Adaptive Flash Sorting for Memory-Constrained Embedded Devices

Ramon Lawrence
ramon.lawrence@ubc.ca
University of British Columbia
Kelowna, BC, Canada

ABSTRACT

Databases running on embedded devices require efficient sorting algorithms for aggregation, joins, and output ordering. Previous research has produced algorithms optimized for small-memory, flash embedded devices that either use multiple read passes to avoid writes or optimize the external merge sort algorithm. Depending on the data input distribution and memory characteristics, neither approach always outperforms the other. This work produces an adaptive flash sorting algorithm that dynamically determines the best sorting approach at run-time. Experimental results demonstrate that the adaptive sorting algorithm combines the best features of both approaches and allows overall superior performance.

CCS CONCEPTS

• **Theory of computation → Sorting and searching**; *Data structures and algorithms for data management*; • **Computer systems organization → Embedded software**; • **Hardware →** *Sensor devices and platforms*; • **Software and its engineering →** *Embedded software*.

KEYWORDS

adaptive, sorting, Arduino, embedded, database, Internet of Things

ACM Reference Format:
Ramon Lawrence. 2021. Adaptive Flash Sorting for Memory-Constrained Embedded Devices. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3412841.3441914

1 INTRODUCTION

There has been increased focus on processing data on the edge of the network to reduce network transmissions, improve response time, and minimize energy usage. Edge data processing requires executing database system functionality on edge devices, which often have limited CPU and memory resources. Implementing sensor network databases [14, 16] and databases on embedded systems [5, 7, 9] have been widely studied. A fundamental challenge is implementing external sorting algorithms [11] on embedded devices. Sorting is required for analysis using grouping and aggregation, duplicate elimination, joins, and output ordering.

Previous research on embedded sorting can be classified in two basic categories. The first approach relies on reading the input data multiple times to select the next values in the sorted order. This approach includes algorithms such as FAST [15] and MinSort [8]. Avoiding writes is beneficial, but the disadvantage is algorithm performance is data dependent. The other approach adapts external merge sort by using replacement selection (FSort [6]) or reducing the buffers used (NOBsort [13]). These algorithms have predictable performance, but may be slower than reading the data multiple times for data distributions with low variability.

The contribution is the development and validation of an adaptive, flash-memory optimized sort for small-memory embedded devices. The algorithm analyzes the data distribution during the first pass to determine the best sorting approach to use. Theoretical and experimental results demonstrate that the adaptive algorithm combines the best performance characteristics of the two approaches with limited overhead. The algorithm can be used for all data distributions and input sizes and works with any flash memory. Adaptive sorting allows for efficient implementation of databases on embedded devices.

The paper contains a background on embedded external sorting, a description of the adaptive sorting algorithm with a theoretical analysis, and an experimental section demonstrating performance benefits. The paper closes with future work and conclusions.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8104-8/21/03...$15.00
https://doi.org/10.1145/3412841.3441914

2 BACKGROUND

External sorting is required when the data set is larger than main memory. Small memory embedded devices may only have 2 to 64 KB of RAM but large flash storage. Example hardware includes ATMega2560 [2] (16 MHz, 8 KB) that are used in Arduino devices, PIC18F57Q43 [4] (16 MHz, 8 KB), and ATtiny1634 [3] (12 MHz, 16 KB). Sorting is required for both stand-alone databases on embedded devices and networks of database sensors [14, 16]. Database operations including ordering, joins, and gropuing and aggregation use sorting [11].

The notation used is in Figure 1. N denotes the data input size in blocks, and M is the sort memory size. M is typically smaller than the physical memory as the device must use memory for other functions besides sorting. The number of distinct values for the sort key is D.

2.1 External Sorting by Multiple Read Passes

Given that reads in flash memory are often significantly faster than writes, external sorting algorithms such as FAST [15] and MinSort [8] favor multiple read passes over writing. In FAST, each

Notation	Definition
N	Input size in blocks
M	Sort memory size in blocks
B	Number of values in one block
D	Number of distinct values of sort key
R	Block read operation cost
W	Block write operation cost
S	Number of scans (complete passes) through data
L	Number of intermediate sorted runs generated

Figure 1: Sorting Parameters

read pass reads the next *n* elements in sorted order where *n* is the number of elements that can fit in the sort memory M.

