Faster Sorting for Flash Memory Embedded Devices

by James (Riley) Jackson

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

B.A. COMPUTER SCIENCE HONOURS

in

Irving K. Barber School of Arts and Sciences

(Computer Science)

Supervisor: Dr. Ramon Lawrence

THE UNIVERSITY OF BRITISH COLUMBIA
(Okanagan)

April 2019

Abstract

Abstract—Embedded devices collect and process data in a wide variety of applications including consumer and personal electronics, healthcare, environmental sensors, and Internet of Things (IoT) deployments. Processing data on the device rather than sending it over the network for analysis is often faster, more energy efficient, and supports decision-making closer to data collection. A fundamental data manipulation operation is sorting. Sorting on embedded devices with flash memory is especially challenging due to the very low memory and CPU resources. Previous work developed customized algorithms that avoided writes and minimized memory usage. The standard external merge sort algorithm has limited application on small devices as it requires a minimum of three memory buffers and is not flash-aware. The contribution of this work is an extension of external merge sort that requires only two memory buffers and is optimized for flash memory. The result is an algorithm that improves on the state-of-the-art and applies to a wider range of devices. Experimental results demonstrate that when sorting large data sets with small memory the algorithm reduces I/Os and execution time by about 30%.

Index Terms—sorting, Arduino, embedded, performance, Internet of Things

Contents

List of Figures

1 Introduction

Sorting is required for data processing tasks including aggregate calculations, joins, and result ordering. Performing sorting on devices improves performance, reduces network transmissions, and is more energy efficient, allowing devices to operate longer under battery power. Main-memory sorting algorithms are insufficient for embedded devices that typically have small RAM (2 to 128 KB) but large flash storage (MBs or GBs). External sorting algorithms such as external merge sort [1] are required, but these algorithms were developed for servers with different resource and performance features.

External merge sort has been adapted for use with flash memory and solid state drives (SSDs) with specific focus on servers. Algorithms such as [6] and MONTRES [8] use various optimizations such as sort run lengthening, block value indexing, and dynamic merge on-the-fly to increase performance. The general techniques of these algorithms are beneficial but cannot always be directly adapted to the embedded context due to high memory usage.

Previous research has also developed sorting algorithms specifically for embedded devices such as FAST [3] and MinSort [4]. These algorithms increase sorting performance by using more reads rather than writes due to the asymmetric costs of reading and writing in flash memory. They also adapt to the lower memory environment. However, performance may be reduced due to the increased number of reads.

1.1 Motivation and Contribution

The contribution of this work is an optimized external merge sort for sorting with minimal memory. Specifically, no prior work has supported a minimum of two memory buffers by eliminating the output buffer during merging. For devices with minimal memory (i.e. a few KBs), reducing the memory usage is a critical factor. Requiring fewer buffers during merging decreases the number of merge passes, which reduces I/Os, especially costly writes. Another optimization is that only a single continuous memory area is used for sorting which makes the algorithm easily adaptable to raw flash chips where no flash translation layer or file system is available. Performance results show that the number of I/Os and time can be reduced by about 30% when sorting large data sets with small memory.

2 Background

Sorting algorithms have been extensively researched for database operations [1] as they are fundamental for data processing involving ordering, joins, and aggregation. The standard external merge sort algorithm works in two phases. Assume M blocks are available in memory. The first phase is the *run generation phase* that reads chunks of 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*. 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, the number of merge passes S is $\lceil log_{M-1}(\lceil \frac{N}{M} \rceil) \rceil$. 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).

Various optimizations [1] have been performed on external merge sort such as run generation with replacement selection, double buffering, and parallelization during merging. [7] defines a natural page run to be a sequence of blocks whose values do not overlap but are not necessarily sorted. During run generation, these natural page runs are detected and indexed but not sorted. Natural page runs are sorted during the merge phase which reduces I/Os during run generation.

2.1 Sorting on Flash Memory

Storage devices based on flash memory such as solid-state drives (SSDs) have two important characteristics. First, the cost of writing can be multiple times larger than the cost of reading. Second, writing data in the same location requires an expensive erase operation, so it is often more efficient to write in a different location rather than overwriting the same location. In SSDs and SD cards, a flash translation layer (FTL) handles the mapping of logical addresses to physical addresses in order to provide wear-leveling across the device and maintain performance.

External sorting optimizations for flash memory fall into two common approaches. The first is to reduce the number of write operations performed during the run generation phase. This can be achieved by reading the input multiple times [3] or using random reads to search for minimum values [4, 5]. The second technique is to optimize the run merge phase by indexing the

data and using random reads to retrieve tuples in minimum order [7].

MONTRES [8] is a sorting algorithm designed for SSDs that uses three optimization techniques during run generation: ascending block selection (using a minimum value index), continuous run expansion to generate larger runs, and merge on-the-fly to reduce the number of values merged. It was shown to improve on external merge sort in cases when the input size is a large multiple of the memory size. During the run merge phase, MONTRES proceeds in a single pass by using an index that stores the minimum value of each block in every run in order to determine the next block to read. MONTRES assumes all blocks can be indexed in memory which consumes too much memory for embedded devices. Flash-specific sorting was also developed in [6] which used a decision rule to determine when to use clustered (sequential-based) or unclustered (index-based) sorting.

