FSort: External Sorting on Flash-based Sensor Devices

Panayiotis Andreou, Orestis Spanos, Demetrios Zeinalipour-Yazti,
George Samaras, Panos K. Chrysanthis[‡]

Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus
[‡] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
{panic,cs05os1,dzeina,cssamara}@cs.ucy.ac.cy, panos@cs.pitt.edu

ABSTRACT

In long-term deployments of Wireless Sensor Networks, it is often more efficient to store sensor readings locally at each device and transmit those readings to the user only when requested (i.e., in response to a user query). Many of the techniques that collect information from a sensor network require that the data is sorted on some attribute (e.g., range queries, top-k queries, join queries, etc.) Yet, the underlying storage medium of these devices (i.e., Flash media) presents some unique characteristics which renders traditional disk-based sorting algorithms inefficient in this context.

In this paper we devise the *FSort* algorithm, an efficient external sorting algorithm for flash-based sensor devices with a small memory footprint. *FSort* minimizes the expensive write/delete operations of flash memory minimizing in that way the consumption of energy. In particular, *FSort* uses a top-down replacement selection algorithm in order to produce sorted runs on flash media in a log-based manner. Sorted runs are then recursively merged in order to yield the sorted result. Our experimentation with real traces from Intel Research Berkeley show that *FSort* greatly outperforms the traditional External Mergesort Algorithm both in regards to time and energy consumption. We found similar advantages in regards to the wearability constraints of flash media.

1. INTRODUCTION

The improvements in hardware design along with the wide availability of economically viable embedded *sensor devices* make it feasible today to interact and understand the physical world at an extremely high fidelity [23, 21, 13]. Applications of *Wireless Sensor Networks* (WSNs) devices range from environmental monitoring (such as atmosphere and habitant monitoring [23, 20, 3]) to seismic and structural [16] monitoring as well as industry manufacturing [6, 13].

In many WSNs, recorded measurements are continuously transmitted to a base station for storage and analysis. However, in long-term deployments it is often preferred to store measurements locally at each sensor and transmit them to the user only when requested. Such a scheme is particularly favored because communication over the radio in a WSN is far more energy demanding than all other functions, such as storage [26, 2, 1] and processing [13, 22, 24, 25].

Although storing the data locally at each sensor is more efficient than transmitting it continuously to a sink point, the storing process by itself raises some important challenges. For instance when users perform *range queries* by a given attribute, (e.g., *"Find GPS locations where humidity is in the range A to B"*), then that requires that the data is sequentially organized (i.e., sorted) by the humidity attribute or that some access method that organizes the given attribute sequentially (e.g., B^{+}Tree) is available. Otherwise, the query will end up traversing a very large number of data pages incurring a large read cost. Notice that reading data off the flash media in large quantities has a significant cost in its own right. Another example where data needs to be sorted on flash is when a query aims to *JOIN* data under a specific attribute. Such JOIN operations are executed more swiftly when sorted data exist [15].

Sorting in WSDs can be performed in two modes, i) *online*: the data is continuously sorted upon the acquisition of new measurements from the sensors, and ii) *offline*: the data is stored on flash using some database scheme and is periodically sorted upon request. A question that arises is which of the two approaches delivers the best performance on flash-based WSDs. Our experimental study presented in Section 5 suggests that online sorting is two to three orders of magnitudes worse than offline sorting both for NAND and NOR flash memory [14].

The majority of WSDs utilize NAND-based flash memory for storing local measurements; we will further describe this specific type of flash memory in Section 2.1. NAND-based flash memory has some unique characteristics which need to be considered when designing data processing algorithms for it such as sorting. In Section 4 will address these constraints when presenting the FSort algorithm.

- **Write-Constraint A:** The minimum unit of write operation in flash is a page. Page sizes vary between 256B to 4KB.
- **Write-Constraint B:** Pages on flash can only be written in sequential order, i.e., after page p_i has been written, any page $p_j, 1 \leq j < i$ can not be written even if it is empty.
- **Erase-Constraint:** A page p_i cannot be deleted unless the block that contains page p_i has been erased.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DMSN '09, August 24, 2009, Lyon, France
Copyright © 2009 ACM 978-1-60558-777-6/09/08 ...$10.00.