MinSort uses a small in-memory region index that stores the minimum value in each region. Given sufficient memory, a region is a single block, but may consist of multiple adjacent blocks to ensure the index is small enough to fit in memory. The algorithm then repeatedly finds the region with the smallest value according to the index, searches the region for that smallest value, and then updates the index entry to the next smallest value in the region. The algorithm will read each block as many times as there are distinct sort key values in the region, denoted by D_R. Run generation and merging is combined into a single phase and no writing is performed.

Both algorithms are highly affected by data input characteristics. For FAST, the ratio of input size to memory size $\frac{N}{M}$ is critical. When sorting large inputs, FAST is very slow. MinSort is affected by the number of distinct sort key values D as it makes a read pass for every unique value in the input. In many embedded database applications, D may be small if processing sensor data (e.g. less than 20), where in other cases it may be very large. Previous research [8] has shown MinSort is faster than FAST.

2.2 External Merge Sort Variants

External merge sort is widely used in database systems [11], and variants have been implemented for flash memory embedded sorting. The external merge sort algorithm has two phases. The *run generation phase* builds sorted runs of the input in one of two ways. The load-sort-store method reads M blocks from the input into memory, sorts them using a main memory sort, then writes the sorted data to storage as an intermediate file called a *run*. This repeats until all input data is in sorted runs.

Replacement selection [10] builds sorted runs by reading the input data into a heap and then writing out records in sorted order as the heap fills the available memory. When no more records larger than the largest value wrote out so far in the run are in the heap, then the previous run is closed and a new run is started. Replacement selection has been shown to produce runs of size approximately $2M$ for random data. For sorted or near-sorted data, the number of runs is greatly reduced (down to 1 run for completely sorted data). For reverse sorted data, all runs are of size M. Thus, replacement selection offers promise for data collected by embedded devices that may be partially sorted.

The *run merge phase* combines runs into a sorted output by merging $M - 1$ runs at a time and using the other memory buffer as an output buffer. If there are more than $M - 1$ sorted runs, the merge phase is performed recursively. Given input data size of N blocks partitioned into L runs during the run generation phase, the number of merge passes S is $\lceil log_{M-1}(L) \rceil$. The number of block reads is $2*N*S$, and the number of block writes is $2*N*S$ (includes the cost of writing the final output).

FSort [6] implemented replacement selection for flash sorting. NOBsort [13] modified the algorithm to avoid using an output buffer during merging, which reduces the number of merge passes ($\lceil log_M(L) \rceil$). The advantage of algorithms based on external merge sort is their predictable I/O that is independent of the data distribution. The I/Os performed are based only on the input size N and memory available M. The disadvantage is that multiple merge passes may be required with costly writes. Previous work has shown NOBsort [13] as the highest performing implementation with consistent performance. However, NOBsort does not always execute faster than MinSort, depending on the value of D.

Figure 2 contains I/O performance formulas for the various external sorting algorithms. NOBsort may be implemented with run generation using either load-sort-store or replacement selection (NOBsort (replace) in the table).

3 ADAPTIVE FLASHSORT ALGORITHM

The Adaptive FlashSort algorithm combines the best properties of MinSort and NOBsort into a single algorithm that adapts to the input data distribution. For an input data distribution with a small number of distinct values, MinSort is used. Otherwise, NOBsort is the algorithm selected. The determination of the algorithm to use depends on parameters: sort memory (M), input size (N), read cost (R), write cost (W), and the number of distinct sort key values (D). The parameters that vary between executions are N and D, which can be estimated during the first read pass through the data. MinSort has improved performance compared to NOBsort when $D_R < S * (1 + \frac{W}{R})$. D_R is the number of distinct values per region, and S is the number of merge passes required.

Algorithm pseudocode is in Figure 3. The algorithm may be executed in either *optimistic* or *pessimistic* mode (lines 1-4), which determines what actions are performed in the first read pass through the data. In optimistic mode, the algorithm proceeds like MinSort in the first pass by building a region index in memory. In pessimistic mode, the algorithm proceeds like NOBsort during the first pass and performs run generation using either load-sort-store or replacement selection. In both cases, the number of distinct values per region is estimated so that after the first pass a decision is made on what algorithm to proceed with.

