Efficient External Sorting for Memory-Constrained Embedded Devices with Flash Memory

RILEY JACKSON, JONATHAN GRESL, and RAMON LAWRENCE, University of British Columbia, Canada

Embedded devices are ubiquitous in areas of industrial and environmental monitoring, health and safety, and consumer appliances. A common use case is data collection, processing, and performing actions based on data analysis. Although many Internet of Things (IoT) applications use the embedded device simply for data collection, there are benefits to having more data processing done closer to data collection to reduce network transmissions and power usage and provide faster response. This work implements and evaluates algorithms for sorting data on embedded devices with specific focus on the smallest memory devices. In devices with less than 4 KB of available RAM, the standard external merge sort algorithm has limited application as it requires a minimum of three memory buffers and is not flash-aware. The contribution is a memory-optimized external sorting algorithm called no output buffer sort (NOBsort) that reduces the minimum memory required for sorting, has excellent performance for sorted or near-sorted data, and sorts on external memory such as SD cards or raw flash chips. When sorting large data sets, no output buffer sort reduces I/O and execution time by between 20 to 35% compared to standard external merge sort.

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.*

Additional Key Words and Phrases: sorting, Arduino, embedded, performance, Internet of Things

ACM Reference Format:
Riley Jackson, Jonathan Gresl, and Ramon Lawrence. 2021. Efficient External Sorting for Memory-Constrained Embedded Devices with Flash Memory. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2021), 22 pages. https://doi.org/10.1145/3446976

1 INTRODUCTION

Embedded devices are typically optimized to minimize cost while achieving an acceptable level of performance. This tradeoff results in designers selecting hardware configurations with the minimal CPU and memory resources to reduce cost. If an embedded device is performing data processing in addition to data collection, the resources available may be limited. A common data processing task is sorting for ordering output, performing aggregations and calculations, and data linking and joins. Main-memory sorting algorithms rapidly hit limits when the memory available on the smallest devices ranges from 1 KB to 128 KB. External merge sort [14] allows the use of flash memory to sort large data sets, but was developed for servers with increased resources and capabilities.

Prior work has adapted external merge sort for sorting on solid state drives (SSDs) on servers. MONTRES [17] adapts to the slower write versus read performance of SSDs by using optimizations that favor reading over writing. Algorithms

Authors' address: Riley Jackson; Jonathan Gresl; Ramon Lawrence, ramon.lawrence@ubc.ca, University of British Columbia, 3187 University Way, Kelowna, BC, Canada, V1V 1V7.

Manuscript submitted to ACM

such as FAST [21] and MinSort [11] designed for embedded devices also make the read versus write tradeoff and use less memory, but increasing reads often reduces overall performance.

This work develops and experimentally validates no output buffer sort (NOBsort), which is an optimization of external merge sort for small-memory embedded devices that reduces the minimum memory consumption to two buffers by eliminating the output buffer used during the merge step. By decreasing the minimum memory consumption, external sorting can be used on more devices with higher performance. Reducing the memory usage during merging allows for fewer merge passes with the same amount of memory, which results in higher overall performance. The algorithm functions with any form of flash memory (SD cards, raw flash chips, etc.) and does not require a file system or flash translation layer (FTL).

A version of the approach was presented in [16]. This work extends the algorithm to support reduced storage consumption on flash, minimizing swaps using heaps, support for variable run sizes produced by replacement selection, block headers for variable sized blocks, and iterator input. The experimental evaluation compares using replacement selection versus the load-sort-store method for small memory on data with various sorted levels and identifies the memory characteristics when NOBsort is superior to external merge sort. NOBsort improves performance by 20 to 35% over external merge sort, and replacement selection was shown to be the best approach for run generation.

The following sections provide background on external sorting with specific focus on embedded devices. NOBsort and its optimizations are discussed in Section 3 with a theoretical evaluation in Section 4. Experimental results are in Section 5. Conclusions and future work are in Section 6.

2 BACKGROUND

2.1 External Sorting

For notation, we will use N to denote the data input size in blocks, and M to be the memory size in blocks for use in sorting. Note that M may be considerably smaller than the physical memory on the device as the sorting algorithm will typically be able to use only a portion of device memory. The number of distinct values for the sort key is D, and the number of records that can be stored in a block is B. For example, a record size of 16 bytes may include a 32-bit integer sort key. The sort key is the field being sorted on, which is often an integer or numeric value, but may also be a string or other data type. With a block size of 512 bytes, B could be at most 32 records per block. The cost to read and write a block from the memory device will be denoted as R and W respectively. Many algorithms perform a complete scan or pass through the data. The number of scans used is denoted by S. For algorithms that generate sorted runs, the number of these runs is denoted by L. For example, the external merge sort algorithm performs a pass through the data to generate L sorted runs and then performs $S = \lceil log_{M-1}(L) \rceil$ merge passes to generate the sorted output. A summary of these parameters is in Figure 1.

Database operations including ordering, joins, and aggregation use sorting [14]. The external merge sort algorithm has two phases. The algorithm is given M blocks in memory. 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 [13] 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

Notation	Definition
N	Input file size in blocks
M	Main 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

Fig. 1. Sorting Parameters

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.

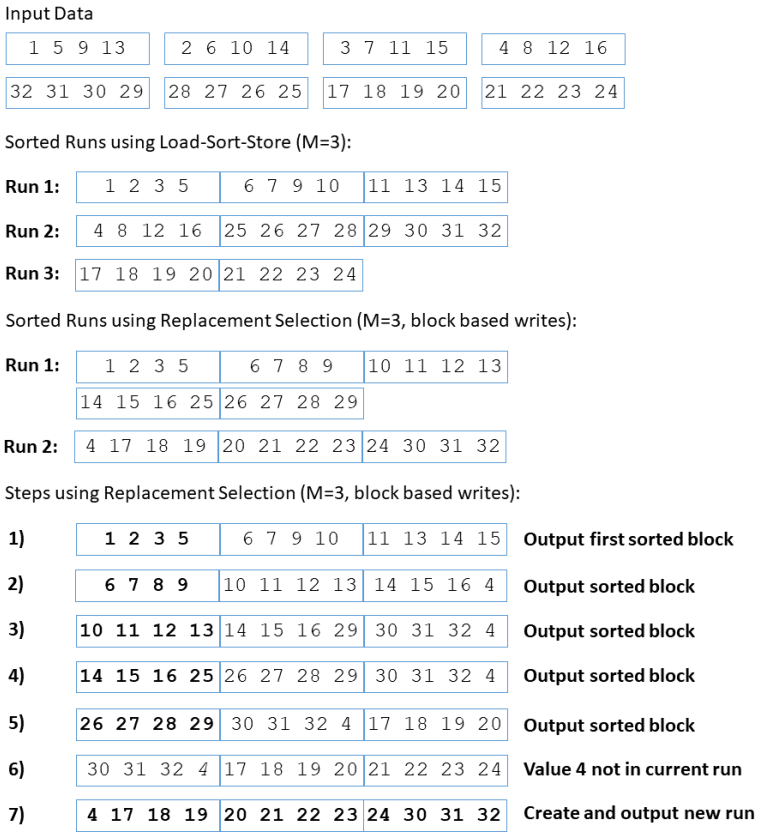

Fig. 2. Run Generation using Load-Sort-Store and Replacement Selection

An example of both run generation techniques is in Figure 2. For a memory size $M = 3$, load-sort-store reads 3 input blocks at a time, sorts them, and outputs the sorted run. The last run only has 2 blocks. The output of replacement

selection is shown as well as the step-by-step processing. The example assumes an output block must be written (not just a single record), and the data is shown in sorted order for easier readability (rather than as an array-based heap which is how it is actually implemented). Step 1 reads three blocks into memory and organizes them into an array-based heap. Then the smallest block is written to output for run 1. In Step 2, the next input block is read (into first block buffer) and the values merged into the heap. Note that the value 4 is the last value in sorted order as it must be in the next sorted run as the largest value output in run 1 already is 5. The smallest block is output to the run, and then the next input block is read and merged into the heap. This process continues for the next few input blocks. At step 6, we have a block with a value (4) that cannot be put into the current run. Thus, we start a new run. Since we have read all input, we can sort and output the remaining blocks into run 2. If there was more data, the first block would be output, and the algorithm would build a heap and proceed as previously. Note that replacement selection resulted in fewer runs and that the runs are not all the same size.