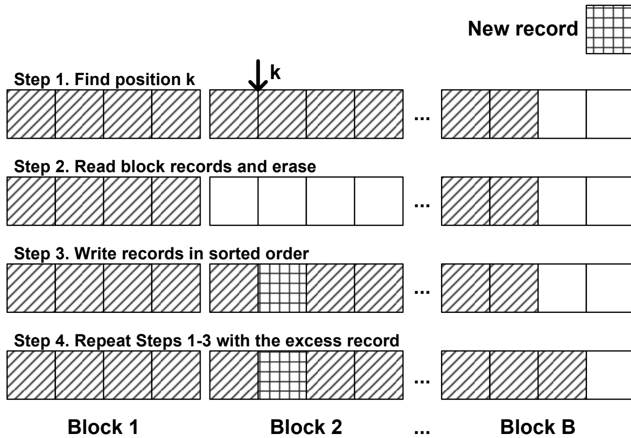

Figure 1: Example of Online Sorting

- **Wear-Constraint:** The number of times a block can be written or erased is limited (typically between 10,000 to 100,000 times).
- **Fast Read/Slow Write-Constraint:** Whereas typically read and write access on a hard disk are almost identical, in flash it is much faster to read ($\approx$60μs) rather than to write ($\approx$800μs).

With the above characteristics/constraints of NAND-based flash memory in mind, we will examine which of the two sorting approaches (online vs. offline) is superior by performing some preliminary analysis. Let us consider the example depicted in Figure 1, which illustrates a flash memory consisting of B blocks each block containing P=4 pages. Also let the flash memory contain N elements distributed throughout the blocks in sorted order (at some time τ). Let us assume that on time $\tau+1$ a new element arrives and needs to be stored in sorted order. The first step is to discover the position k that the new element must be inserted (this requires $log(N)$ reads assuming a binary search). When k is found, the block containing k is first read into memory (P reads) and then erased (due to the Erase-Constraint) such that it can be updated. If the block contains $\geq$1 empty pages then the procedure finishes, otherwise the procedure continues to the next block with the excess page at hand. This technique is extremely inefficient to flash memory as it consists of many write/erase operations that decrease both the performance and lifetime of flash memory [5]. In fact, we will demonstrate this level of detriment using a series of microbenchmarks in Section 5.

On the other hand, offline sorting requires the input to be sorted and written back to flash in a sequential manner only when requested. This is generally preferable in the case of flash memory as it has been shown that continuously executing random writes (as is the case of online sorting) greatly degrades the performance (wearability, throughput) of flash media [5].

Our FSort algorithm extends the basic principles of offline sorting to further enhance its performance by minimizing the write/erase operations. This provides efficient access to the data stored on flash memory while increasing the longevity of the flash memory by spreading page writes out uniformly so that the available storage capacity does not diminish at particular regions of the flash media.

NAND-based flash memory installed on a WSD			
	Page **Read** 1.17mA	Page **Write** 37mA	Block **Erase** 57mA
Time Energy	6.25ms 24μJ	6.25ms 763μJ	2.26ms 425μJ
	Page **Erase-Write** 43mA	Flash **Idle** 0.068mA	Flash **Sleep** 0.031mA
Time Energy	6.75ms 957μJ	N/A 220μJ/sec	N/A 100μJ/sec

Table 1: Performance Parameters for NAND-based flash memory using a 3.3V voltage, 512B Page size and 16KB Block size

Our Contributions

In this paper we make the following contributions:

- We devise a sorting algorithm, coined FSort, which intelligently exploits the characteristics of flash memory to deliver good performance both in terms of time and energy;
- We provide an extensive experimental evaluation using traces from a real sensor network deployment at Intel Research Berkeley [7] that shows the advantages of offline over the online sorting and the superiority of FSort over existing algorithms.

The remainder of the paper is organized as follows: Section 2 overviews the related research work and Section 3 formalizes our system model and assumptions. Section 4 introduces the FSort algorithm and its two phases. In Section 5 we present our experimental study and finally Section 6 concludes our paper.

2. BACKGROUND AND RELATED WORK

2.1 Flash Memory

Flash is a new generation of non-volatile memory that introduces many advantages compared to traditional storage media, including: shock-resistance, fast read access, low production cost and power efficiency. Flash memory is nowadays the de-facto storage medium for WSDs and a variety of other mobile devices.

