flash-Agnostic vs Flash-Optimized Indexes. We now evaluate the relative merits of a flash-optimized (delta-based) versus a flash-agnostic B-tree index. For the flash-optimized index, three types of deltas were used for: (a) insertion of a new key, (b) movement of a key from one node to another, and (c) update of a child pointer. In this experiment, we inserted 5000 keys chosen from a uniform distribution into the index, and queried all the keys back. This was done for different choices of memory allocated to the index. We assumed that the buffers used for construction and querying were not shared, hence there were no caching effects to distort the results.

Fig. 5(a) shows that the flash-optimized index has considerably lower construction cost, per value inserted, than the flash-agnostic index (more than 3x improvement for 6 KB memory). This is mainly because the flash optimized B-Tree avoids rewriting nodes for small updates and thereby lowers the construction cost. However, this increases the read cost for some nodes, as a long delta chain might need to be read to reconstruct them. Fig. 5(b) shows that this increases the query cost (per value queried) of the flash-optimized index (by about 10% for 1.5 KB memory).

Since the improvement in construction cost far outweighs the reduced query performance, a flash-optimized index is generally more efficient than a flash-agnostic one, with the exception of query-heavy workloads.

Although read/write cost varies on different flash chips, this conclusion holds as long as in-place update is much more expensive than read.

3. Storage Reclamation. We now evaluate the efficiency of storage reclamation. In particular, we look at the benefits of using a bitmap index for priority-based deletion as opposed to keeping it as an array on flash. The priority based bitvector is a collection of 8 bitvectors, one for each priority (as explained in section III-C). We assume that each reclamation operation involves locating a single partition that has the lowest priority and deleting it. Since the reclamation index is exercised whenever a partition is reclaimed, its energy cost is measured per partition reclamation.

Fig. 6(a) shows the performance of the two implementations versus the amount of memory given to them. An array-based implementation would typically require a full scan of the priority table to locate the lowest priority partition. In contrast, the bitmap index is already sorted on priority and the 8

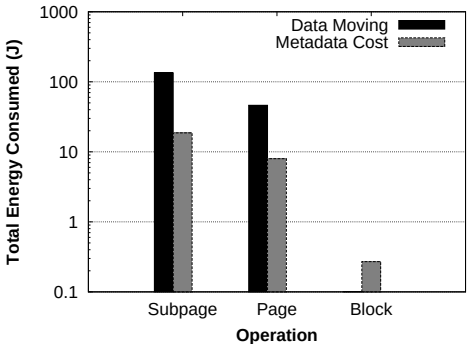
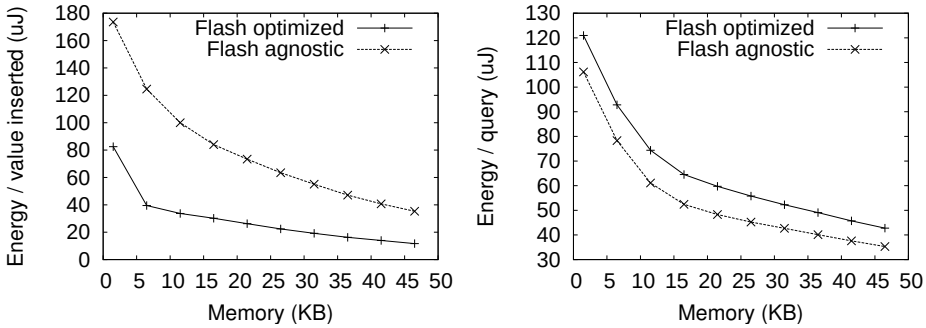
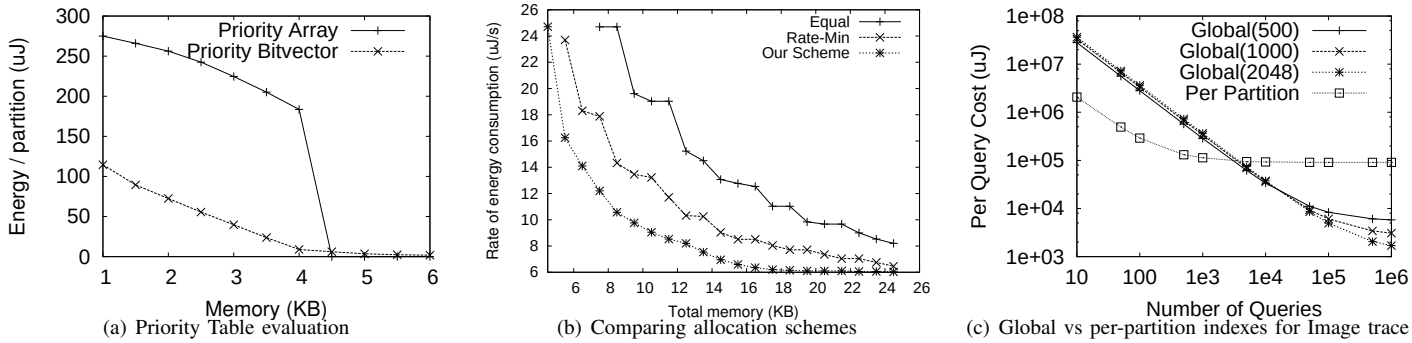


Fig. 4. Impact of partition size.