Once sorted runs have been created, the *run merge phase* combines runs into a sorted output. The merge phase merges $M-1$ runs at a time and uses 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). An example of the run merge phase is in Figure 3. When merging runs, one of the memory buffers is used as output, so with $M = 3$, two runs can be merged at a time. With three input runs, the result is runs 1 and 2 are merged together first. Then that output is merged with run 3 to generate the sorted result. When performing a merge, the algorithm continually searches for the smallest value across the runs, puts that value in the output block, and reads input blocks as needed.

Optimizations [14] for external merge sort include double buffering to reduce I/Os and parallelization during merging. Natural runs were defined in [18] that were sequences of blocks with non-overlapping values but not sorted within the block. These blocks were sorted during the merge phase in [18].

2.2 Flash Memory Embedded Devices

Sorting on embedded devices requires optimizing for flash memory characteristics and handling the reduced CPU and memory resources available. There is a wide variety of embedded architectures. The test architecture used in this work is the Arduino MEGA2560 [1] that has a 8-bit, 16 MHz CPU and 8 KB SRAM. The Arduino platform is widely used for rapid prototyping and embedded development. Although the Arduino platform is very common, there are numerous processors with similar characteristics, specifically low RAM, where algorithms must be memory efficient in order to execute. Figure 4 contains some examples of embedded processors.

Flash memory storage devices have two key properties. Due to the nature of the memory hardware, writing may take considerably more time than reading. Unlike hard drives and RAM, writing over a previously written location is not possible without erasing the memory location first. An erase is an expensive operation that often is performed in units of multiple blocks at a time. A write is more efficient when performed at a different location every time a data value is updated. Thus, in solid-state drives and SD cards, a flash translation layer (FTL) is used to map logical addresses to physical addresses. The software writes to a logical address space, and the FTL writes to physical blocks on the device. This allows the software to be unaware of actual physical block locations and the performance issues with overwriting an existing block. The FTL also performs wear leveling across the device that improves memory lifetime and overall performance.

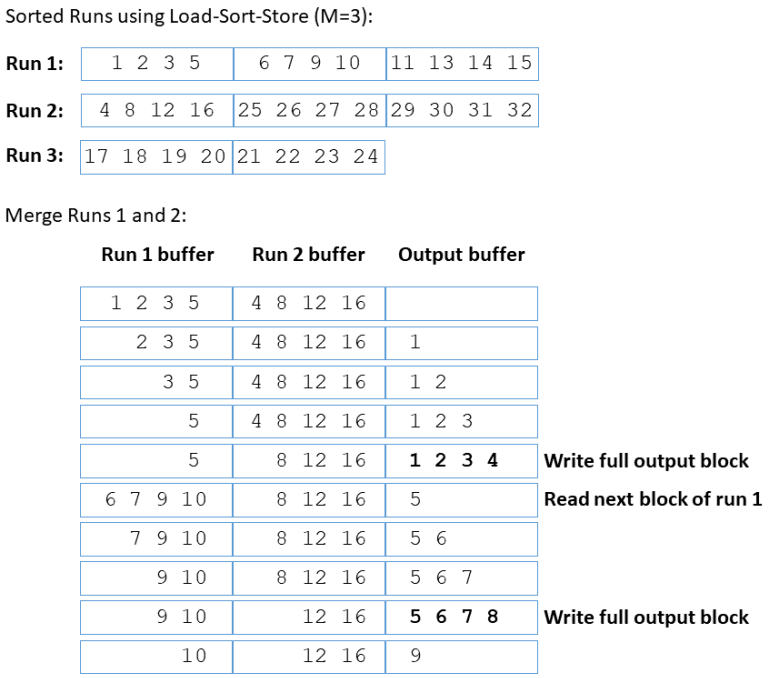

Fig. 3. Run Merge Phase of External Merge Sort

Device Name	CPU	Memory
ATtiny48 (Microchip) [6]	12 MHz	8 KB
ATMega2560 (Microchip) [3]	16 MHz	8 KB
LPC8N04 (NXP) [8]	8 MHz	8 KB
PIC18F57Q43 (Microchip) [7]	16 MHz	8 KB
MSP430FR6007 (Texas Instruments) [9]	16 MHz	8 KB
ATtiny1634 (Microchip) [5]	12 MHz	16 KB
AT89LP3240 (Microchip) [2]	20 MHz	32 KB
ATSAMD21 (Microchip) [4]	48 MHz	32 KB

Fig. 4. Example Embedded Devices

2.3 Flash Memory External Sorting

Given the flash memory properties, external sorting optimizations can be categorized into two areas. A common approach is reducing write operations during run generation by reading the input multiple times [21] or searching for minimum values using random reads [11, 19]. Optimizations are also possible in the run merge phase by favoring reads over writes or using data indexing and random reads to read data in order [17, 18].

2.3.1 Sorting on SSDs. MONTRES [17] performs external sorting for SSDs with optimizations in both phases of external sort. In run generation, run sizes are maximized by using data indexing to find blocks with minimum values and expand existing runs. Data indexing is used in the run merge phase to always retrieve the block with the next smallest value.

The index stores the minimum value of each block in every run. Unlike the merge phase in external merge sort, this allows merging in one pass as the index allows the block containing the next smallest value to be put into memory. MONTRES improved on external merge sort in cases when the input size is a large multiple of the memory size. In the I/O formula for MONTRES, P_R and P_W are the number of additional reads and writes used during the run generation phase to build larger runs and directly output records to the sorted file. G is the number of blocks directly written to the sorted file. α is the number of additional times a block must be read during run merging and is dependent on the input data distribution. It is bounded by $N * \frac{B}{M}$.

Although block indexing is powerful, as the data size grows the index becomes too large for the memory in embedded devices. For example, the index requires 1K of RAM for every 128 blocks of data to sort. That memory usage is reasonable on servers when sorting data for SSDs, but quickly exceeds the memory size of small devices, which often have between 4K and 32KB of RAM. Deciding when to index and when to retrieve data without an index was studied in [23], which produced a decision rule when to use clustered (sequential-based) or unclustered (index-based) sorting.

2.3.2 Sorting on Memory-Constrained Embedded Devices. Embedded devices are processing more data on device rather than simply transmitting raw data over the network for analysis. Given the energy cost of network transmissions, it is valuable to only transmit data as needed. This requires the ability to store and process data on the device. Foundational work on sensor networks such as TinyDB [20] treated the network as a distributed database system. Queries were sent to the devices and processed on the locally stored data to return results. In this architecture, each device must be capable of performing database operations, and sorting is critical for query processing. In many environmental and monitoring applications, the amount of data transmitted can be significantly reduced by performing aggregations on the data before transmission. Sort-based aggregation requires the use of external sorting.

Embedded external sorting algorithms often read the input data multiple times to avoid costly writes and compensate for the limited SRAM available. Flash MinSort [11] built a data index of the minimum values in each region of adjacent blocks. Similar to MONTRES, this produces sorted data by continuously reading the block into memory with the next smallest input. Since the index is per region not per block, it can adapt to the memory available including memory as small as 100 bytes (at the minimal size, the index would store only 2 values for 2 regions). The tradeoff is that many block reads are now required within the region to find the exact block with the minimum value. Run generation and merging is combined into a single phase and no writing is performed, although the data may be read multiple times.