There are two types of flash memory, namely *NOR* and *NAND* [14] (both names are according to the internal structure used in the respective media). NOR flash provides fast read access, slow write/erase time, low density and a random-access interface. The latter property makes NOR flash ideal for application or boot code execution as it allows direct access any address location. On the other hand, NAND-flash has faster write/erase times and requires a smaller chip area per cell; thus increasing the overall storage capacity; this also lowers the cost of NAND flash. However, NAND flash does not allow random access to any memory location like NOR flash as it operates with a page-size interface.

2.2 Flash-based Databases

In this section we briefly describe some of the most popular approaches to flash-based databases that try to cope with the constraints mentioned in the Introduction.

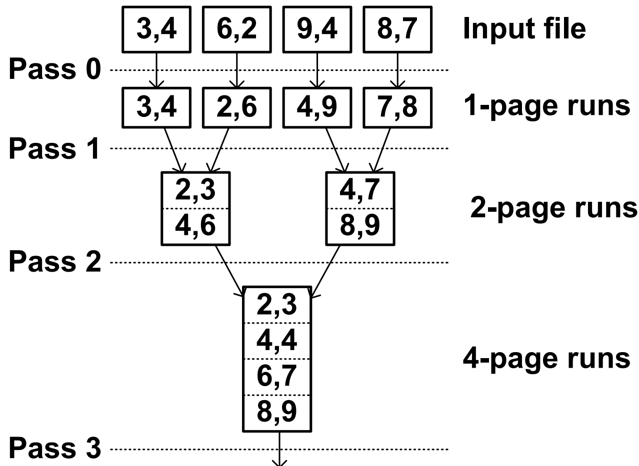

Figure 2: External Sorting example with M=3 (2 input and 1 output page buffers).

- **Flash Translation Layer (FTL):** One way to deploy traditional hard disk-based database software on flash memory is by utilizing a *Flash Translation Layer* (FTL) [12]. FTL maintains an internal mapping table between logical and physical pages that translates write request to flash memory. This approach simplifies the deployment of traditional database software but it also introduces a significant overhead on flash memory as most popular database systems require frequent small log writes to store the old and updated values of data.
- **Log Structured:** The *Log Structured approach* (LSA) [19] differs from FTL as it enables databases to access flash memory directly. LSA considers the database as one large log and any update to the database (i.e., log) is appended at the end of flash memory. This approach uniformly distributes write requests on flash thus improving both performance and wearability. However, like FTL, LSA suffers from the existence of frequent small log writes. Additionally, since only log pages are written on flash, when a database object (e.g., a record) is requested it has to be recreated by reading all associated log pages.
- **In-page Logging:** A more recent approach, coined *In-page Logging* (IPL) [11] tries to overcome the problems of LSA by storing the log records associated with a specific database page on the same block thus allowing faster recreation of the database page. However, frequent small writes also affect the IPL approach.
- **AceDB flashlight:** AceDB [10] attempts to overcome the small log frequent writes problem by maintaining a logical to physical page map in RAM. Additionally, physical pages are allocated to logical pages on a transaction basis which allows for more sequential writes within a block.

In this work, we assume that the data is structured on flash media using a pack and append approach. In particular, new records arising from continuous queries are maintained in a buffer and when this buffer reaches the size of a page the records contained are packed and appended sequentially at the end of the flash, i.e., the next available empty page. This scheme is highly beneficial as it eliminates frequent small writes to flash media. Additionally this scheme does not fragment flash memory as it operates in a top-down, left-to-right manner (assuming a tabular representation of the flash media), which satisfies the Write-Constraint B.

Symbol	Definition
N	Flash capacity
M	Available main memory (pages)
B	Number of Blocks in flash
b_i	Block with identifier $i : 1 \leq i \leq B$
P	Number of Pages in a Block
p_i^j	Page with identifier i in block j

Table 2: Definition of Symbols

2.3 External Sorting

Sorting is one of the most fundamental problems in computer science and has been extensively studied by the research community. External sorting arises when the number of records to be sorted is larger than the available main memory of the system. The fundamentals of external sorting, including replacement selection and polyphase merging strategies have been thoroughly analyzed in [8] using tapes as the storage medium. Since then, many works devised solutions to integrate and improve the performance of external sorting algorithms (I/O operations, memory requirements) on hard drives disks [18, 17, 9].