While performing the first read pass, an estimate of the number of distinct values per block (D_B) and per region (D_R) is performed (line 6). In pessimistic mode, this can be done easily by determining the number of key value changes in a block whenever a block is output. Since tuples in the block are sorted, the algorithm can detect differences between adjacent records in the block using the comparison function used for sorting. In optimistic mode, the number of distinct values can be estimated using various techniques such as Bloom filters [12].

The decision on the algorithm to use to complete the sort follows the performance formula $D_R < S * (1 + \frac{W}{R})$ (line 8). For MinSort to

Algorithm	Reads (Gen)	Writes (Gen)	Reads (Merge)	Writes (Merge)
external merge sort	$N * R$	$N * W$	$N * \lceil log_{M-1}(\lceil \frac{N}{M} \rceil) \rceil * R$	$N * \lceil log_{M-1}(\lceil \frac{N}{M} \rceil) \rceil * W$
MinSort	$N * (1 + D) * R$	$N * W$	0	0
FAST	$N * \frac{N}{M} * R$	$N * W$	0	0
FSort	$N * R$	$N * W$	$N * \lceil log_{M-1}(\lceil \frac{N}{2M} \rceil) \rceil * R$	$N * \lceil log_{M-1}(\lceil \frac{N}{2M} \rceil) \rceil * W$
NOBsort	$N * R$	$N * W$	$N * \lceil log_{M}(\lceil \frac{N}{M} \rceil) \rceil * R$	$N * \lceil log_{M}(\lceil \frac{N}{M} \rceil) \rceil * W$
NOBsort (replace)	$N * R$	$N * W$	$N * \lceil log_{M}(\lceil \frac{N}{2M} \rceil) \rceil * R$	$N * \lceil log_{M}(\lceil \frac{N}{2M} \rceil) \rceil * W$

Figure 2: Existing Sort Algorithms I/O Performance Formulas

```
1   if (optimistic)
2       Perform read pass to generate region min value index
3   else
4       Create sorted runs (using replacement selection)
5
6   Estimate number of distinct values (D) during first pass
7
8   if D < S*(1+W/R)
9       if !optimistic and min index fits in memory
10          perform optimized MinSort with sublists
11      else
12          perform MinSort on the file of sorted runs
13  else
14      perform NOBsort
```

Figure 3: Adaptive FlashSort Implementation

be selected, typically D_R should be small (i.e. $D_R <= 32$), but this depends on the relative write cost versus reads. On SD cards, $\frac{W}{R}$ may be between 1.5 to 3, while for raw memory $\frac{W}{R}$ may be 5 or higher.

If the algorithm was optimistic and the number of distinct values is too high for MinSort execution, then the first read pass is discarded, and the algorithm proceeds with run generation and using NOBsort. If the algorithm was pessimistic and the number of distinct values is small so MinSort should be run, then MinSort is applied on the file containing the sorted runs rather than the input data file. Depending on the number of runs, this may allow for a more efficient implementation of MinSort.

Once the algorithm decision has been made, there are two further optimizations. If using MinSort was the best choice and the algorithm started in pessimistic mode, then the first pass produced an input file consisting of sorted sublists. If the minimum value index of the sublists fits in the memory available, then a new variant of MinSort can be performed (lines 9-10). This variant, called MinSortSublist, has each sorted sublist as a region. Each index entry consists of the minimum key in the region and a file offset in the region. This allows for the next value in the region to be quickly found without scanning the region. The downside is that each index entry requires another 4 bytes, so if the key is 4 bytes long, each index entry is 8 bytes. Thus each index entry may be twice as large, which results in the number of regions that can be indexed decreasing by a factor of two.

The second optimization improves on NOBsort execution. When the number of sublists is small enough so that the minimum value index fits into memory, there is a performance advantage to finishing the sorting using MinSortSublist rather than more merge passes. In the $M = 2$ example, an index size of 64 supports 64 sublists which would require 6 merge passes to completely sort. Performing MinSortSublist, depending on D_R, may be faster than these multiple merge passes.