The FAST [21] algorithm reads the input data multiple times. For each read pass, it outputs the next smallest m values where m is the number of records that can fit in memory. When sorting larger data sets, FAST can be used as the run generation phase for external merge sort. This algorithm, denoted as FAST(N), produces fewer runs by using a configurable parameter Q which is the sublist size produced during run generation. Q can be larger than M by using multiple scans of the input with the FAST algorithm. A prior experimental comparison [11] between MinSort and FAST showed that MinSort was superior for sorting on small-memory embedded devices.

External merge sort has been evaluated on embedded devices using replacement selection for run generation in FSort [10]. The average run size was $2M$. MONTRES [17] was shown to outperform external merge sort, FSort, MinSort, and FAST when sorting large data sets on SSDs. MONTRES was not evaluated on embedded devices as the memory usage for its block index is too high to be stored on these devices.

Figure 5 contains I/O performance formulas for the various external sorting algorithms. Overall, there is a fundamental need for sorting for data processing, and for algorithms that reduce memory usage and offer stable performance.

Algorithm	Reads (Gen)	Writes (Gen)	Reads (Merge)	Writes (Merge)
external merge sort	$N * R$	$N * W$	$N * \lceil log_{M-1}(\lceil \frac{N}{M} \rceil) \rceil * R$	$N * \lceil log_{M-1}(\lceil \frac{N}{M} \rceil) \rceil * W$
FSort	$N * R$	$N * W$	$N * \lceil log_{M-1}(\lceil \frac{N}{2M} \rceil) \rceil * R$	$N * \lceil log_{M-1}(\lceil \frac{N}{2M} \rceil) \rceil * W$
MinSort	$N * (1 + D) * R$	$N * W$	0	0
FAST	$N * \frac{N}{M} * R$	$N * W$	0	0
FAST(N)	$N * \frac{Q}{M} * R$	$N * W$	$N * \lceil log_{M-1}(\lceil \frac{N}{Q} \rceil) \rceil * R$	$N * \lceil log_{M-1}(\lceil \frac{N}{Q} \rceil) \rceil * W$
MONTRES	$(2 * N + P_R) * R$	$(N + P_W) * W$	$(N - P_W - G + \alpha) * R$	$(N - P_W - G) * W$
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)	$N * R$	$N * W$	$N * \lceil log_M(\lceil \frac{N}{2M} \rceil) \rceil * R$	$N * \lceil log_M(\lceil \frac{N}{2M} \rceil) \rceil * W$

Fig. 5. Existing Sort Algorithms I/O Performance Formulas

3 NO OUTPUT BUFFER SORT ALGORITHM

No output buffer sort (NOBsort) eliminates the output buffer used by external merge sort. This reduces the memory used and the number of merge passes, especially for small values of M. The number of merge passes is log_M instead of log_{M-1}. For small embedded devices with limited memory, this has a significant effect on performance, as every merge pass requires reading and writing the input data and many merge passes are needed for sorting larger data. Even more important, by eliminating the output buffer the minimum memory required is only two buffers (1 KB for 512 byte buffers). This allows external sorting to be practical on even the smallest devices where it was not executable before.

The run generation phase may either use load-sort-store or replacement selection. The run merge phase is in Figure 6. The key insight is that all M buffers can be used during the merge process if buffer index 0 is used both as an input and an output buffer. The process starts by reading the first block of M runs into the M buffers. The smallest record in the M buffers is found and moved to the output block. Unlike traditional merge sort where the output block is dedicated and there is space available for the output record, in this case the smallest record is swapped with a record in the output block. The record in the output block is placed at the top of the block with the smallest record. If there are multiple records that were originally in the output block in another block, they are organized as a heap for efficiency. As the algorithm proceeds on each step to find the next smallest record, it must examine the current smallest records in the input buffers as well as the smallest record in the heap of every non-output block. When all input records in the current block have been processed, the next block of that run is read. This also requires special handling as it may be necessary to swap records currently in the block to another block to allow for an empty buffer to read into. If the next input block is being read from buffer 0, this will require swapping any current output records to other blocks and then swapping them back into the output block after the input block has been read. Overall, the algorithm will perform fewer I/Os as it can merge more runs per pass but will have higher number of comparisons and in-memory record copies than external merge sort. Since reducing the number of I/Os and merge passes is critical for performance, NOBsort outperforms external merge sort while using less memory.

3.1 Algorithm Description

The following is a detailed description of the pseudocode algorithm in Figure 6. Input to the algorithm is the number of buffers M and the number of runs produced during the run generation phase. The implementation has all runs stored in a contiguous address range (i.e. a single file) which allows it to be used both for embedded systems that have file system support and for raw memory chips. Memory or file offsets are passed in to the algorithm to know the starting and ending ranges of the runs.

```
1  while numRuns > 1
2      Initialize by reading first block of up to M runs
3      Buffer 0 is output buffer and also stores block of one of the M runs
4
5      while still records to process
6          Find smallest record and block that it is in
7
8          if smallest record not in output block
9              if no space in output block
10                 Swap smallest record with record currently in output block
11                 Update heap
12             else
13                 Move record to output block and update heap
14
15         if output block is full, write to storage
16
17         if record containing smallest block has no more records and there is another block in the run
18             if block is not the output block
19                 Move any records originally from output block into other blocks
20                 Read next block in the run into buffer
21             else
22                 Swap output records into other blocks temporarily
23                 Read next run block into output block buffer
24                 Swap output records back into output block
25
26     Write any remaining output records to output
27
28     numRuns = numRuns - M + 1
29     Every third pass wrap around in memory space/file
```

Fig. 6. NOBSort Run Merge Implementation

For initializing merging, the algorithm must determine the offset of the first block in each run being merged in this pass, and read each one of those blocks (lines 2-3). This turns out to be an interesting implementation challenge. If the runs are generated using load-sort-store, then this is trivial as all runs (except perhaps the last one) have the same size, and the start of any given run can be directly calculated based on its index number and the size of the runs during this pass. With replacement selection, the runs can be variable size, so the starting offset of any run cannot be directly calculated. An obvious solution is to store the starting offset of each run in memory, which works fine for a small number of runs. However, by 500 runs at 4 bytes per offset, that is now consuming about 2 KB, which is more than all of the rest of the algorithm and is not practical. Note that storing the runs in separate files does not solve this problem as file identifiers would also need to be stored and take up valuable memory.

The solution implemented to handle variable length runs is to read the runs in reverse order they were produced. By reading the last block of the run and using the block index in its header, it is possible to calculate the offset of the first block of the run. This technique requires one extra block read every time a run is processed, but requires no additional memory.

The number of in-memory comparisons and record copies is minimized if the output buffer 0 contains the smallest input value present in any of the M blocks buffered into memory. When the data is already in sorted order, run 0 will have all the smallest values and will be processed completely first before any of the other runs. The result is the algorithm has the same number of comparisons and swaps as standard external merge sort while also reducing the number of I/Os.

The while loop starting on line 5 will repeatedly find the smallest record and block that it is in and copy/swap that record into the output buffer. If the smallest record is already in the output buffer, it may be exactly at the current output record location or need to be copied from its current location in the block to the output location. If the record was not in the output block, then the record is copied from its current block to the output (line 13) if there is space for it or swapped with an input record in the output block (lines 9-11) if there is not. In the case of a swap, the record that was originally in block 0 is put in the heap at the top of the smallest block (line 11). Once the record movement was complete, the output pointer and current input pointer for the smallest record block are updated.