(a) Index construction. (b) Index query.

Fig. 5. Flash-agnostic vs Flash-optimized indexes.



(a) Priority Table evaluation (b) Comparing allocation schemes (c) Global vs per-partition indexes for Image trace

Fig. 6. Performance of storage reclamation, memory allocation, and the image case study

	Structure	Required Memory (B)
Storage Layer (SL)	Priority Table	184 + 1 buffer
	Free partition list	105 + 1 buffer
	Partition Map	356 + 1 buffer
Index Layer (IL)	Buffer pool (size k)	64 + 4*k + k buffers
	In-mem FTL	6 * #elements + 28
	In-mem FTL+Deltachain	10 * #elements + 24
	On flash FTL	116 + 2 buffers

TABLE II
STORAGE AND INDEX LAYER MEMORY CONSUMPTION BREAKUP IN BYTES

bitvectors are simply walked in descending order to locate the lowest priority partition. The sharp dip in the both the curves is because, beyond 4 KB, the memory allocation in either structure is sufficient to minimize flash accesses.

Thus, we observe that the bitmap index is an efficient storage reclamation data structure.

4. System Memory Overhead. Table II shows the breakup of minimum memory required for different components of our system in the case of a 128MB flash with 1 erase block partition size. The memory for each component includes metadata, and buffers. Buffers are required due to restrictions on the granularity of flash writes. It can be seen that for a buffer size of 512 bytes, our storage layer can easily fit within about 2KB of RAM. Note that this overhead can be easily reduced by reducing the buffer size. For example if a flash allows 2 writes to a page, we only need a buffer size of 256 bytes, thereby reducing the storage-layer memory consumption to just above 1 KB.

Our index layer is equally memory efficient. The size of the in-memory FTLs depends on the number of elements (i.e. the number of nodes in the index). For example, for a per-partition index having 32 pages, the FTL has at most 32 entries (assuming the size of the node equals a page). The flash optimized FTLs only require an additional 4 bytes for each entry. Similarly, our global FTL is implemented as a multi-level tree and only requires two buffers to maintain an entire FTL for 256K nodes (one node per 512B-page).

Although our system requires very limited memory, it can make use of additional available memory by allocating more buffers. *These results demonstrate that our system is memory-optimized and can scale from low memory sensor platforms (few KB) to larger memory platforms (100s of KB)* .

5. Buffer Management Evaluation. Next, we evaluate the performance of the energy-optimized memory manager. We consider the Uniform and Temperature traces in this study, and a per-partition flash-agnostic B-tree index constructed over these traces. A partition size of one erase block (16KB) is assumed, which can store a few thousand data records. The index is assumed to be co-located with data within the partition for easy reclamation. For this experiment, we inserted one million records into the system. Our evaluation takes two steps.

Energy-memory profile: First we study the energy-memory profile of the index layer, using the B-tree as an example. Our results show that the energy consumed by the index for the Temperature data is much lower than that for the Uniform data whey memory is limited (e.g., $\leq 6KB$) due to temporal

correlation in the data and hence better locality for caching. As stated in § III-D, we also capture interactions between the index and storage layers for memory allocation. The details of these results are left to [1] due to lack of space.

Memory allocation: We next evaluate the overall performance of the memory allocation strategy. This experiment has two input streams, Temperature and Uniform. The write rate of the Temperature stream is twice that of the Uniform stream. We compare our strategy with two other approaches: (a) *Equal* allocation is a baseline strategy that is completely blind to the system parameters and allocates equal amounts of memory to each system component, (b) Rate-Min is a rate-based strategy that allocates the minimum amount of memory necessary to the storage layer data structures (1 buffer each for the three structures), and splits the rest of the memory across the two streams in proportion to the rate of the streams (*i.e.* 2:1 with higher memory to the Temperature stream).

Fig. 6(b) shows that Equal allocation does the worst as it is completely agnostic of the system behavior. Since the rate of updates to the storage layer is far lower than that to the index, the Rate-Min policy correctly gives the lion's share of memory to the indexes to reduce their construction costs. However, it performs worse than our scheme since it does not consdier the actual energy-memory profiles of the different structures. In contrast, our scheme is able to reduce the total system energy consumption compared to the Rate-Min and Equal allocation schemes, by up to 33.33% and 50% respectively.