The mode used may be determined by the particular implementation environment or pre-existing knowledge available to the query optimizer or user. In order to execute in optimistic mode, the input data to be sorted must be available as a random access file on storage before the algorithm begins. This is required as the read pass builds an index of minimum values in each region (or block) and uses file offsets to navigate within the data set. Although this is reasonable for many sorting situations, it is not typically the case if the sorting algorithm is used within a database system implementation that uses iterators to generate tuples one at a time. In that situation, the pessimistic mode must be used as tuples are read one at a time, sorted into sublists, and wrote out into a temporary file. This temporary file becomes the input to the remainder of the sorting algorithm.

3.1 Example and Analysis

Figure 4 shows the indexing techniques used by MinSort and the new MinSortSublist variant. MinSort produces an index for each region, which in the best case consists of a single block. In the example, each block stores 2 records, and the MinSort index consists of 4 entries. Note that if each region consists of 2 blocks, then the MinSort index would be 2 entries. Whenever the smallest value is retrieved from a region, for example 1 in the first region, the algorithm must scan the rest of the region to determine the next smallest value.

In comparison, MinSortSublist builds an index after sorted sublists have been produced. The example with $M = 2$ shows two sorted sublists. Thus, the index consists of 2 entries, with each entry consisting of the key and an offset. The offset in the figure is the record index, but in practice is an offset in the file. MinSortSublist is more efficient as it does not need to scan the entire sublist every time to find the next smallest value as the sublist is sorted. It can directly retrieve the next smallest record using the offset.

Consider sorting on a hardware device with a write-to-read ratio ($\frac{W}{R}$) of 3. In Figure 5 is a contour plot showing the relative difference between MinSort and NOBsort for varying values of distinct values, D_R, and number of merge passes, S. Values less than 0 indicate MinSort outperforms (lower left part of chart), and values above 0 indicate NOBsort outperforms.

MinSort is superior for small values of $D_R <= 16$ and has an increasing advantage as the number of passes increases. Note that as

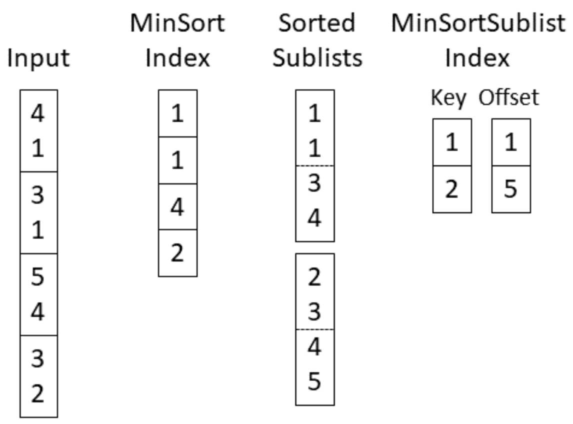

Figure 4: Sorting Index Example

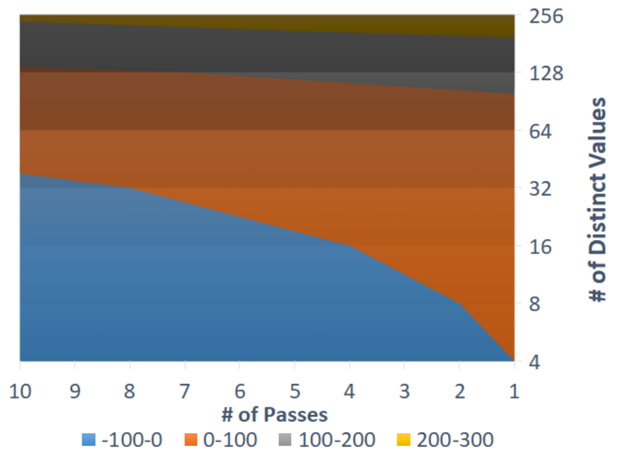

Figure 5: MinSort vs NOBsort for M=2 and Varying Number of Passes and Distinct Values

D_R increases beyond 64, the performance rapidly degrades. When all values are unique, the algorithm is far too slow, and this case must be avoided. For $M = 2$, MinSort uses one block for a read buffer and the other for the minimum index. For a 512 byte block, this means the maximum index size is 64 for MinSortSublist and 128 for regular MinSort.