Traditional external sorting algorithms usually consist of two phases: i) *internal sorting*, and ii) *external merging*. In the internal sorting phase, the input file (size S pages) is streamed into main memory, sorted on a page-to-page basis and then written back to flash (Pass 0). The resulting sorted pages are called *runs* [8]. As soon as all runs are generated, the external merging phase starts. Assuming an available memory of size M (pages), the external merging phase utilizes 1 page for output and M-1 pages for input buffers. On each subsequent pass, M-1 runs are fed into the input buffers and are merged using the output buffer to produce longer runs. The output buffer is written out to flush and emptied so as to be used in the next pass. Figure 2 illustrates an example of external sorting with M=3 (2 input and 1 output page buffers).

3. SYSTEM MODEL

In this section we will formalize our basic terminology and assumptions. The main symbols and their respective definitions are summarized in Table 2. For brevity in our descriptions let us denote *NAND-based flash memory* as *flash*. Flash contains B blocks $(b_1, b_2, ..., b_B)$ each containing P pages. We define p_i^j as page with identifier $i : 1 \leq i \leq B$ that belongs to block b_j. We also assume that the WSD has a main memory module of capacity M (M pages). According to our data model, each time a WSD acquires measurements from the sensor-board it stores them sequentially on flash in a top-down, left-to-right manner. In our experiments, we assume that a measurement is acquired every 1ms.

At periodic time intervals $\tau_1, \tau_2, ...$(e.g., every 1024ms), a user posts a query Q to the sensor network that requires

sorted data. Upon reception of Q, a sensor sorts the locally stored data and transmits it over to its parent in the formed query routing tree. Note that the amount of data, X, sorted each time is linearly correlated with τ_i, i.e., $\tau_1 = X, \tau_2 = 2X, \tau_3 = 3X, ...$ and so on.

4. THE FSORT ALGORITHM

In this section we describe our FSort algorithm. Similar to the traditional external mergesort algorithm, FSort consists of two phases, namely the internal sorting phase and the external merging phase which are described below:

1. **Internal Sorting phase:** During the internal sorting phase, M-1 buffers are created in available main memory that access blocks stored on flash in a sequential manner. Specifically, this phase produces sorted runs by streaming data pages into memory, sorting them using a top-down replacement selection algorithm and outputting/appending them to the end of the flash media.

2. **External Merging phase:** As soon as the sorted runs of the internal sorting phase are completed the external merging phase produces the sorted output by merging the sorted runs recursively until a single sorted output is produced.

Our initial sorting phase algorithm accesses flash memory in a top-down manner by initiating a page buffer array in available memory. Since in WSDs the amount of available memory is very limited we have chosen to set the size of the buffer array equal to the number of pages in a block (usually 8-16) [6, 4]. Data pages are continuously traversed until all input is examined.

The intuition behind our approach is that the measurements generated by continuous queries in WSNs are not altered erratically in small time windows. As a result, our approach generates longer runs on each pass thus minimizing the overall number of passes; decreasing in this way both read, write and erase operations. Furthermore, accessing pages in a top-down manner frees up empty space at the top (as soon as all pages in a block have been fed to the buffer array the block can be erased). This is particularly useful as the erase of individual pages is not permitted in flash memory due to the Erase-Constraint mentioned in Section 1. Algorithm 1 outlines the operation of the internal sorting phase of FSort.

Assuming that we have D unsorted data pages and an available main memory M, our algorithm starts out by initiating a P-page buffer array, coined Buf, that will be used to traverse the flash memory in a top-down manner (line 2). We also set the counter $CountEmpty$ to zero (line 3); this counter will be used to halt the procedure when all buffers are accessing empty or new pages (i.e., we have read all data pages). The algorithm continues by streaming the pages of the first block into the Buf array (lines 5-7). Next, the algorithm proceeds continuously by selecting in each step the smallest key (or highest depending on the request) including the buffer index, $BIndex$, that the key was discovered (line 10) and forwards it to the output buffer (line 11). This is done efficiently by utilizing a selection tree. As soon as the key is outputted onto flash we need to retrieve the next page for Buf_{Bindex}. To do this, we store in the header of the buffer the current page position