2.2 Sorting on Embedded Devices

Embedded devices are characterized by limited memory and CPU resources and data storage on flash memory such as SD cards. Increasingly, embedded devices are performing more substantial data processing rather than just data collection and transmission. A particular target device for this research is the Arduino Uno [9] that uses a 8-bit, 16 MHz microcontroller and has 2 KB of SRAM. The Arduino was designed to be an easily programmable prototyping tool for students, however it has since become a popular and inexpensive option for rapid prototyping and sensor deployment in a variety of fields.

Embedded external sorting algorithms avoid writes extensively for increased performance. FSort [2] uses replacement selection during the run generation phase to increase the average size of runs to $2 * M$, which reduces the number of runs. Flash MinSort [4] uses memory to build an index that stores the minimum value in each region. A region may contain one or more adjacent data blocks. The algorithm uses the index to determine the next smallest value, reads only the region containing this value, and then outputs the record. This process repeats until the data is sorted. Random reads are used, and the algorithm does not perform writes as it combines run generation and merging into a single phase. FAST [3] also performs in one phase and scans the input file several times. Each time it retrieves and outputs the next smallest m values where m is the number of records that can fit in memory. FAST performs up to M/N scans on the input to save N write

operations. This algorithm is generalized to process larger files by using runs generated by FAST as initial runs for the external merge sort algorithm. Experimental results [4] show than MinSort is significantly faster than FAST [3] when sorting data sets on embedded devices with small memory.

No prior algorithm considered reducing the minimal memory usage of external merge sort to make it more competitive on small embedded devices.

3 External Merge Sort Implementation

A custom, open source implementation of external merge sort was created for this project. The custom implementation uses less memory and only requires one file to be open at a time. Another advantage is that this implementation can handle different record and key sizes as well as comparison functions.

3.1 Standard Merge Sort Implementation

External merge sort creates the initial sorted runs and must be able to perform recursive merging (see Fig. 1). Several implementations of external merge sort were examined on GitHub. Most of these implementations depended on storing runs in different files. Creating multiple files and having them open is not ideal for use on the Arduino. An open file may require several variables to be stored in memory. Variables may include information about the file's directory and storage, cursor position, cursor block position and file size.

3.2 Implementation

An implementation that is not dependent on creating and opening several files was created for this project. Runs are written to the end of the original file instead of to their own files as shown in Fig. 2. The locations of runs are calculated using file offsets instead of using separate files, which saves memory.

3.2.1 Required Merge Variables

The start (read) position and end (write) position of the last merge are stored. At the very start of the algorithm, the read location will point to the start

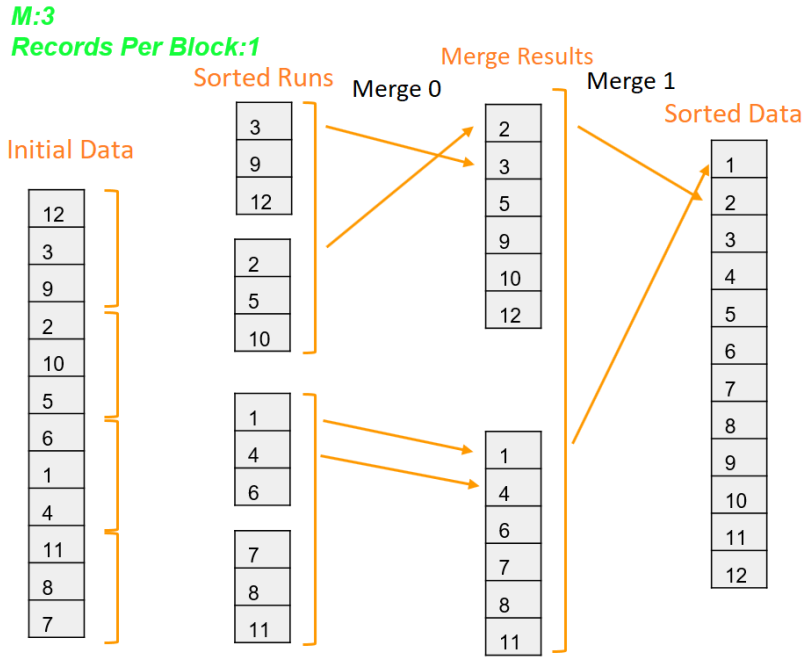

Figure 1: Data written by external merge sort where data is shown as blocks containing numbers.

of data and the write location to the end. The first stage of the algorithm creates several runs of size M. After this stage of the algorithm the read position will be at the start of the first run and the write position will be at the end of the last run. The subsequent merges will update the read and write position in the same manner. This allows merges to locate runs from the last merge for reading, merges always write to the end of the last merge which means they usually write to the end of the file.

The number of remaining runs, number of blocks per run and some information about the runs needs to be stored. Two arrays are allocated to the size of $(M-1) * sizeof(int)$, each array has one space of memory available for each run being merged. One array stores the current block of the run that records are being taken from. The other array stores the position of the record that is being examined in the current block of that run. This information is required to iterate through the runs.