Special handling is required whenever the next block for a run must be read. There are two cases: reading the next block from the run buffered in buffer 0 and reading a block from any other run. When reading a block that is not from run 0, then there may be records from run 0 in a heap at the top of the block that must be moved to other buffers (preferably back to buffer 0) before reading (lines 18-20). When reading the next block from run 0, any output records currently buffered in buffer 0 must be swapped to other buffers temporarily. After the block from run 0 is read, these output records are swapped back into buffer 0 (lines 22-24).

Every iteration through the outer while loop will merge M runs, so the number of runs is reduced by $M - 1$. On every third pass, the memory space/file used for storing temporary results can be wrapped around to the start of the memory/file. The implementation is wrapping every third pass as the runs are being merged from the end not the start. If all runs are the same size so that the technique used to determine the start of the runs can allow merging from the start of the runs, then every second pass can start at the beginning of the memory space. Being able to re-use memory or file space is important for increased performance as many passes and intermediate runs may be generated.

An original implementation of NOBsort was presented in [16]. That implementation only handled fixed-size runs generation by load-sort-store, did not support reading input records using an iterator, and used insertion sort to maintain the run 0 records in other buffers. Using a heap minimizes the number of comparisons and record copy operations. This is especially important when the number of records per block is large.

3.2 Example

An example execution with $M = 2$ is in Fig. 7. Buffer 0 is both the output buffer and input buffer for run 0. **C** is the current record pointer in the buffer. **O** is the location of the last record from buffer 0 that was moved to the buffer. Any records moved from buffer 0 to another buffer are stored in a heap with the smallest record in the first record location. Records originally from run 0 are shown in italics. Records in bold and italics are the current sorted output in the output buffer. At step #5, the output buffer is full and written to storage. There are still records for run 0 that are currently in buffer 1, so the next input block from run 0 is not read. (Note: An optimization is that the block for run 0 can be read immediately as its maximum value remaining (6) is smaller than the other buffer value (7). This is not shown in order to illustrate the case where swapping records from the output buffer is required.) At step #7, the records from run 0 are exhausted, so the next block from run 0 must be read. This requires swapping the output records temporarily into buffer 1 and reading the block (step #8), then swapping the records back (step #9). At step #11, the output block is written. The block for run 1 is exhausted, so the records from run 0 are swapped back to buffer 0 before the next input block for run 1 is read. The example demonstrates merging using only two buffers, but the technique generalizes to any number M buffers.

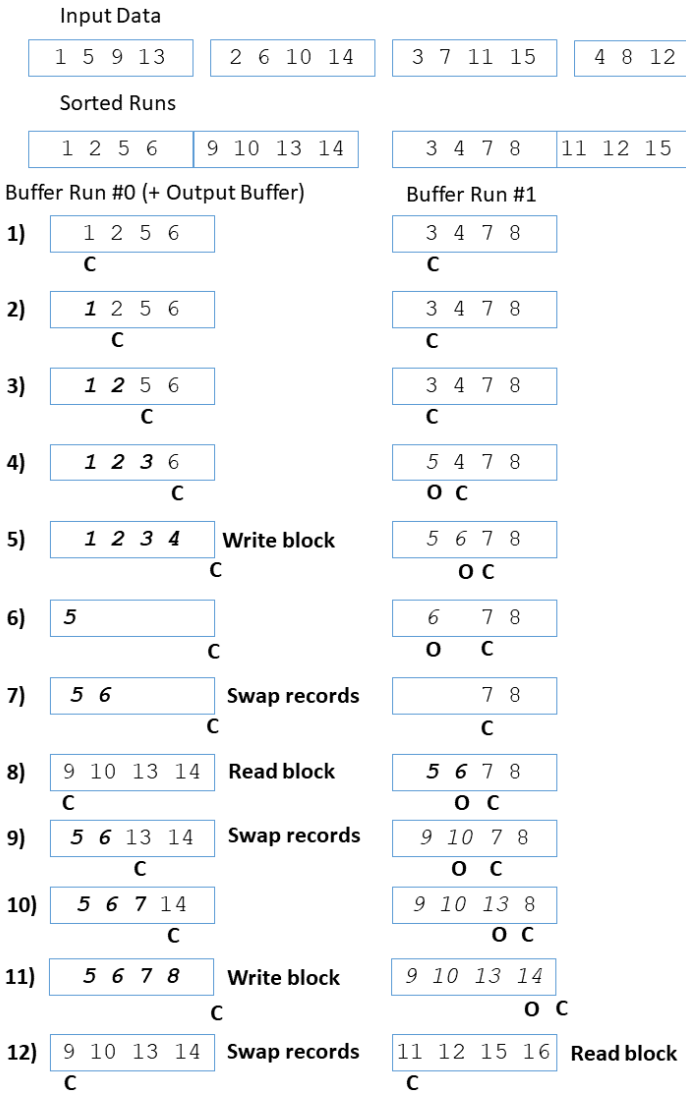

Fig. 7. Example for M=2

3.3 Replacement Selection

Replacement selection for run generation has the potential benefit of reducing the number of runs and merge passes for data that is partially sorted. Implementing replacement selection on embedded devices has its own set of challenges. First, reading and writing is at the block-level so when removing items from the heap, a block of records must be removed and written out not just one record at a time. Second, runs will be of variable sizes in terms of number of blocks as well as number of records in the last block. This requires special handling during the merge phase. The algorithm for replacement selection run generation is in Figure 8.

```
Fill M-1 blocks in memory with tuples from input
Build heap of M-1 input blocks
Initialize lastOutputKey = null

while more input blocks
    read next input block b
    sort block b
    for each record r in b
        let heapKey = heap.top().key
        if r.key < lastOutputKey and heapKey < lastOutputKey
            close previous run
            create new run
            set lastOutputKey = null
            resort block b
            restart record iterator of b at first record
            continue
        else if heapKey >= lastOutputKey and (heapKey < r.key or r.key < lastOutputKey)
            swap heap record with record r in b
            reheapify
       else
            leave r in current location in b
   write to current run block b
if heap is not empty
    close previous run
    create new run
    sort records in heap in memory
    output sorted records as last run
```

Fig. 8. Replacement Selection Run Generation Algorithm (M > 2)

The algorithm works by first loading $M - 1$ input blocks and building a heap from the records. Then, the next input block is loaded into a buffer in memory. This block of records is sorted. A merge process is performed comparing the records in the block with the records in the heap. By the end of the merge process, the smallest records are in the block and output to the run. The process then repeats with the next input block. A new run is started when there are no records in memory with a key larger than the last key output to the current run.

Variable-sized runs generated by replacement selection complicate the merge process. When all runs are fixed size, the starting location (offset) of each run can be calculated based on the run index. For variable-sized runs, it is necessary to store for each run its size in blocks and its starting location. Within a block, a block header is required to store the number of records within the block and the block index within the run.

For M=2, an optimization is possible for replacement selection. Since there is only two memory blocks available, one will be used as input and the other as output. The block index that is output switches every iteration. Instead of using a heap, both blocks are sorted and an in-place merge is performed. This technique reduces the number of comparisons and in-memory swaps, especially for near-sorted data.

Prior work [10] has shown for random data runs are twice as large as M, which reduces the number of runs by a factor of 2. For sorted or near sorted data, the number of runs can be significantly reduced.

4 THEORETICAL ANALYSIS

The number of I/Os saved using NOBsort compared to external merge sort is approximated by $(log(M) - log(M - 1))/log(M)$ and shown in Fig. 9. The savings is substantial for small values of M but decreases quickly. Since the number of merge passes is computed by $\lceil log_{M-1}(L) \rceil$ (for external merge sort) and $\lceil log_M(L) \rceil$ (for NOBsort), in practice the

savings is a step-wise function that also depends on the number of initial runs L. This results in potentially more or less I/O savings depending on the effect of the ceiling function. Since the focus for embedded sort is for small values of M, NOBsort has significant I/O improvements.