Algorithm 1 : FSort

Input: D unsorted data pages, Available Flash Memory M, Buf:$P = (M-1)$ input buffers, min:1 output buffer
Output: D **sorted** data pages

1: **procedure** SORT
2: Allocate(Buf(P)); //Allocate P buffers for input
3: CountEmpty=0; //Terminating counter
4: //Initiate the Buf buffer array
5: **for** $i = 1$ to P **do**
6: set $Buf_i = p_i^0$;
7: **end for**
8: **while** true **do**
9: //find the smallest key (min) and its index in Buf
10: set $(min, BIndex)$ = SelectionTree(Buf);
11: Output(min); //write sorted output to flash
12: //find the next page that the Buf_{BInx} must retrieve
13: set $Buf_{BIndex} -> lastPageRead += P$;
14: set $PIndex = Buf_{BIndex} -> lastPageRead$;
15: //check if empty or new page
16: **if** (!IsEmptyOrNew(p_{PIndex}^{BIndex}) **then**
17: set $Buf_{BIndex} = p_{PIndex}^{BIndex}$;
18: **else**
19: set $CountEmpty++$;
20: **end if**
21: **if** ($CountEmpty == P$) **then**
22: //all buffers read empty pages
23: Break;
24: **end if**
25: **end while**
26: Merge();
27: **end procedure**

(annotated as "$Buf_i -> lastPageRead$") and assign it to a temporary variable $PIndex$ while increasing it by 1 at the same time (to access the next page) (line 13-14). If the page pointed by $PIndex$ is empty or if a sorted page has been outputted to this position (line 16) then we simply increase counter $CountEmpty$ (line 19); this also removes the Buffer from the selection tree, otherwise we load the new page to the current buffer (line 17). Finally, we test if there exists more than one buffer accessing data pages and if not we exit the loop (line 21-24). The procedure ends by recursively merging the runs (line 26).

As soon as the internal sorting phase begins, the sorted runs are merged into longer runs by the external merging phase. This procedure continuous recursively and terminates when a single sorted run is generated.

5. EXPERIMENTS

In this section we present an experimental evaluation of the FSort algorithm.

5.1 Experimental Methodology

In order to evaluate the efficiency of our approach we developed a flash-based simulator in C++. Our simulator supports both NOR and NAND-based flash media settings and the functions *read* and *write*. All experiments were performed on a PC running Ubuntu Linux with 2GB of memory and an Intel Core 2 Duo CPU running at 2.4GHz.

Power Model: We adopt the power model of the RISE platform [26] which consists of the following parameters: We use a 14.8 MHz 8051 core operating at 3.3V with the following current consumption 14.8mA (On), 8.2mA (Idle), 0.2ţA (Off). We utilize a 128MB flash media with a page

size of 512B and a block size of 16KB. The current to read, write and block delete was 1.17mA, 37mA and 57μA and the time to read in the three pre-mentioned states was 6.25ms, 6.25ms, 2.27ms.

Datasets: We utilize a real trace of sensor readings that is collected from 58 sensors deployed at the premises of the Intel Research in Berkeley [7] between February 28th and April 5th, 2004. The sensors utilized in the deployment were equipped with weather boards and collected time-stamped topology information along with humidity, temperature, light and voltage values once every 31 seconds (i.e., the epoch). The dataset includes 2.3 million readings collected from these sensors.

Algorithms: We first implement the External InsertionSort and External MergeSort algorithms for comparing the online vs. the offline sorting approaches on flash-based media. We then implement the FSort algorithm and compare it against a version of the External MergeSort algorithm that utilizes M-1 buffer pages for input and one for output.

5.2 Experimental Results

In this Section we present the results of our experimental study.

A. Online vs. Offline Sorting on Flash
In our first experimental series we have conducted a micro benchmark that shows the deficiencies of online sorting on flash memories used in WSDs. We have implemented: i) the InsertionSort algorithm, and ii) the External MergeSort algorithm for benchmarking the online and offline sorting approaches respectively. We feed 1000 records (1 record per page) to our simulator and record the number of read, write and erase operations which we then translate into time and power consumption. Note, that we do not consider the energy required for executing the query as both approaches require the same energy for the given operation.

The results for both NOR and NAND-based flash memories are depicted on Table 3. We observe that the offline sorting approach greatly outperforms the online approach on NAND and NOR-based flash memory both with regards to power and time. Specifically, we observe a decrease of power consumption and time by 3 orders of magnitudes on NAND-based flash memory and 2 orders of magnitudes on NOR-based flash memory.