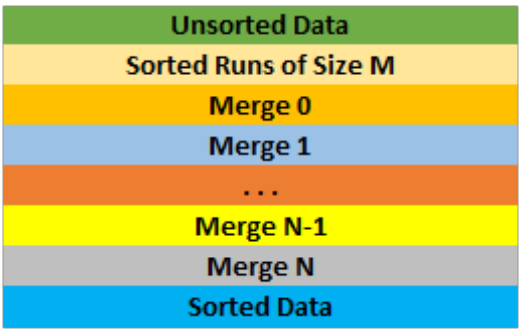

Figure 2: Ordering of the contents of the file when requiring N merge passes

3.2.2 Calculating Run Locations

Run size is counted in blocks, all runs in a merge are normally the same size. Merges will reduce the number of runs as well as increase the size of runs in a predictable manner.

$$runsInNextMerge = \frac{runsInMerge}{M-1}$$

Runs are created by combining M-1 runs so their size after a merge is described by the following formula.

$$runSizeNextMerge = runSizeThisMerge * (M-1)$$

Merges continuously combine M-1 groups of runs until there is no runs left to merge. The file position in bytes of the run at the start of the next group to merge is determined with the following formula.

$$groupFilePos = readPos + groupNum * (M-1) * runSize * blockSize$$

where blockSize = size of a block in bytes,
groupNum = number of groups merged so far,
readPos = file position of the start of the last merge,
runSize = number of blocks per run for this merge

The location of each run in a group denoted as i, where $i = 0..groupSize-1$, is determined with the following formula

$$runFilePos[i] = groupFilePos + i * runSize * blockSize$$

Note that these calculations assume that all initial runs (except for the last one) are the same size.

3.2.3 Buffers

Each block stored in the buffer is assigned a single counter to keep track of the current record. New records are read in when the counter goes beyond the number of records that fit in a block. The output blocks counter counts how many sorted records it contains and is used to determine the position to place the next sorted record. Records are copied into the output block, blocks that belong to a depleted run are ignored Fig 3.

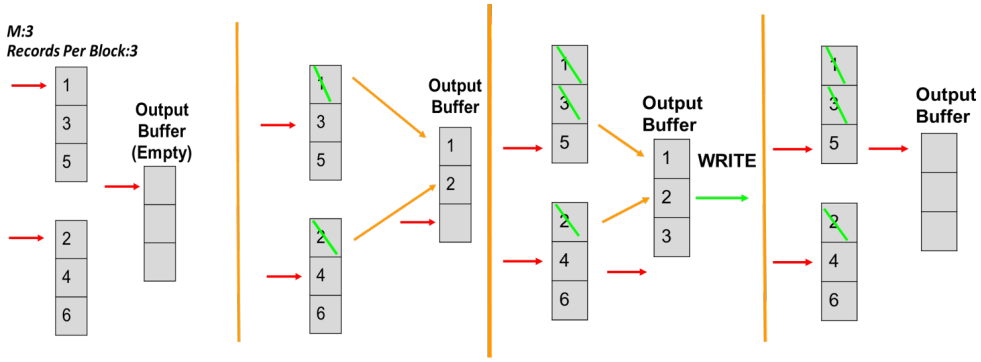

Figure 3: Buffer of external merge sort when using M=3.

3.2.4 Handling Undersized Runs

In some cases there will be less than $M - 1$ runs left during the final run merge pass of a merge phase. This merge will create an undersized run which causes problems because the merge sort implementation assumes all runs are the same size, which saves memory. Saving the location and size of all undersized runs and then merging them at the end of the algorithm would require more memory. Continuously writing the undersized run to the end of future merges, merging it with any other undersized runs, results in unnecessary reads and writes when the future merges do not produce an undersized run.

Undersized runs were handled by only allowing one undersized run to exist, and only writing it when necessary. Limiting the number of undersized runs to one means that the size and file location of only one undersized run ever has to be stored. The undersized run will be combined with any other undersized runs that are created which can save reads and writes. An example is shown in Fig 4.

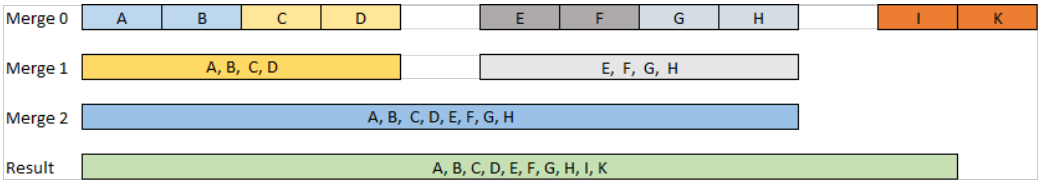

Figure 4: Handling an undersized block created by a merge. The undersized block is combined at merge 2 saving a read and a write at merge 1

3.2.5 Handling Partially Filled Blocks

This external merge sort implementation assumes that all blocks are filled except for the last one. The number of records in the last block is passed as a parameter if the block isn't full. When a merge reads from the last block of its last run it will check if that blocks record counter is equal to the number of records in the last block. If the record counter is equal to the number of records in the last block the last run is considered depleted and the partially filled block is written out.

4 Sorting Algorithm

4.1 Introduction

The no output buffer external merge sort works by eliminating the output buffer normally used. For small values of M, this can have a dramatic impact on performance as the number of merge passes (and consequently reads and writes) is now log_M instead of log_{M-1}. It also allows external merge sort to be used with as little as two buffers (1 KB) which makes it feasible for very small devices. The trade-off is that more comparisons and movement of records within the buffers must be performed as well as careful handling when buffering a new input block.