Thus, our memory allocation policy is able to allocate memory in an energy-optimized manner by taking into account the energy-memory profiles of each component and by accurately capturing cross-component interactions.

B. Case Study: Image Storage and Indexing

In this case study, we consider an image retrieval application that needs to store images on flash, and answer queries of the form: "Find all images that match a given target image" [22]. We use the Image stream trace in this case study. We assume that this application needs to be designed using sensor platforms similar to Yale XYZ, which has about 256KB of memory in total, of which we assume that 64KB is available to our system (the rest is used by the OS and image processing software). We consider a 2 GB flash, with a 512 bytes page size and a 16 KB block size, for use by the application.

Configurations of our storage and indexing system: Our system stores image features using the group-based inverted index as described in Section III-B, which maintains a list of image Ids for each group of features. This index is used to support queries that retrieve images containing a specific feature. Images are aged in order of priority, captured by their *tf-idf* scores; that is, images with lower scores are aged earlier than those with higher values. We consider two system configurations for storage, indexing, and reclamation:

Global Index: In this configuration, a single large data volume contains all the images, and a separate index volume contains an inverted index over ***all*** images. The data volume uses a partition size of 4 erase blocks (64KB) to enable reclamation at a single image granularity, yielding about 32K partitions to be managed by the storage layer. Hence each of the SL data structures has at most three levels.

We prioritize memory allocation to the index over the storage layer (SL) as the SL data structures are only activated for each partition allocation, whereas the index is updated thousands of time for each image (as there are thousands of features for each image). Thus we give each SL structure 1 KB of memory. The other system metadata fits in about 2KB of RAM. The global index is given the remaining 59KB memory. As Fig.3(b) shows, has a table that, for each group, maintains a pointer to the last page of its linked list. We restrict the number of groups to be $\leq$2048 so that we can assign 8KB of RAM to keep this table in memory. That leaves 51 KB free to cache at most 102 of the 2048 last pages in memory.

Per-Partition Indexes: In the second configuration, we construct a small inverted index over a sequence of 16 consecutive images (64 KB each). We can compactly represent each list of Fig. 3 by a small 2 byte bitvector, wherein a bit is set if the corresponding image has a feature belonging to the group. The 16 images and the index are packed into a single 1 MB partition to facilitate aging. This yields 2K partitions and thus the SL structures are at most 2 levels deep.

Memory allocation is similar to that for the global index. Each SL structure is given the minimum 0.5 KB memory it needs, and 60 KB RAM is assigned to the per-partition index. We make the per-partition index completely fit in memory to minimize its construction cost. Thus, the per-partition index has 30K groups (each requiring a 2 byte bitvector).

Fig. 6(c) compares these two approaches. The workload in this experiment consists of inserting all 1985 images for index construction followed by Q queries. When using bitvector-based per-partition indexes, the construction cost includes writing both the indexes and the features in each image to flash. This figure shows *per query cost* on a logarithmic scale, which is the total cost of construction and querying amortized by Q (hence decreasing with Q). For the global index, varying the number of feature groups gave only small differences as the total number of features is large (around 1 million).

The most important trend is that as Q increases, the per-partition index outperforms the global index until Q=5000 and performs worse afterwards. This is because the construction cost of per-partition indexes is low (a total of $3J$), whereas the construction cost of a global index is very high (e.g., 281 to $363J$) due to excessive page swapping by the buffer manager. In fact, the construction cost of a global index is higher than executing 1 million queries. On the other hand, the query cost using per-partition indexes is more than 60 times the cost using a global index due to accesses to all partitions. Combining both factors, the benefit of per-partition indexes outweighs the shortcomings when Q is small, and otherwise when Q is large.

In summary, we use an image case study to explore different storage and indexing layer configurations of our system, and make appropriate tradeoffs based on the workload.

Case Study: A Temperature Data Store. In our second

case study, we consider storage and a B-tree index of the Temperature stream. This study is targeted at a low-end TelosB mote platform with 10KB RAM and 1GB flash. Results show that *(a) our interval B-tree, a flash and memory optimized B-tree, works well for temperature data given 10KB RAM, and (b) per-partition indexes offer better efficiency when queries concern data in a temporal range*. See [1] for more details.

VI. Related Work

Sensor Storage: A few recent efforts such as ELF [6], Capsule [14], Matchbox [11], and Microhash [20] have targeted storage management for sensor devices on mote-class sensor platforms. Matchbox is a simple file system that allows only file appends and reads; ELF uses a chain of deltas to avoid in-place updates on flash; Capsule offers a library of simple data structures over flash to hide flash idiosyncrasies from a developer; and Microhash is a specialized flash-based hash table index structure for indexing sensor streams. None of these systems are memory-adaptive, and all of them are optimized only for the lowest end sensor devices. In addition, all these techniques use a log-structured write pattern [18], hence they would incur huge overhead for value-based aging due to the cost of moving data.