2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
∞	37%	21%	14%	10%	8%	6%	5%	5%

Fig. 9. NOBsort Theoretical Performance Improvement by M

NOBsort using replacement selection reduces the number of runs for random data by a factor of two similar to the results shown in FSort [10]. FSort always performs at least as many I/Os as NOBsort using replacement selection. Thus, NOBsort will always have the same or fewer I/Os than external merge sort or FSort with only a small increase in CPU load.

MONTRES [17] is not directly comparable to NOBsort on small embedded devices as MONTRES requires too much memory to implement its optimizations. Similar to the approach used in MinSort, MONTRES builds a block index in memory that contains the minimum value in each block. If the sort key is a 32-bit integer, each index record will consume 8 bytes (4 for the key and 4 for the block number/offset). The block index consumes 1 KB of RAM for every 128 blocks. Sorting 4096 blocks would require 32 KB of RAM, which substantially exceeds the RAM specifications of the target devices.

MinSort introduced the block indexing concept used in MONTRES for sorting on embedded devices. It is able to work in low memory environments by shrinking the index so not every block is indexed, but rather adjacent blocks are grouped into regions and indexed. Although this adapts to the low memory constraints, performance suffers due to numerous reads per input block. The performance formula depends on the number of distinct values D in the data. If there is enough memory to index every block, then given typical block sizes of 512 bytes and record sizes of about 16 bytes, the number of records per block $B = 32$ and $D <= 32$. Often there is not enough memory to index every block, and adjacent blocks are grouped into regions. Thus, performance depends on the number of distinct values per region, D_R, which can be substantially higher than the maximum number of records per block B. In the worst case, the sort key is unique, and D_R is equal to the number of records in the region.

As discussed in [12], MinSort is faster than two-pass external merge sort when the number of distinct values per region is less than the write-to-read cost ratio, $D_R < \frac{W}{R} + 1$. Generalizing to multiple merge passes S, $D_R < S(\frac{W}{R} + 1)$. NOBsort reduces the number of passes S compared to external merge sort as it is log_M rather than log_{M-1}.

There are cases where MinSort is faster than NOBsort especially with low D and a large write-to-read ratio. The write-to-read ratio is hardware dependent. On the hardware MinSort was tested on, this ratio was 1.8, and for the SD cards tested in this work the ratio was 1.67. Overall, neither algorithm is faster in all cases. NOBsort has improved on external merge sort by reducing the minimum memory to $M = 2$ and the number of merge passes. Experimental results in the next section illustrate the performance of the algorithms.

5 EXPERIMENTAL EVALUATION

The experimental evaluation compared NOBsort with external merge sort and MinSort. FAST was not compared as prior work [11] demonstrated that MinSort had better performance than FAST. MONTRES [17] could not be evaluated on the embedded hardware used as its algorithm required building a block index of input blocks. As discussed in the background section, this block index consumes 1 KB of RAM for every 128 blocks, which would be over 64 KB

for the data sizes tested and greatly exceed available RAM on the device. It may be interesting to evaluate NOBsort versus MONTRES on SSDs. However, NOBsort is focused exclusively on small memory embedded cases, so such experimentation is left to future work.

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 2 GB SanDisk 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 must be 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 code and device functions. Note that M is bounded by the hardware memory size, and that the sorting algorithm typically does not have all this memory available as memory is also used for other functions on the embedded device.

Benchmark reading and writing tests on the SD card show sequential read performance of 408 blocks/sec. (204 KB/s) and sequential write performance of 245 blocks/sec. (123 KB/s). Random read performance was 400 blocks/sec. (200 KB/s) and random write performance was 100 blocks/sec. (50 KB/s). On this platform, sequential writes are about 66% slower with a write-to-read ratio of 1.67. Unlike raw flash chips, the write performance issues on the SD card are minimized by its FTL.

Both merge sort and NOBsort have the same reading and writing characteristics. The run generation step performs sequential reads and sequential writes. The run merge step performs random reads and sequential writes. MinSort performs sequential reads for the first pass then performs random reads.

The experimental data set was chosen to be representative of the real-world data sets evaluated in MinSort [11]. 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 [11] were collected from soil moisture sensors used for automatic irrigation and consisted of 16 bytes with a 2 byte integer key.

The record size selected was 16 bytes with a 4 byte integer key. A key size of 4 bytes allows for a larger range of keys to be sorted. 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. For most of the experiments, the record keys were generated randomly in the 32-bit key space so the number of duplicate values were small. The I/O performance of NOBsort and external merge sort are not affected by the key distribution, the number of distinct values D, or the number of records per block B. The experiments vary both the number of memory buffers M as well as the input size in blocks N.

For experiments with MinSort that is affected by D, the number of distinct values D was varied during performance testing. When evaluating the effectiveness of replacement selection that is also impacted by the data distribution, the input data distributions were varied as was the amount of the input data that was already sorted. The results are the average of five runs.

5.1 NOBsort versus External Merge Sort

External merge sort and no output buffer sort were compared for memory sizes M of 2, 3, and 4 on records with randomly generated keys for increasing input data sizes N. Fig. 10 shows the sort time, and Fig. 11 shows the number of I/Os performed. Only NOBsort is executable for $M = 2$, and its performance is almost identical to external merge sort with $M = 3$ (one more memory buffer). When both algorithms are given the same amount of memory, the performance of NOBsort is superior and very close to the theoretical I/O improvement of 37% for $M = 3$ and 21% for $M = 4$. The performance of NOBsort versus external merge sort increases as the size of the input increases. This makes sense as larger inputs require more merge passes and I/Os.

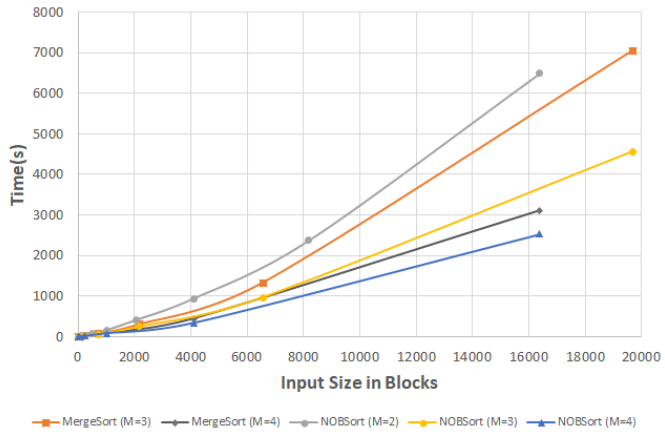

Fig. 10. Sorting Performance by Time (s)

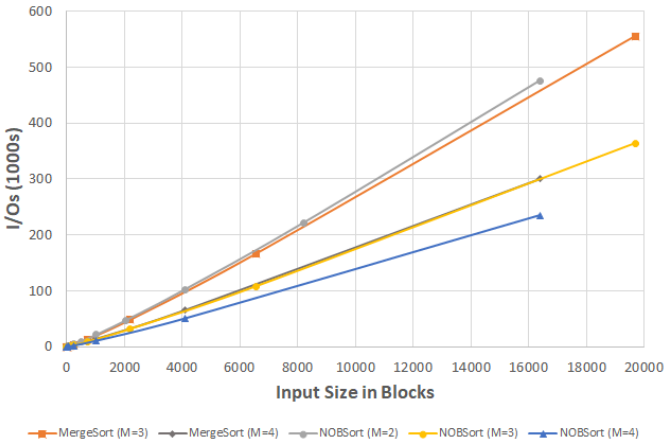

Fig. 11. Sorting Performance by I/Os

5.1.1 NOBsort versus External Merge Sort for Larger Memory Sizes. Given that NOBsort reduces memory usage by increasing in-memory swaps and comparisons, it is natural to question when using external merge sort is preferable. Theoretically, NOBsort will always have the same or less I/Os, but that advantage decreases rapidly as M increases. Experiments were performed to determine the crossover point where NOBsort is no longer faster than external merge sort. This crossover point is hardware and platform specific. It also depends on the data input size as the number of passes is computed by the ceiling function of $\lceil log_M(L) \rceil$ (NOBsort) versus $\lceil log_{M-1}(L) \rceil$ (external merge). There may not be a difference in the number of passes (i.e. I/Os) between the two approaches for certain values of L.