Pseudocode for the run generation phase is in Fig. 5 and the run merge phase is in Fig. 6 and Fig. 7. The run generation algorithm uses standard load-sort-store to generate runs. This phase sorts M blocks at a time to produce sorted runs. Replacement selection was considered, but has challenges for small memory sizes, as a dedicated input and output buffer is required during run generation. This is not acceptable for $M < 5$ and eliminates any opportunity for generating runs larger than M. Further, runs are now different sizes which requires maintaining in memory run starting offsets and lengths. Future work may modify replacement selection to handle very small M.

```
dataFile <- file containing data to sort
buffer <- capable of holding M blocks
numRuns = 0

// Create initial sorted runs of size M
while dataFile.hasRecords
   run <- read next M blocks of dataFile
   sort(run)
   dataFile.append(run)
   numRuns++
end
runSize = M
```

Figure 5: Run Generation Code

4.2 Buffer Management

To enable merging using all M buffers, the algorithm uses a pointer in each buffer ($bufCurrPtr$) to track the current record in each run. Buffer index 0 is selected to be the output buffer as that results in the smallest amount of data movement for sorted or near-sorted data. Note that buffer index 0 stores both the output and records from run 0.

The next smallest record to output is determined by finding the minimum current record in each buffer. The minimum record is then swapped with the current record in the output buffer. Since these records must be retained, each buffer also maintains a count ($bufOut$) of the records that were transferred from the output buffer to this buffer. When determining the next smallest record, it is required to look at both the current records

```
runStartOffset <- get_start_of_runs(dataFile) // Determine start of runs
bufCurrPtr <- int[M]    // Current record pointer in each buffer (run)
bufOut <- int[M]    // Position of last output buffer (run 0) record in this buffer
runOffsetPtr <- int[M]  // Offset in file for next block to read from run

while numRuns > 1
   numOutputRuns = ceiling(numRuns / M)
   for run=0; run < numOutputRuns; run++
      // Read block from each run and initialize pointers
      for i=0; i < M; i++
         runOffsetPtr[i] <- runStartOffset + i*runSize
         buffer[i] <- read_block(dataFile, runOffsetPtr[i]);
         bufCurrPtr[i] <- buffer[i] // Position of smallest record in each block
         // Position of last output record block in this block
         bufOut[i] <- EMPTY
      end

      while still records to process (either in buffer or on storage)
         smallRecordPtr <- get_smallest_record(buffer, bufCurrPtr, bufOut)
         smallBlock <- get_block(smallRecordPtr)

         // First buffer (index 0) is used as output buffer
         if smallRecordPtr == bufCurrPtr[0]
            // Smallest record is in buffer 0.
            if bufCurrPtr[0] != bufOutPtr[0]
                //Empty space in middle of buffer 0, need to hop record over it
                copy_record(bufCurrPtr[0], bufOutPtr[0])
            //else No movement necessary
            bufCurrPtr[0] += recordSize
         else if smallRecordPtr is an bufOut record pointer
            // Copying a record originally in buffer 0 back to output buffer
            // Smallest record is always first record in block
            if bufCurrPtr[0] != EMPTY
               swap_records(buffer[smallBlock], bufCurrPtr[0])
               // Swapped record may not be in order. Use insert sort.
               insertSort(buffer[smallBlock], bufOut[smallBlock])
               bufCurrPtr[0] += recordSize
            else
               // No record to swap with. Just copy over.
               copyRecord(buffer[smallBlock], bufOut[0])
            end if
         else
            if bufCurrPtr[0] != EMPTY
                // Swapping an record in buffer 0 with a record in another buffer
               swap_records(bufCurrPtr[smallBlock], bufCurrPtr[0])
               bufOut[smallBlock] += recordSize
            else
               copy_record(bufCurrPtr[smallBlock], bufCurrPtr[0])
            end if
            bufCurrPtr[0] += recordSize
            bufCurrPtr[smallBlock] += recordSize
         end if
```


Figure 6: Run Merge Code Part 1

```
        // For buffer 0, bufOut[0] stores offset to write next output record
        bufOut[0] += recordSize
        // Write full output buffer block
        if (bufOut[0] == FULL)
           write(buffer[0], dataFile)

        // Determine if a new block must be read in from buffer
        if bufCurrPtr[smallBlock] == EMPTY && smallBlock != 0
           while (bufOut[smallBlock] != EMPTY)
               // Move any records from run 0 in this block to others
              destBlock <- find_block_with_space_other_than(smallBlock)
              put_value_into_block(smallBlock, destBlock)
              bufOut[destBlock] += recordSize
           end
           bufOut[smallBlock] = EMPTY
           runOffsetPtr[i] += block_size
           if runOffsetPtr[i] != EMPTY
               buffer[smallBlock] <- read(dataFile, runOffsetPtr[i])
           else
               buffer[smallBlock] <- EMPTY
           end if
        end if
        if bufCurrPtr[0] == EMPTY && (bufOut[i] == EMPTY for i=1..M ||
            max(bufOut[i])<min(bufCurrPtr[i]))

           // Block 0 is empty and bufOut is empty OR
           //max value for run 0 < than min in other runs
           swap result records into other blocks temporarily
           read new run block into output block buffer
           swap result records back into output block
        end if
     end
  end
  numRuns <- numOutputRuns
  runSize <- runSize * M
end
```

Figure 7: Run Merge Code Part 2

$bufCurrPtr$ and the first record in each buffer when $bufOut > 0$. When the $bufCurrPtr$ for a buffer is exhausted (past end of buffer), then the next block of the run must be read into the buffer. If there are records in the buffer from the output buffer ($bufOut > 0$), then those records must be transferred to another buffer (see Fig 8).

