Faster Sorting for Flash Memory Embedded Devices

Riley Jackson, Ramon Lawrence
Department of Computer Science
University of British Columbia - Okanagan Campus

***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**

I. 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 significantly higher resources and performance.

External merge sort has been adapted for use with flash memory and solid state drives (SSDs) with specific focus on servers. Algorithms such as [2] and MONTRES [3] 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 [4] and MinSort [5]. 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.

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 (FTL) 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.

The organization of the paper is as follows. Section 2 has background on external sorting with specific focus on embedded devices. Section 3 presents the no output buffer external merge sort algorithm. Section 4 contains experimental results, and the paper closes with conclusions and future work.

II. 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. [6] 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.

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 [4] or using random reads to search for minimum values [5], [7]. The second technique is to optimize the run merge phase by indexing the data and using random reads to retrieve tuples in minimum order [6].

MONTRES [3] 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 [2] which used a decision rule to determine when to use clustered (sequential-based) or unclustered (index-based) sorting.

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 [8] 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 [9] 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 [5] 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 [4] 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

```
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
```

Fig. 1. Run Generation Code

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 [5] show that MinSort is significantly faster than FAST [4] 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.

III. Sorting Algorithm

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.

The algorithm for the run generation phase is in Fig. 1 and the run merge phase is in Fig. 2. 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 for very small M.

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

```
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
         bufOut[i] <- EMPTY  // Position of last output record block in this block
      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. 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

         bufOut[0] += recordSize    // For buffer 0, bufOut[0] stores offset to write next output record
         // 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
            buffer[smallBlock] <- read(dataFile, runOffsetPtr[i])
         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
```

Fig. 2. Run Merge Code

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 $bufCurrPtr$ 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.

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.

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.

In Fig. 3 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.

The percentage reduction in I/Os is given by the formula $(log(M) - log(M-1))/log(M)$. Fig. 4 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.

IV. Experimental Evaluation

The experimental evaluation was conducted on an Arduino MEGA 2560 [8] 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

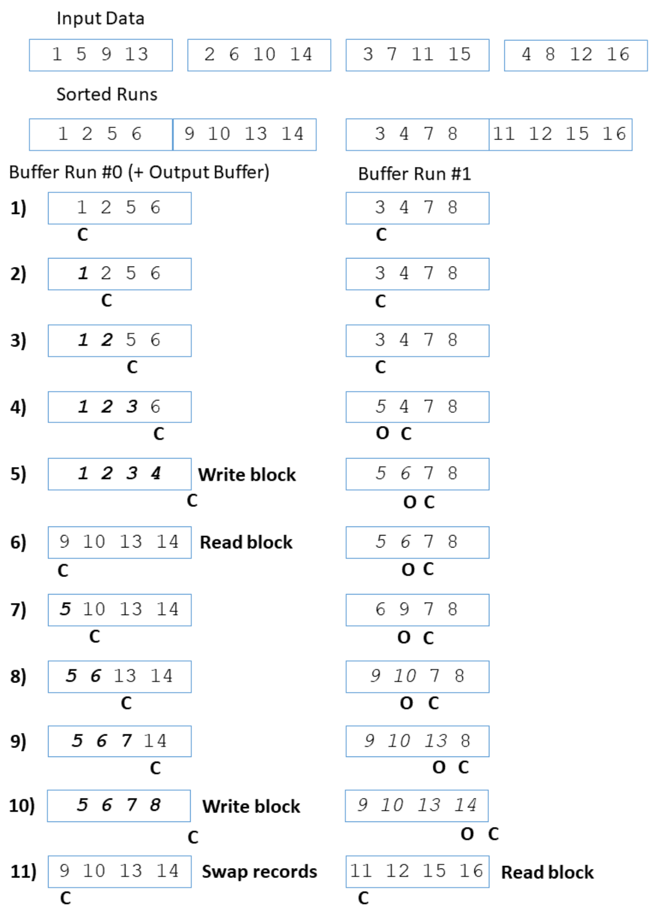

Fig. 3. Example for M=2

2	3	4	5	6
∞	37%	21%	14%	10%

Fig. 4. Theoretical Performance Improvement by M

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.