The number of merge passes has an indirect impact on D_R. MinSort builds a minimum value index, and in the best case each index entry is for an individual block. For $M = 2$ case with 512 byte blocks, the index can store 128 4-byte index entries. Thus, $D_R = D_B$ which is bounded by the number of records per block. Doubling the input size to 256 blocks, requires MinSort to make each region consist of 2 adjacent blocks, so D_R may no longer be bounded by the number of records per block. In the worse case, it is equal to the number of records that can be stored in 2 blocks. Thus, D_R may grow as the input size grows and MinSort indexes larger regions. In comparison, 128 blocks requires 7 merge passes to sort by NOBsort.

The performance of Adaptive FlashSort can be precisely quantified. In pessimistic mode, if the algorithm stays with NOBsort after the first pass, then there is no additional I/O cost compared to NOBsort. There is only a minor CPU cost for the estimation of the number of distinct values. If the algorithm switches to MinSort, in the worst case it has performed an extra read and write pass ($N*(R+W)$) compared to MinSort. This overhead may be reduced if the algorithm is able to use the MinSortSublist variant. In optimistic mode, the maximum overhead compared to NOBsort is one read pass ($N * R$) if the algorithm starts with MinSort then switches to NOBsort.

4 EXPERIMENTAL EVALUATION

The experimental evaluation compared Adaptive FlashSort with NOBsort and MinSort for varying data distributions (D) and data file sizes (N). The experimental hardware was an Arduino MEGA 2560 [1] that uses a 8-bit AVR ATmega2560 microcontroller and has 256 KB of flash program memory, 8 KB of SRAM, 4 KB EEPROM, and supports clock speeds up to 16 MHz. Storage was on a 8 GB microSD card attached with an Arduino Ethernet shield. The block size was 512 bytes. There were no hardware buffers on the SD card accessible to the algorithm, so all block buffering was in RAM. With a block size of 512 bytes, the practical limit of M on the device was $M = 8$ (4 KB) as the rest of the RAM was used for other functions.

The SD card sequential read performance was 345 blocks/sec. (172 KB/s), and sequential write performance was 175 blocks/sec. (88 KB/s). Random read performance was also 345 blocks/sec. (172 KB/s). The write-to-read ratio is 1.95.

The experimental data set was chosen to be representative of the real-world data sets evaluated in MinSort [8]. Sensor data records are typically small (8 to 32 bytes) consisting of a timestamp, a sensor reading, and sensor information. The data records evaluated in [8] were collected from soil moisture sensors used for automatic irrigation. The record size is 16 bytes with a 4 byte integer key. With a block size of 512 bytes, the number of records per block B was 31 as each block had a block header that consumed some space. Data sets were randomly generated with different number of distinct values D. The results are the average of three runs.

Figure 6 shows the relative performance of Adaptive FlashSort versus NOBsort and MinSort for values of $D = \{16, 64, 256\}$. A point on the graph is calculated by taking the time for NOBsort (or MinSort) and dividing by the time taken by Adaptive FlashSort. The MinSort runs are dotted lines, and NOBsort is solid lines. A value greater than 1 means that the NOBsort (or MinSort) algorithm is slower than Adaptive FlashSort. The memory size is $M = 2$, and the input file sorted (N) increases from 2 blocks to 2048 blocks. The Adaptive FlashSort algorithm is running in pessimistic mode as the first pass is performing run generation using replacement selection.

There are several important results. First, MinSort is the superior choice for $D = 16$, and the adaptive sort dynamically switches to using the MinSort algorithm after the first pass. Although the performance is not quite as fast as if we started with MinSort initially, the algorithm is able to realize almost all the performance benefits of the switch. NOBsort at $D = 16$ is between 1.5 and 2 times slower than the adaptive sorting algorithm.