4.3 Reading Into Buffer 0

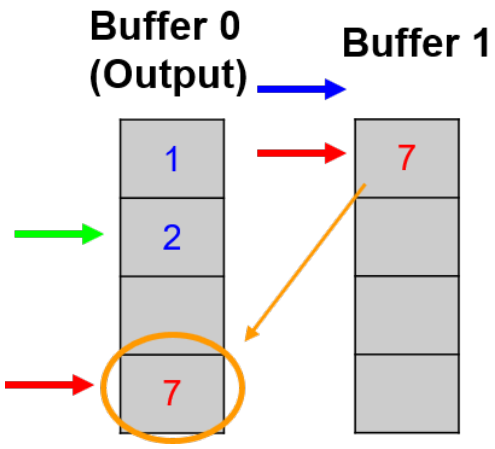

Figure 8: Buffer 0 record being put back into buffer 0 from buffer 1. Buffer 1 has no records left so it has to be emptied and then new buffer 1 records are read in.

The most complex case is reading the next block from the run that is in buffer 0 (the output buffer). In that case, records currently in the output buffer are transferred to one or more other buffers temporarily. Then, the next block from the run is read. Records in the output are swapped back into the output buffer and then the algorithm continues (see Fig 9).

4.4 Reads and Writes

After every second merge pass is completed, the next writes can occur at the start of the file (memory space) again. Thus, the algorithm requires at least the input size of space in secondary storage to function. This is a common requirement for external merge sort.

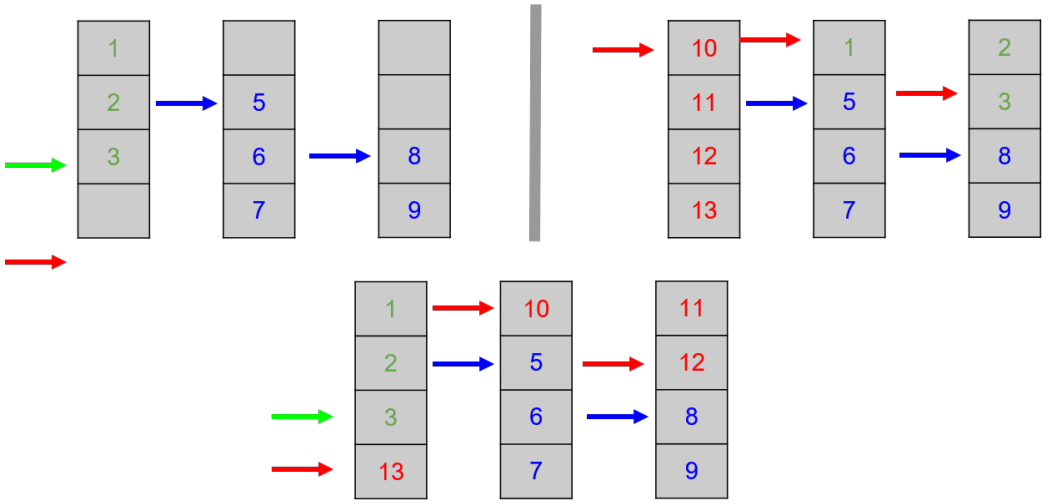

Figure 9: Three part diagram showing a read into the output buffer. Top left: the output block at position 0 is out of records (red numbers) but still contains results (green numbers). Top Right: Results are placed into buffers 1 and 2 and buffer 0 records are read in. Bottom: Results are swapped back into their original positions. The read is now done.

In Fig. 10 is an example execution for $M = 2$. Buffer 0 is used for buffering run 0 as well as an output buffer. **C** represents a current record pointer in the buffer. **O** is location of the last record from buffer 0 that was moved to the buffer. Note that the smallest such record is always in the first record index, and these records are maintained in ascending order. These records are also 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. The next block for run #0 can be read immediately as its maximum value left (6) is smaller than the other buffer value (7). After step #10, the first block for run #1 has been exhausted. Before the next block can be read in, the records (pointed to by **O**) originally in buffer #0 are swapped back and the current pointer is updated. Then the next block for run 1 is read in and the process continues. With this technique it is possible to merge with only two buffers. The technique generalizes to any number M buffers.

In Fig. 12 is an example execution for $M = 3$. More comparisons and in memory swaps are required as M increases. In step two, the three records at