B. Power Consumption
In our second experimental series we measure the energy and time performance of FSort and compare it with the external Mergesort algorithm which the latter utilizes M-1 input buffers. We run experiments using three different attributes included in our dataset, namely temperature, humidity and light. The sensor records one attribute every 1ms and a query requests sorted results every 1024ms. Our simulator executes both the external MergeSort and FSort algorithms every 1024ms and collects energy and time statistics every 1024=1K records. Each experiment is executed 10 times and the average is collected.

Figure 3 illustrates the results of our evaluation with regards to energy consumption[1] for the light attribute; similar results apply to the humidity and temperature attributes.

Power (mJ)		Read	Write	Erase
NAND-flash	Online	47815	1519376	105789
	Offline	96	3052	213
NOR-flash	Online	5998	189921	48279
	Offline	96	3052	776
Time (s)		Read	Write	Erase
NAND-flash	Online	12452	12446	563
	Offline	25	25	1
NOR-flash	Online	1562	1546	122
	Offline	25	25	2

Table 3: Performance comparison of the Online (InsertionSort) and Offline (MergeSort) Sorting approaches

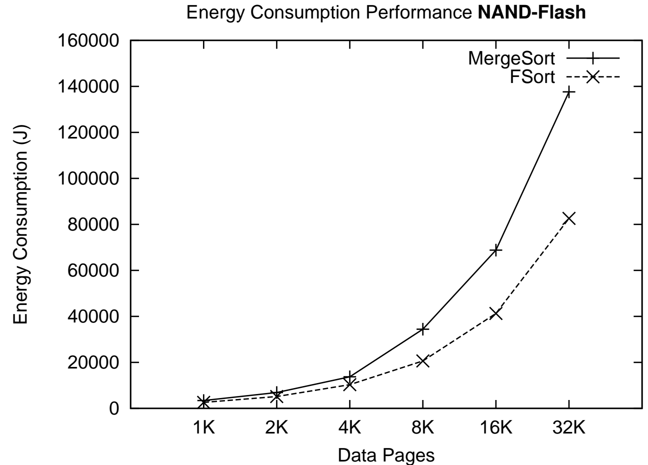

Figure 3: Energy Consumption performance comparison of External MergeSort and FSort for the light attribute. Sorting occurs every 1K records.

We observe that FSort always maintains a competitive advantage over MergeSort for all data sizes (1-32K). The initial energy savings for 1K are ≈25% but as the number of records increases the energy savings also increase reaching as high as ≈41%. This is augmented to the fact that FSort minimizes write and erase operations by releasing continuous blocks of pages every time which are the most expensive operations; this also increases the longevity of flash memory.

6. CONCLUSIONS

In this paper we proposed a novel flash-aware and power-efficient algorithm coined FSort. FSort takes into account the limitations of WSNs as well as the characteristics / constraints of flash memory used in WSDs and improves both overall execution and response time performance.

Our experiments with real traces show that FSort, greatly outperforms the traditional external mergesort approach both with regards to time and energy performance (25-41% decrease in overall energy consumption), by minimizing write and erase operations which are the most expensive operations on flash. This also increases the longevity of flash

[1]We omit the results of time performance as they are highly correlated with energy consumption and do not present any new findings.

memory as it only allows a limited number of write operations on each cell.

In the future we plan to extend our work by including index structures in order to access the data more efficiently (i.e., minimize read operations). Additionally, we plan to assess the efficiency of our approach on other popular types of flash memory like SSDs.

Acknowledgments

This work was supported in part by the Open University of Cyprus under the project SenseView, the US National Science Foundation under the project AQSIOS (#IIS-0534531), the European Union under the project mPower (#034707) and IPAC (#224395), and the Cyprus national project GEITONIA (#ΠΛΥΠΗ/0506/31).

7. REFERENCES

[1] Aly M., Chrysanthis P.k., Pruhs K., "Decomposing Data-Centric Storage Query Hot-spots in Sensor Networks", In *MOBIQUITOUS*, 2006.

[2] Aly M., Pruhs K., Chrysanthis P.k., "KDDCS: a load-balanced in-network data-centric storage scheme for sensor networks", In *CIKM* pp.317-326, 2006.