Fig. 12 displays the time improvement for NOBsort compared to external merge sort for increasing values of M for sorting 16,000 blocks. There is no clear crossover point on the experimental platform, but by $M = 6$ the I/O advantage varies between 7 and 15% (depending on L) resulting in small time differences. $M = 8$ represents over 50% of the available memory on the test device and is a practical limit for memory available for sorting.

In summary, NOBsort has similar or better performance than external merge sort even for $M = 6$ and above, but the advantage drops considerably. That is acceptable as minimizing memory is critical, and using the algorithm in the range of $M = 2$ to $M = 4$ is desirable. Saving I/Os has the added benefit of reducing wear on flash memory in the embedded devices, as fewer writes are performed.

The experiments were also performed on a PC for validation on a different platform. By $M = 5$, the time between the algorithms were essentially identical even though the I/O savings for NOBsort was about 10%. These results demonstrate that NOBsort is superior to external merge sort for typical low memory configurations on embedded devices.

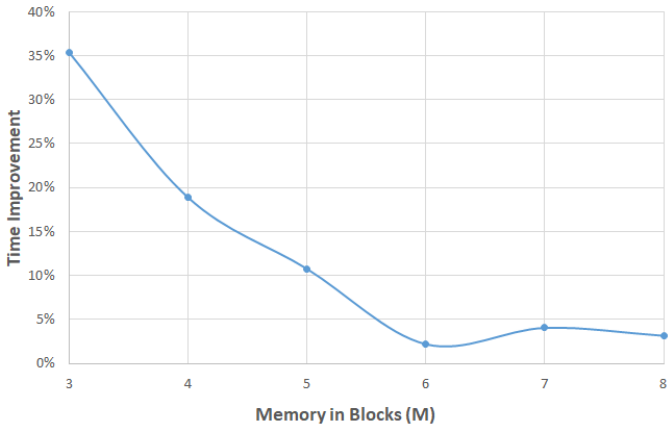

Fig. 12. Percentage Time Improvement for NOBsort versus External Merge Sort for Increasing M

5.2 NOBsort Optimizations

The performance of the NOBsort implementation in [16] was compared with the current version that uses heaps and a variety of new optimizations. The algorithms have numerous implementation differences, so it is not a perfect comparison. For instance, the original implementation did not support different run sizes, variable number of records per block, or block headers. However, the comparison was mostly focused on the choice of using a heap versus insertion sort for record management. Both implementations were adapted to use the file wrap around technique, which has a major performance benefit. The file size grows large with multiple passes, and random I/O in a large file is not handled well by the FAT16 file system [22]. When a file system is available, the wraparound technique works well and only requires one file to be used. This technique also works when no file system is available and direct memory addressing is used.

In Fig. 13 is a summary of key performance characteristics including time, I/Os, comparisons, and record copies for random data. These results show that the new heap-based version results in a significant reduction in record copies with other metrics having a minor improvement. I/Os are slightly higher in the new version as the algorithm performs more of these operations to identify the start of each run when handling variable-sized runs. The time performance is slightly improved but within experimental variation. A key improvement is for sorted or near sorted data where the ability to use replacement selection and minimizing record copies is substantial. The performance difference is especially notable for the smallest memory case, $M = 2$. In summary, using a heap is theoretically superior, but the benefits have a minor improvement in overall algorithm performance for random data as the algorithm performance

is dominated by I/Os. The advantage of using heaps becomes more significant when record sizes are smaller (more records fit in a block resulting in larger heap sizes).

% Change Heap vs Original Implementation				
Memory	**Time**	**I/O**	**Compares**	**Copies**
M=2	-3%	4%	-26%	-51%
M=3	0%	3%	-3%	-39%
M=4	-6%	2%	-2%	-33%
M=5	-8%	2%	-2%	-28%

Fig. 13. Performance Difference between Heap-based Implementation and Original Implementation of NOBsort (random data)

5.3 NOBsort versus MinSort

MinSort performs no writing and uses multiple random read passes to sort the data. Unlike NOBsort and external merge sort, its I/O performance depends directly on the data distribution, specifically the number of distinct values per region, D_R. A region is a group of adjacent blocks. MinSort uses 4 bytes of memory (the size of the sort key) for each region to build a region index.

For the experiments, when the algorithm is given M blocks of memory, it uses those blocks to store the region index. One of the M blocks must be used as an input buffer as the SD card has no hardware buffers to use. Since the algorithm produces sorted output one record at a time, an output buffer is not allocated from the M blocks in memory, but if the sorted result was required to be saved back to flash, an output buffer would also be required.

The data set used for prior experiments had almost all unique sort keys, which is the worst case for MinSort. In that case, the number of distinct records in a region is equal to the number of records in the region. Even if a region is one block, then $D_R = B$, and each block is read D_R times. The performance in that case is very poor.

Data sets were generated varying D from 4 to 1024. The results are in Figure 14. The figure displays the time ratio of MinSort divided by NOBsort time. Numbers higher than 1 indicate how much slower MinSort is than NOBsort, and numbers less than 1 occur when MinSort is faster than NOBsort.

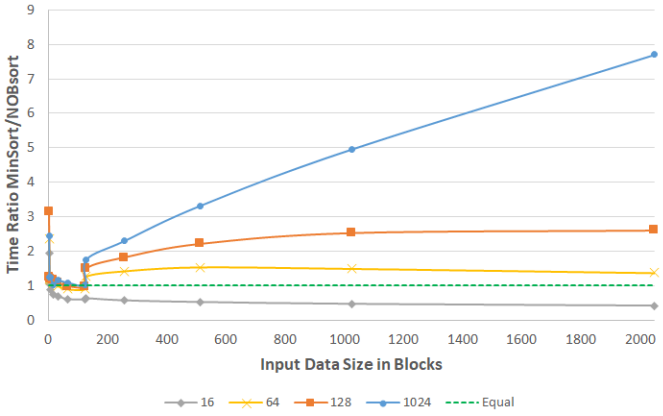

Fig. 14. Time Ratio for MinSort/NOBsort for M = 2

The results directly align with the theoretical I/O analysis presented earlier. For all values of D, for very small number of input blocks (e.g. 10 or less), NOBsort is faster because few passes are required for sorting. As the input size increases, the number of passes increases and very quickly MinSort improves its relative performance. A cross-over point occurs when $N = 125$ as at that point MinSort must conserve memory by indexing a region consisting of 2 blocks rather than individual blocks. Depending on the value of D, NOBsort becomes more competitive. NOBsort with replacement selection was also evaluated, and the performance was similar as when using load-sort-store. It is important to emphasize that NOBsort performance is consistent and stable for all values of D. The number of I/Os are always the same regardless of D, and the time variation is minimal. In comparison, the performance of MinSort is completely dependent on D.