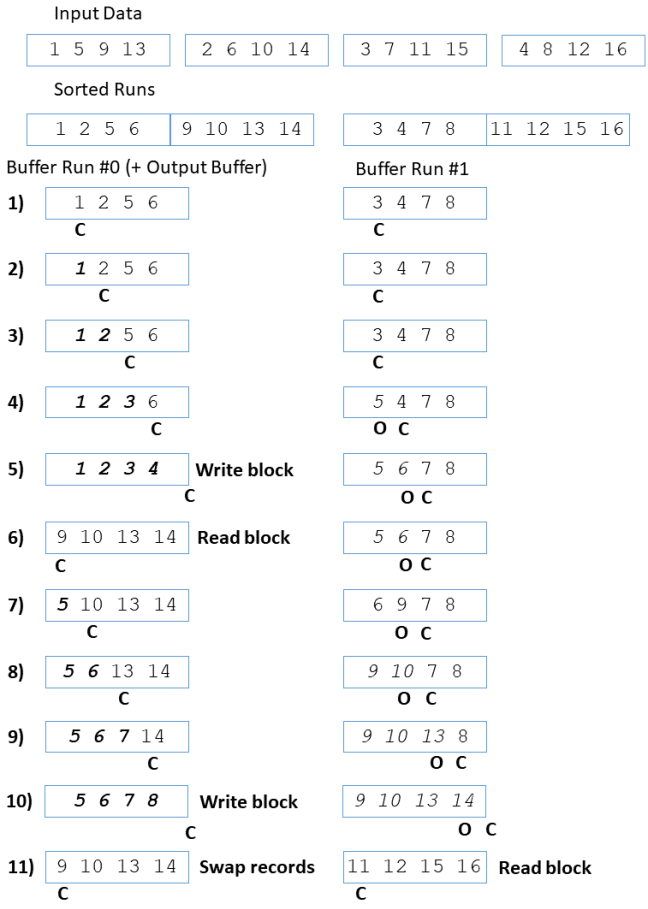

Figure 10: Example for M=2

the C counters and two records at the O counters must all be compared to find the the smallest record. There are several edge cases that occur when $M > 2$. At step 6 buffer 0 is emptied with a write and then the 15 value in buffer 1 is put back into buffer 0 before the next block is read into buffer 0. In step 8 there is a gap between the R and O counters. The buffer 0 value of 15 would have to be copied to the next R position if buffer 2 was not about to be read in. Having to copy the buffer 0 value differs from most cases where incrementing the R counter would be sufficient, as the record is already in the correct location. At the end of step 9 the 16 value is put back into buffer 0, this shifts the 15 value down and the R counter is incremented without requiring the 15 value to be copied. This technique can be further generalized to any greater number of M buffers.

The percentage reduction in I/Os is given by the formula $(log(M) - log(M-1))/log(M)$. Fig. 11 shows that this is significant for small values of M but decreases rapidly. Note that in practice there may be some deviation

as the number of merge passes is computed by $\lceil log_{M-1}(\lceil \frac{N}{M} \rceil) \rceil$, and the ceiling function may cause an extra pass in certain cases.

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

Figure 11: Theoretical Performance Improvement by M

5 Experimental Results

The experimental evaluation was conducted on an Arduino MEGA 2560 [9] 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. A 2 GB SanDisk microSD card was attached with an Arduino Ethernet shield. 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). Although for raw flash chips write performance is signficantly slower than read performance, the FTL on the SD card compensates for this and writes are only 66% slower. The results are the average of several runs. The page size is 512 bytes. The record size is 16 bytes with a 4 byte integer key. Records are generated randomly.