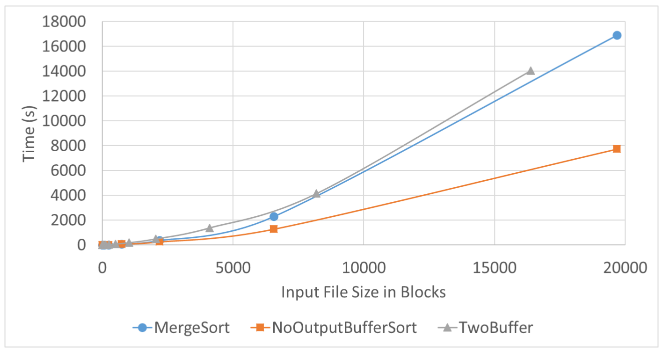

Fig. 5. Sorting Performance by Time (s)

The standard external merge sort was compared with the optimized version that uses no output buffer. The sort time in seconds (Fig. 5) and number of I/Os (Fig. 6) were captured.

The results show merge sort without an output buffer for the $M = 3$ and $M = 2$ cases. For $M = 3$, the theoretical 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.

V. 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 [5] and MONTRES [3] with the external merge algorithm to achieve even better performance.

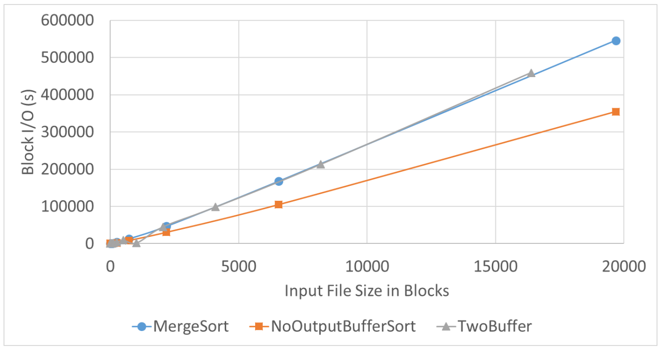

Fig. 6. Sorting Performance by I/Os

References

[1] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132960.1132964

[2] C.-H. Wu and K.-Y. Huang, "Data sorting in flash memory," *Trans. Storage*, vol. 11, no. 2, pp. 7:1–7:25, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2665067

[3] A. Laga, J. Boukhobza, F. Singhoff, and M. Koskas, "MONTRES: Merge ON-the-Run External Sorting Algorithm for Large Data Volumes on SSD Based Storage Systems," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1689–1702, Oct 2017.

[4] H. Park and K. Shim, "FAST: Flash-aware external sorting for mobile database systems," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1298 – 1312, 2009.

[5] T. Cossentine and R. Lawrence, "Fast sorting on flash memory sensor nodes," in *Proceedings of the Fourteenth International Database Engineering and Applications Symposium*, ser. IDEAS '10. New York, NY, USA: ACM, 2010, pp. 105–113. [Online]. Available: http://doi.acm.org/10.1145/1866480.1866496

[6] J. Lee, H. Roh, and S. Park, "External mergesort for flash-based solid state drives," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1518–1527, May 2016.

[7] Y. Liu, Z. He, Y.-P. P. Chen, and T. Nguyen, "External sorting on flash memory via natural page run generation," *The Computer Journal*, vol. 54, no. 11, pp. 1882–1990, 2011.

[8] Arduino homepage. [Online]. Available: http://arduino.cc

[9] P. Andreou, O. Spanos, D. Zeinalipour-Yazti, G. Samaras, and P. K. Chrysanthis, "FSort: external sorting on flash-based sensor devices," in *DMSN'09: Data Management for Sensor Networks*, 2009, pp. 1–6.