For this experimental hardware, NOBsort is superior when $N >= 128$ for values of $D >= 64$. These results demonstrate that neither algorithm dominates the other in all cases. If the value of D was known before sorting starts, then it would be straightforward to select the best algorithm to use. In practice, the value of D is not known until the input is read the first time. Further, there are many instances where D is small in sensor networks, but there are also cases where the sort key is unique (i.e. D is equal to number of records to sort). In that case, MinSort has very poor performance and must be avoided.

5.4 Adapting to Data Distributions

NOBsort has superior performance over other algorithms in almost all cases except when the number of distinct sort key values D is small. In that case, MinSort, which performs only reads, has better performance. This section explores optimizations for NOBsort that dynamically support different data distributions.

The essence of the approach is to switch between the merge based approach of NOBsort and the indexing approach of MinSort after D is known. Theoretically, MinSort is faster than NOBsort when $D_R < S(\frac{W}{R} + 1)$, where D_R is the number of distinct values per region and S is the number of merge passes. The block write (W) and read (R) performance is known or can be determined by testing for a given memory hardware. The number of merge passes S depends on the number of blocks to sort in the input. The number of distinct values D requires reading or sampling the input to estimate. The number of distinct values per region, D_R, is approximated by D, and is bounded by the number of records in a given region. Recall that a region may consist of multiple adjacent blocks.

In some cases, D may be known if statistics are collected by the system. In that case, it is possible to decide on using MinSort or NOBsort directly using the performance formula. If the input is available for reading or sampling, D can also be estimated. In these cases, there are no changes to either NOBsort or MinSort. The appropriate algorithm is used based on the values of D, S, R, and W.

Consider the case where D and N are not known before the algorithm starts. Figure 15 contains an algorithm optimization for NOBsort that starts with reading the input in the first pass to estimate D and builds the minimum value index as used in MinSort. If the performance formula indicates that proceeding with MinSort is superior, the algorithm stays with using MinSort. Otherwise, the first read pass is discarded, and the algorithm proceeds with using NOBsort and building sorted runs. In the best case, this algorithm has the same superior performance as MinSort for small values of D. In the worst case, only one additional read pass is required over the data, and the performance is the same as NOBsort. This optimization combines the stability and overall performance of NOBsort with the MinSort optimization for small values of D. The number of distinct values is determined using cardinality estimation techniques such as Bloom filters [15].

```
Read the input data and generate MinSort minimum value index

Estimate number of distinct values (D) while reading data

if D < S*(1+W/R)
    continue with MinSort
else
    discard minimum value index
    switch to NOBsort to generate sorted runs and complete sort
```

Fig. 15. NOBsort Implementation Optimizing for Different Data Distributions

Experiments were conducted comparing the optimized NOBsort version for different data distributions with NOBsort and MinSort. The results in Figure 16 show the performance of the optimized NOBsort that adapts to data distributions versus the base NOBsort algorithm. The ratio shown is for various values of D and takes the time for the base NOBsort algorithm and divides it by the optimized version. A ratio above 1 indicates that the base version is slower than the optimized version.

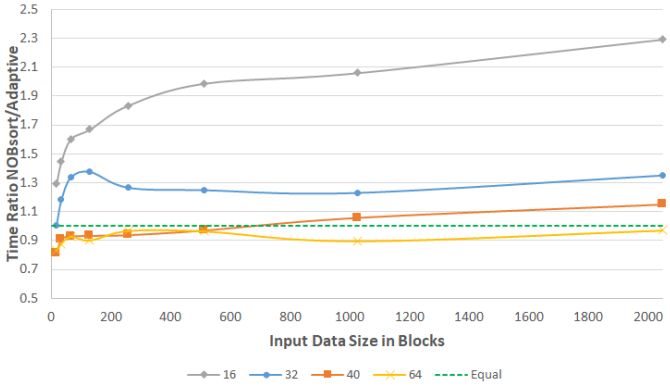

Fig. 16. NOBsort Optimization to Adapt to Data Distributions

The results demonstrate that the NOBsort algorithm adapts to changing data distributions, retains all the benefits of MinSort for small D, and has minimal overhead when required to switch to NOBsort for larger D. On this experimental platform, for values of $D < 40$, the adaptive version uses MinSort and has identical performance to the MinSort algorithm. This gives a significant performance advantage over the base NOBsort algorithm that did not adapt and used multiple merge passes. For $D = 16$, the optimized approach is twice as fast for large data sets. At approximately $D = 40$ is the point where NOBsort and MinSort have similar performance. The results for $D = 40$ show how the algorithm adapts to the changing input size. For small input sizes less than 1000 blocks, using NOBsort is preferable as there are fewer merge passes. For inputs larger than 1000 blocks, using MinSort is prefered. This explains why the performance is originally about 5% lower for the adaptive version (due to read overhead then switching to NOBsort) but then becomes about 10% better for larger input sizes. As D increases beyond 40, the algorithm always uses NOBsort after the first read pass. The only overhead was reading the data once to estimate D, which is about 5% of the I/O and time for larger data sets. This overhead is minimal especially as the size of the input increases and more merge passes are required. The optimized NOBsort avoids using MinSort for large values of D where MinSort's performance is very poor. Overall, the

optimized NOBsort version that adapts to different data distributions is superior to MinSort and has benefits compared to the original NOBsort algorithm.

5.5 Replacement Selection

Replacement selection was evaluated versus load-sort-store for data of varying characteristics. The data tested included sorted data, reverse sorted data, and sorted data where random values replaced original sorted data at given probabilities. For example, sorted data with 10% random data has 10% of the data file with random data and the other 90% is in sorted order.

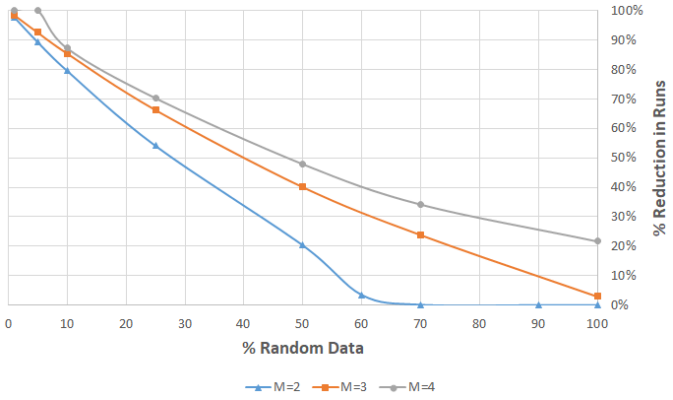

Fig. 17. Replacement Selection Run Reduction for Various M

Fig. 17 shows the percentage reduction in the number of runs when using replacement selection versus load-sort-store with $N = 10000$ blocks and various values of M. In the best case of sorted data, the number of runs is only one for replacement selection, and in the worst case of reverse sorted data, the two techniques have the same number of runs. As the percentage of random data increases, the reduction in the number of runs decreases (equivalently the length of each run decreases). There is clearly an advantage of using replacement selection for reducing the number of runs, especially for nearly sorted data. Even completely random data has some reduction in the number of runs.

The overall time and I/O performance improvements are not nearly as significant, as reducing the number of runs L is only a linear factor in the formula $log_M(L)$. Fig. 18 contains the time savings. The time savings directly correspond to the I/O savings. For $M = 2$, a 50% reduction in the number of runs is required to save one merge pass. By $M = 4$, the number of runs has to be reduced by a factor of 4 to save one merge pass. For example, in the $M = 3$ case, there is no I/O difference between random 5% and 10% data and the random 25% and 50% data. Despite differing number of runs, the same number of merge passes are required (6 and 7 respectively). There are minor time fluctuations (shown by slight bumps in graph) but that is within experimental variation.