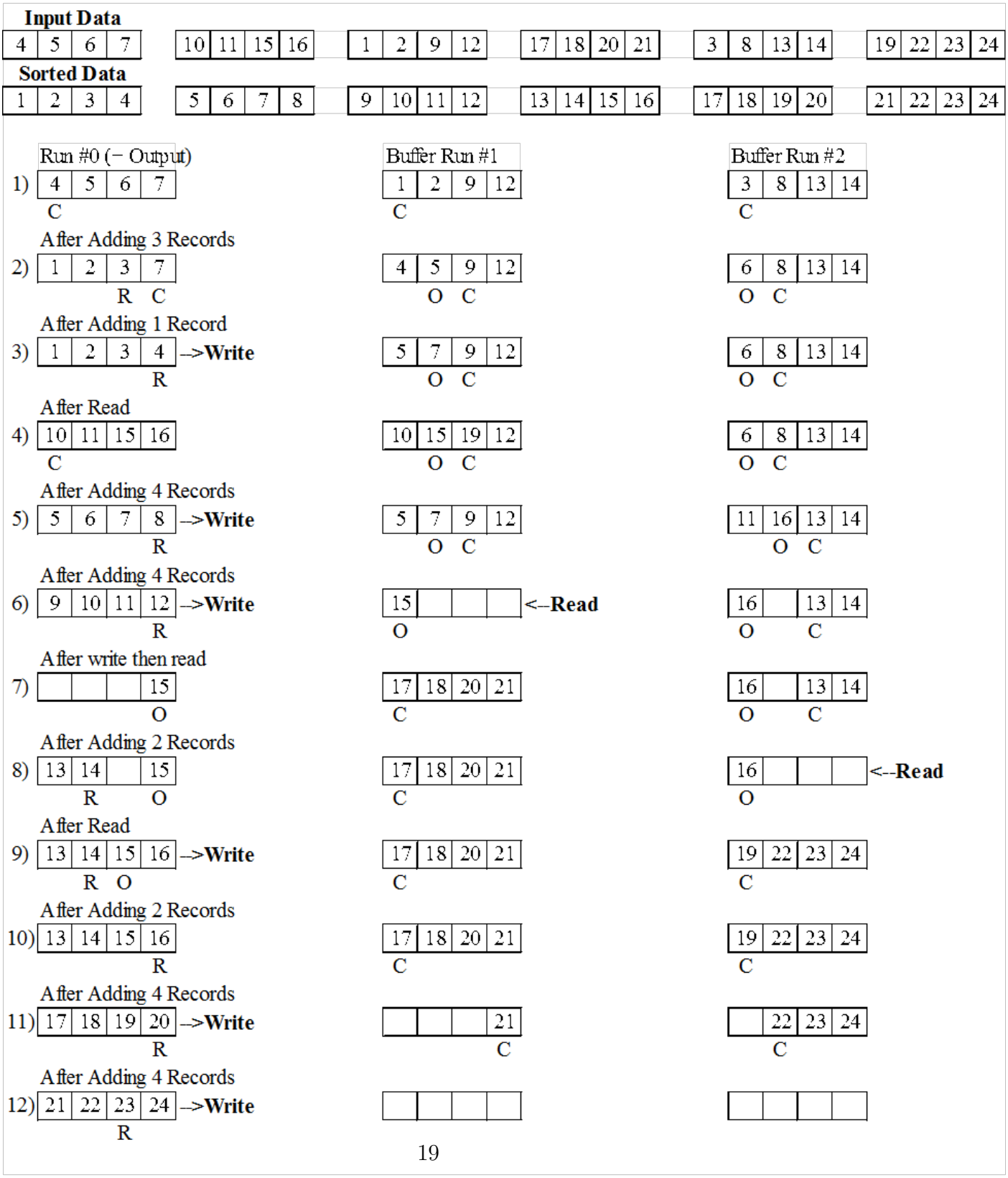

Figure 12: Example for M=3.

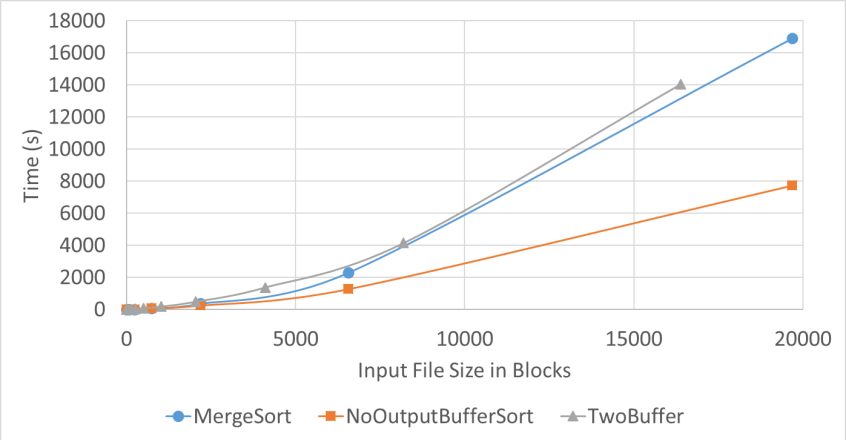

Figure 13: Sorting Performance by Time (s)

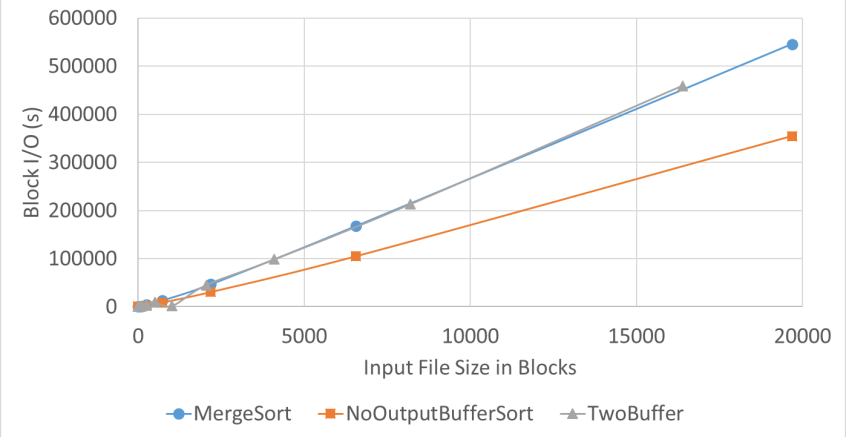

Figure 14: Sorting Performance by I/Os

The standard external merge sort was compared with the optimized version that uses no output buffer. The sort time in seconds (Fig. 13) and number of I/Os (Fig. 14) were captured.

The results show merge sort without an output buffer for the $M = 3$ and $M = 2$ cases. For $M = 3$, the theoretic I/O improvement is 37% and that is seen in the results. The time improvement is close but not quite the same due to CPU and memory transfer overhead. Further, the $M = 2$ case has performance characteristics almost identical to the regular

external merge sort with $M = 3$. Thus, it has all the same performance with 33% less memory usage. Performance was also compared with MinSort with $M = 1586$ bytes given to MinSort. For small data sizes up to $N = 128$ blocks, MinSort had comparable performance. By time $N = 1024$, MinSort was over 9 times slower and the performance difference was increasing. This makes sense as MinSort was designed for non-random data and for very small memory.

Using a a heap instead of shifting the values did not greatly alter the speed for $M = 1024$ bytes and $M = 1536$ bytes, as shown in Fig. 15 and Fig. 16. This was not consistent with the prediction that using a heap should reduce the number of memory transfers and improve performance.