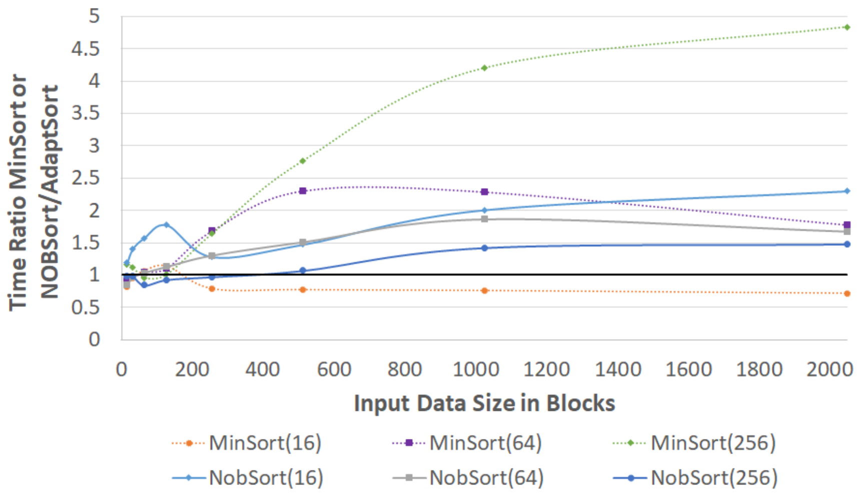

Figure 6: Time Ratio Adaptive FlashSort versus NOBsort and MinSort for Various D and M = 2

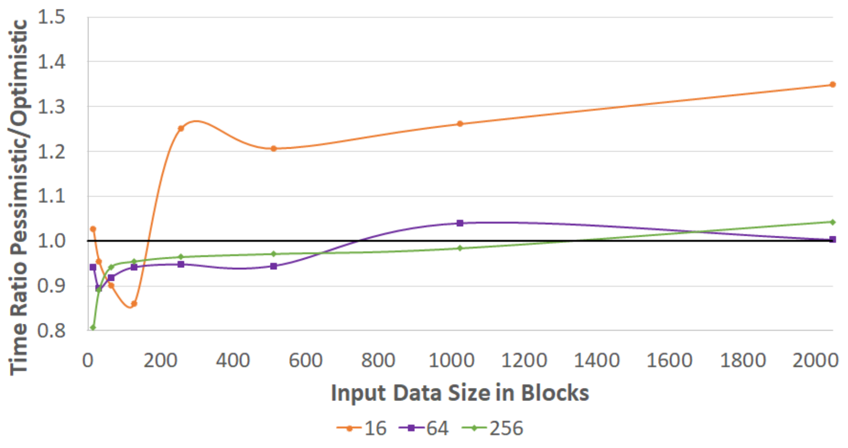

Figure 7: Time Ratio Adaptive FlashSort (pessimistic versus optimistic) for Various D and M = 2

For this experimental hardware, starting around $D = 32$ it is more efficient to perform NOBsort, which is what the adaptive algorithm executes. Using MinSort takes twice as long for $D = 64$ and over four times as long for $D = 256$. Adaptive FlashSort avoids the worst case of MinSort for larger values of D by using NOBsort.

The new contribution of the MinSortSublist algorithm that performs MinSort on sorted sublists has the most significant affect on performance. NOBsort takes 50% to 100% more time than Adaptive FlashSort for larger values of D. The reason is that Adaptive FlashSort switches from NOBsort merge passes to the MinSortSublist algorithm when the number of sublists is reduced to 64 (6 merge passes left). This has a major performance improvement as the multiple read passes are faster than the read/write merge passes. For hardware with a larger write-to-read ratio, this would be an even larger factor.

For most database use cases, the pessimistic version of Adaptive FlashSort will be used as databases typically produce records to sort using tuple-at-a-time iterators rather than having the input as a file on storage. If the optimistic algorithm can be used and it is expected that D is low, then the adaptive algorithm can have performance equivalent to MinSort. In Figure 7 is a comparison with the optimistic and pessimistic version of the algorithm for $D = \{16, 64, 256\}$. Note the scale of the y-axis is units of 0.1 (approximately 10%). There are advantages to being optimistic for small values of D. For $D = 16$, the pessimistic version is 20% to 30% slower than the optimistic version except for very small input sizes. For large values of $D = 64$ or $D = 256$, the overhead is limited to one read pass, which is a very small overhead compared to the total time for the sort. The algorithm switches to NOBsort after the first pass and performance is almost identical to the pessimistic version (within experimental variation). In most cases, there is little downside to using the optimistic version assuming that the input data is already available as a file on storage for sorting.