[3] Andreou P., Zeinalipour-Yazti D., Vassiliadou M., Chrysanthis P.K., Samaras G., "KSpot: Effectively Monitoring the K Most Important Events in a Wireless Sensor Network", In *ICDE*, 2009.

[4] Atmel AT45DB041B, http://www.atmel.com/

[5] Bouganim L., Jonsson B.T., Bonnet P., "uFLIP:Understanding Flash IO Patterns", In *CIDR*, 2009.

[6] Crossbow Technology Inc., http://www.xbow.com/

[7] Intel Lab Data http://db.csail.mit.edu/labdata/labdata.html

[8] Knuth D.E., "The Art of Computer Programming: Sorting and Searching" Addison Wesley, Vol. 3, April 1997, ISBN:0-201-89685-0.

[9] Larson P., Graefe G., "Memory management during run generation in external sorting", In *ACM SIGMOD*, pp.472-483, 1998.

[10] Lee K.Y., Kim H., Woo K., Chung Y.D., Kim M.H., "Design and implementation of MLC NAND flash-based DBMS for mobile devices", In *Journal of Systems and Software*, 2009.

[11] Lee S., Moon B., "Design of Flash-Based DBMS: An In-Page Logging Approach", In *ACM SIGMOD*, pp.1-10, 2007.

[12] Lee S., Park D., Chung T., Lee D., Park S., Song, H., "A log buffer-based flash translation layer using fully-associative sector translation", In *ACM (TECS)*, Vol.6, No.3, pp.18, 2007.

[13] Madden S.R., Franklin M.J., Hellerstein J.M., Hong W., "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", In *USENIX OSDI*, Vol.36, pp.131-146, 2002.

[14] Masuoka F., Iizuka H., "Semiconductor memory device and method for manufacturing the same ", US patent:4531203, July,1985.

[15] Elmasri R., Navathe S.B., "Fundamentals of Database Systems (5th Edition):Chapter:15.3.2" Addison Wesley, 2007, ISBN:0-321-41506-X.

[16] Ning X., Sumit R., Krishna K.C., Deepak G., Alan B., Ramesh G., Deborah E., "A wireless sensor network For structural monitoring", In *ACM SenSys*, pp.13-24, 2004.

[17] Nyberg C., Barclay T., Cvetanovic Z., Gray J., Lomet D., "AlphaSort: a cache-sensitive parallel external sort", In *VLDB*, Vol.4, No.4, pp.603-628, 1995.

[18] Pang H., Carey M.J., Livny M., "Memory-Adaptive External Sorting", In *VLDB*, pp.618-629, 1993.

[19] Rosenblum M., Ousterhout J.K., "The design and implementation of a log-structured file system", In *ACM TOCS*, Vol.10, No.1, pp.26-52, 1992.

[20] Sadler C., Zhang P., Martonosi M., Lyon S., "Hardware Design Experiences in ZebraNet", In *ACM SenSys*, pp.227-238, 2004.

[21] Selavo L., Wood A., Cao Q., Sookoor T., Liu H., Srinivasan A., Wu Y., Kang W., Stankovic J., Yound D., Porter J., "LUSTER: wireless sensor network for environmental research", In *ACM SenSys*, pp.103-116, 2007.

[22] Sharaf M.A. , Beaver J., Labrinidis A. and Chrysanthis P.K., "TiNA: a scheme for temporal coherency-aware in-network aggregation", In *MobiDe*, pp.69-76, 2003.

[23] Szewczyk R., Mainwaring A., Polastre J., Anderson J., Culler D., "An Analysis of a Large Scale Habitat Monitoring Application", In *ACM SenSys*, pp.214-226, 2004.

[24] Yao Y., Gehrke J.E., "The cougar approach to in-network query processing in sensor networks", In *SIGMOD Record*, Vol.32, No.3, pp.9-18, 2002.

[25] Zeinalipour-Yazti D., Andreou P., Chrysanthis P. and Samaras G., "MINT Views: Materialized In-Network Top-k Views in Sensor Networks", In IEEE *MDM*, 2007.

[26] Zeinalipour-Yazti D., Lin S., Kalogeraki V., Gunopulos D., Najjar W., "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices", In *Usenix FAST*, pp.31-44, 2005.