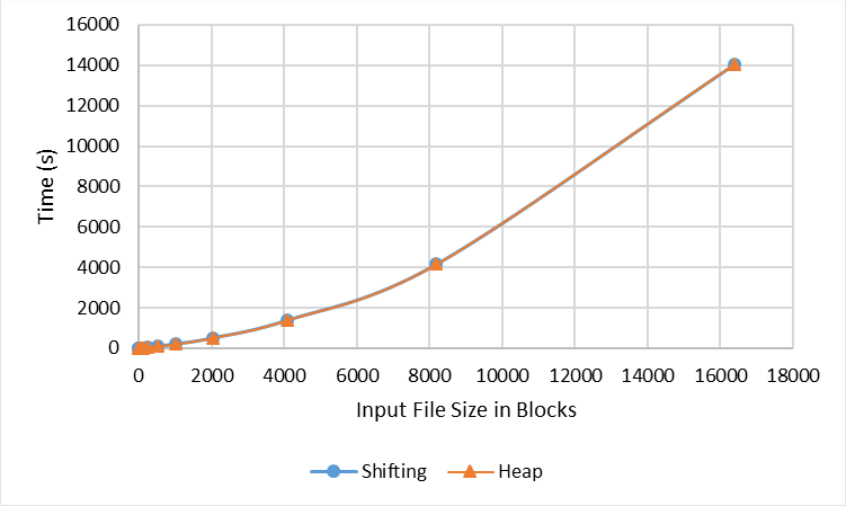

Figure 15: Sorting Performance by Time (s)

6 Conclusions and Future Work

External sorting on embedded devices with small memory is challenging. This work presented an optimization of external merge sort to only require two buffers during merging and uses all M buffers. This improves performance for small memory cases and makes sorting more practical. Future work will investigate if any run generation optimization is feasible and examine how to combine the indexing technique used in MinSort [4] and MONTRES [8] with the external merge algorithm to achieve even better performance.

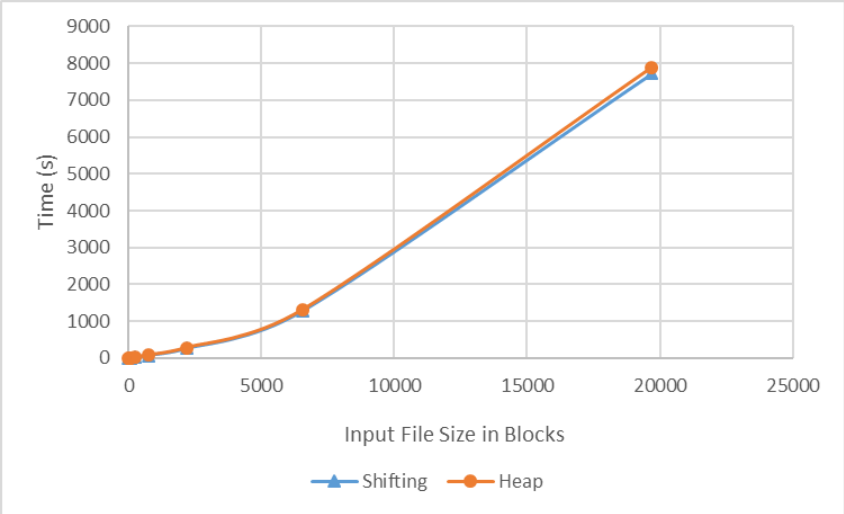

Figure 16: Sorting Performance by Time (s)

References

[1] Goetz Graefe. "Implementing Sorting in Database Systems". In: *ACM Comput. Surv.* 38.3 (Sept. 2006). ISSN: 0360-0300. DOI: 10.1145/1132960.1132964. URL: http://doi.acm.org/10.1145/1132960.1132964.

[2] Panayiotis Andreou et al. "FSort: external sorting on flash-based sensor devices". In: *DMSN'09: Data Management for Sensor Networks*. 2009, pp. 1–6. ISBN: 978-1-60558-777-6.

[3] Hyoungmin Park and Kyuseok Shim. "FAST: Flash-aware external sorting for mobile database systems". In: *Journal of Systems and Software* 82.8 (2009), pp. 1298–1312. ISSN: 0164-1212. DOI: DOI:10.1016/j.jss.2009.02.028.

[4] Tyler Cossentine and Ramon Lawrence. "Fast Sorting on Flash Memory Sensor Nodes". In: *Proceedings of the Fourteenth International Database Engineering and Applications Symposium*. IDEAS '10. Montreal, Quebec, Canada: ACM, 2010, pp. 105–113. ISBN: 978-1-60558-900-8. DOI: 10.1145/1866480.1866496. URL: http://doi.acm.org/10.1145/1866480.1866496.

[5] Yang Liu et al. “External Sorting on Flash Memory Via Natural Page Run Generation”. In: *The Computer Journal* 54.11 (2011), pp. 1882–1990. DOI: 10.1093/comjnl/bxr051.

[6] Chin-Hsien Wu and Kuo-Yi Huang. “Data Sorting in Flash Memory”. In: *Trans. Storage* 11.2 (Mar. 2015), 7:1–7:25. ISSN: 1553-3077. DOI: 10.1145/2665067. URL: http://doi.acm.org/10.1145/2665067.

[7] J. Lee, H. Roh, and S. Park. “External Mergesort for Flash-Based Solid State Drives”. In: *IEEE Transactions on Computers* 65.5 (May 2016), pp. 1518–1527. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2451631.

[8] A. Laga et al. “MONTRES: Merge ON-the-Run External Sorting Algorithm for Large Data Volumes on SSD Based Storage Systems”. In: *IEEE Transactions on Computers* 66.10 (Oct. 2017), pp. 1689–1702. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2706678.

[9] *Arduino Homepage*. URL: http://arduino.cc.