Adaptive FlashSort adapts as the memory available (M) increases. For larger values of M, NOBsort will be used more frequently than MinSort as increasing M quickly reduces the number of merge passes as $S = \lceil log_M(L) \rceil$. The ability to use MinSortSublist to complete the last sorting passes continues to show performance improvements compared to using NOBsort by itself.

Overall, Adaptive FlashSort is a flexible algorithm that optimizes its performance based on the input data characteristics. Switching between NOBsort and MinSort after a first pass estimating the data distribution allows it to capture the best features of both algorithms. The new MinSortSublist algorithm variant produced for this work has a dramatic affect on performance even for larger values of D. The combination of the two approaches in the single algorithm allows for superior performance.

5 CONCLUSIONS AND FUTURE WORK

Sorting on embedded devices is a critical operation for database systems and data analysis. Adaptive FlashSort combines the I/O performance consistency of NOBsort (external merge sort) with the performance advantage of MinSort when sorting data with few distinct values. The result is an algorithm that has superior performance. Using the new MinSort variant on sorted sublists produced after the run generation step was shown to have a significant performance benefit for a wide range of data distributions.

Future work will perform further experiments on other embedded hardware platforms and flash memory configurations, and investigate integrating the sort implementation into an embedded database system.

REFERENCES

[1] 2020. Arduino Homepage. http://arduino.cc

[2] 2020. Microchip ATmega2560 Specifications. https://www.microchip.com/wwwproducts/en/ATmega2560

[3] 2020. Microchip ATtiny1634 Specifications. https://www.microchip.com/wwwproducts/en/ATtiny1634

[4] 2020. Microchip PIC Specifications. https://www.microchip.com/wwwproducts/en/PIC18F57Q43

[5] Nicolas Anciaux, Luc Bouganim, and Philippe Pucheral. 2003. Memory Requirements for Query Execution in Highly Constrained Devices *(VLDB '03)*. VLDB Endowment, 694–705.

[6] Panayiotis Andreou, Orestis Spanos, Demetrios Zeinalipour-Yazti, George Samaras, and Panos K. Chrysanthis. 2009. FSort: external sorting on flash-based sensor devices. In *DMSN'09: Data Management for Sensor Networks*. 1–6.

[7] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. 2001. Towards Sensor Database Systems *(MDM '01)*. Springer-Verlag, London, UK, UK, 3–14.

[8] Tyler Cossentine and Ramon Lawrence. 2010. Fast Sorting on Flash Memory Sensor Nodes. In *Proceedings of the Fourteenth International Database Engineering and Applications Symposium (IDEAS '10)*. ACM, New York, NY, USA, 105–113. https://doi.org/10.1145/1866480.1866496

[9] Graeme Douglas and Ramon Lawrence. 2014. LittleD: a SQL database for sensor nodes and embedded applications. In *Symposium on Applied Computing*. 827–832. https://doi.org/10.1145/2554850.2554891

[10] Martin A. Goetz. 1963. Internal and tape sorting using the replacement-selection technique. *Commun. ACM* 6, 5 (1963), 201–206. https://doi.org/10.1145/366552.366556

[11] Goetz Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3, Article 10 (Sept. 2006). https://doi.org/10.1145/1132960.1132964

[12] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 499–512. https://doi.org/10.1145/3186728.3164145

[13] Riley Jackson and Ramon Lawrence. 2019. Faster Sorting for Flash Memory Embedded Devices. In *2019 IEEE Canadian Conference of Electrical and Computer Engineering, CCECE 2019, Edmonton, AB, Canada, May 5-8, 2019*. IEEE, 1–5. https://doi.org/10.1109/CCECE.2019.8861811

[14] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.* 30, 1 (March 2005), 122–173. https://doi.org/10.1145/1061318.1061322

[15] Hyoungmin Park and Kyuseok Shim. 2009. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software* 82, 8 (2009), 1298 – 1312. https://doi.org/DOI:10.1016/j.jss.2009.02.028

[16] Nicolas Tsiftes and Adam Dunkels. 2011. A Database in Every Sensor *(SenSys '11)*. ACM, New York, NY, USA, 316–332. https://doi.org/10.1145/2070942.2070974