Flash Memory: There has also been significant recent work on FTLs and flash storage systems [10]. Although there have been efforts targeting the development of memory-efficient FTL mechanisms [12], all existing FTL implementations require at least an erase block map to be maintained in memory (either on-chip or off-chip), which can be anywhere between 128KB to 4MB in size. Commercial and open source flash file systems (e.g.: YAFFS [5]) are not suited for our system since: (a) they write to flash in a log structured manner making it inefficient for aging, and (b) they consume considerable memory resources; for instance, YAFFS needs about 512KB of RAM for a 128 MB flash [5].

Databases: Database research relates to many aspects of our system design. The notion of splitting the data stream into multiple components, where each component has an associated index has been proposed in LHAM [15], although it focuses on improving write throughput rather than energy-efficiency. There is also a wealth of research on database buffer allocation and buffer replacement algorithms [9], the most related being work on static memory allocation based on query access paths [3]. While relevant to our work, these efforts are not designed for energy-efficiency. Also related are databases that have been designed for embedded smart cards such as PicoDBMS[2] but their focus is not on energy efficiency.

Memory allocation in Operating systems: While the OS memory management literature is replete with techniques for dynamically profiling an application's memory behavior and using these statistics to optimize memory allocations [4], [19], those techniques are not directly applicable to our context because of two reasons: 1) these techniques assume that the application's reference pattern is independent of its actual memory allocation, which is not the case due to the interactions between modules, and 2) current techniques trap a substantial number of memory references to update their histograms, which could potentially waste many cycles on CPU constrained sensor platforms.

VII. Conclusion

In this paper, we focused on the design and implementation of an adaptive memory management system for embedded sensor nodes running flash-based data-centric applications. Our paper takes the first step towards the long-term goal of a memory-adaptive system that enables data-centric applications to run on a wide range of memory-constrained embedded devices. Our prototype, experiments and case studies demonstrated the feasibility of designing such a memory-adaptive embedded system. Specifically, our iMote2 Linux-based implementation showed that our system can scale across a range of memory sizes while minimizing energy consumption for given memory resources. While our current implementation is Linux-based, and hence cannot directly be used on very low-memory platforms, we believe that our architecture is general and can be instantiated on other operating systems. We are currently porting our system to TinyOS [13].

References

[1] D. Agrawal, et al. Exploiting the Interplay Between Memory and Flash Storage for Sensor Data Management. UMass Technical Report, 2010.

[2] W. Attiaoui and M. Ahmed. Conception of a pico-data base management system for a health smart card. In *IEEE ICTTA*, 2004.

[3] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1):311–336, 1986.

[4] J. Cipar, et al. Transparent contribution of memory. In *USENIX Annual Technical Conference, General Track*, 2006.

[5] A. O. Company. Yaffs: Yet another flash filing system. http://www.aleph1.co.uk/yaffs/.

[6] H. Dai, et al. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys*, 2004.

[7] G. Mathur, et al. Ultra-Low Power Data Storage for Sensor Networks. In *IPSN-SPOTS*, 2006.

[8] Y. Diao, et al. Rethinking data management for storage-centric sensor networks. In *CIDR*, www.crdrdb.org, 2007.

[9] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, 1984.

[10] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[11] D. Gay. Matchbox: A simple filing system for motes. http://www.tinyos.net/tinyos-1.x/doc/matchbox.pdf.

[12] J.-U. Kang, et al. A superblock-based flash translation layer for nand flash memory. In *ACM & IEEE EMSOFT*, 2006.

[13] P. Levis, et al. TinyOS: An operating system for wireless sensor networks. *Ambient Intelligence*, 2004.

[14] G. Mathur, et al. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys*, 2006.

[15] P. Muth, et al. Design, Implementation, and Performance of the LHAM Log-Structured History Data Access Method. In *VLDB*, 1998.

[16] S. Nath and A. Kansal. Flashdb: Dynamic self-tuning database for nand flash. IPSN, 410-419, 2007.

[17] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *ACM SIGMOD*, 1997.

[18] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 1991.

[19] T. Yang, et al. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI*, 2006.

[20] D. Zeinalipour-Yazti, et al. Microhash: An efficient index structure for flash-based sensor devices. In *FAST*, 2005.

[21] Philbin J, et al. Object retrieval with large vocabularies and fast spatial matching. In *ICVPR*, 2007.

[22] T. Yan, et al. Distributed Image Search in Sensor Networks. In *SenSys*, 155-168, 2008.