By using a large $N = 10000$ blocks, the I/O and time savings are not as significant as multiple merge passes are required. There would be a higher percentage reduction when sorting smaller data sets as the reduction in runs would be a larger factor of the overall performance.

For this test case with at least 10% random data, the run generation time represents between 3 and 10% of the overall execution time, a comparably small amount.

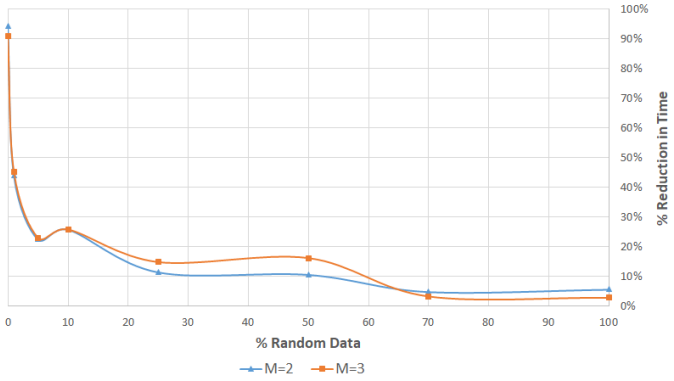

Fig. 18. Replacement Selection Time Reduction for Various M

Interestingly, even for reversed ordered data, which is the worst case for replacement selection, the performance was within a few percent of using load-store. In this case, the number of runs in both approaches is the same, and there is no I/O difference. Even with completely random data, replacement selection reduces the number of runs, especially as M increases. Through all experiments, there was no configuration that replacement selection did not have equivalent or better performance than load-store, and in many cases the performance improvement is very high. Consequently, using replacement selection is recommended for run generation.

5.6 Results Discussion

The experiments compared no output buffer sort with external merge sort and MinSort for many variations of input data size, memory data size, and key distributions.

Overall, no output buffer sort demonstrates consistent I/O improvement over external merge sort in all cases. Theoretically, NOBsort will always perform the same or fewer I/Os, and any additional CPU and memory operations executed have minimal impact. When executed on the smallest devices (with small M), the performance improvement is between 20 and 35%. Most importantly, the algorithm can execute with $M = 2$, which reduces by 33% the minimum amount of memory required. NOBsort should be used over external merge sort in all cases. The experiments demonstrate that the additional comparisons used by NOBsort compared to external merge sort result in an insignificant time increase, and the overall time performance of NOBsort is superior.

Using replacement selection compared to load-sort-store is recommended for embedded devices. With sorted or partially sorted data, the performance improvement is significant. Replacement selection reduces the number of runs (even for random data) that often reduces the number of merge passes that must be performed. Since sensor data is often closer to sorted rather than random data, using replacement selection is the preferred choice.

In comparison to MinSort, there are cases where MinSort outperforms NOBsort. First, for the smallest memory sizes $M < 2$, only MinSort works. MinSort's performance varies widely based on the data distribution and the number of distinct values D. For low values of D, MinSort will perform fewer I/Os as it must scan the blocks only D times. However, as D increases, its performance degrades rapidly. NOBsort has consistent performance for all data distributions. The variability of MinSort is both a benefit and an issue. If it is known the data distribution has low key variability, then it

has advantages. However, the downside to performance is considerable if that is not the case. The data distribution may only be known after the data is read the first time.

Optimizations to extend NOBsort to handle data distributions for low values of D were experimentally validated. By dynamically switching between MinSort and NOBsort based on a known estimate of D or determining D during the first read pass of the data, the optimized NOBsort algorithm captures the benefits of MinSort for small D while preserving stable performance for other data distributions. For many use cases, the data distribution and sort size would be known or easily estimated, which allows for selecting the optimal algorithm to use.

The experimental results were further validated using a second SD card for storage. The results produced on this alternate SD card were consistent with the results presented. The performance differences between the algorithms directly follow the theoretical I/O performance formulas, and variations in storage hardware performance do not change these I/O differences.

For small embedded devices, NOBsort allows for efficient sorting while consuming a small amount of memory. The experiments were performed on a device with 8 KB SRAM, and NOBsort used between 1 KB and 2 KB of SRAM. This allows sorting to be used concurrently with data collection with good performance.

6 CONCLUSIONS AND FUTURE WORK

No output buffer sort reduces the memory consumption of external merge sort on small memory embedded devices. This allows external sorting to be practical on smaller devices. For small memory sizes, NOBsort reduces I/Os by 20 to 35% and execution times by similar amounts. This work also demonstrated the advantage of using replacement selection for sorting sensor data, which has significant performance improvement for data that is partially sorted. NOBsort is a drop-in replacement and improvement over external merge sort in these use cases. Future work will perform further experiments on other embedded hardware platforms and flash memory configurations and integrate the sorting algorithm into a database library.

7 ACKNOWLEDGMENT

The authors would like to thank NSERC for supporting this research.

REFERENCES

[1] 2020. Arduino Homepage. http://arduino.cc
[2] 2020. Microchip AT89LP3240 Specifications. https://www.microchip.com/wwwproducts/en/AT89LP3240
[3] 2020. Microchip ATmega2560 Specifications. https://www.microchip.com/wwwproducts/en/ATmega2560
[4] 2020. Microchip ATSamd21 Specifications. https://www.microchip.com/wwwproducts/en/ATsamd21g18
[5] 2020. Microchip ATtiny1634 Specifications. https://www.microchip.com/wwwproducts/en/ATtiny1634
[6] 2020. Microchip ATtiny48 Specifications. https://www.microchip.com/wwwproducts/en/ATtiny48
[7] 2020. Microchip PIC Specifications. https://www.microchip.com/wwwproducts/en/PIC18F57Q43
[8] 2020. NXP LPC8N04 Specifications. https://www.nxp.com/docs/en/data-sheet/LPC8N04.pdf
[9] 2020. Texas Instruments MSP430FR6007 Specifications. https://www.ti.com/product/MSP430FR6007
[10] 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.
[11] 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
[12] Tyler Cossentine and Ramon Lawrence. 2013. Efficient External Sorting on Flash Memory Embedded Devices. *International Journal of Database Management Systems* 5 (02 2013), 1–20. https://doi.org/10.5121/ijdms.2013.5101
[13] 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

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

[15] 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

[16] 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

[17] A. Laga, J. Boukhobza, F. Singhoff, and M. Koskas. 2017. MONTRES: Merge ON-the-Run External Sorting Algorithm for Large Data Volumes on SSD Based Storage Systems. *IEEE Trans. Comput.* 66, 10 (Oct 2017), 1689–1702. https://doi.org/10.1109/TC.2017.2706678

[18] J. Lee, H. Roh, and S. Park. 2016. External Mergesort for Flash-Based Solid State Drives. *IEEE Trans. Comput.* 65, 5 (May 2016), 1518–1527. https://doi.org/10.1109/TC.2015.2451631

[19] Yang Liu, Zhen He, Yi-Ping Phoebe Chen, and Thi Nguyen. 2011. External Sorting on Flash Memory Via Natural Page Run Generation. *Comput. J.* 54, 11 (2011), 1882–1990. https://doi.org/10.1093/comjnl/bxr051

[20] 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), 122âĂŞ173. https://doi.org/10.1145/1061318.1061322

[21] 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

[22] W. Penson, S. Fazackerley, and R. Lawrence. 2016. TEFS: A flash file system for use on memory constrained devices. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. 1–5. https://doi.org/10.1109/CCECE.2016.7726822

[23] Chin-Hsien Wu and Kuo-Yi Huang. 2015. Data Sorting in Flash Memory. *Trans. Storage* 11, 2, Article 7 (March 2015), 25 pages. https://doi.org/10.1145/2665067