Running Median Algorithm and Implementation for Integer Streaming Applications

Oswaldo Cadenas and Graham M. Megson

***Abstract*—A novel algorithm is proposed to compute the median of a running window of *m* integers in *O*(lg lg *m*) time. For a new window, the new median value is computed as a simple decision based on the previous median and the values removed and inserted into the window. This facilitates implementations based on data structures that support fast ordinal predecessor/successor operations. The results show accelerations of up to factors of six for integer data streaming in typical embedded processors.**

***Index Terms*—Median, median filter, running median, streaming algorithms, embedded processors**

I. Introduction

THE most common solution for the running median problem is to compute, per iteration, the median M of a set of m elements using any known sorting routine such as Merge sort in $O(m \lg m)$ time [1]. This process is repeated over n iterations for a global computational time of $O(nm \lg m)$. This simple solution improves to $O(nm)$ computational time at best when using a linear sorting routine such as counting sort or bucket sort [1]. Clearly, the best you can do for streaming applications is to improve finding the median fast, much faster than $O(m)$. This paper develops a novel solution where on each iteration the median M is found in $O(\lg \lg m)$ time. This is a small cost to obtain the median on a window of m integers yielding a global running time of $O(n \lg \lg m)$ for the running median problem. The solution draws from the nature of the online median problem [2] taking advantage of the operations required when changing from the set of m elements in one iteration to the m elements in the next iteration. The set of m elements per iteration are referred to as a window; they are kept in order of arrival with each element processed as an integer. A new integer arrives into the window per unit of time while the oldest integer is removed. This moving of data is similar to the more familiar concept of moving average used in statistics. This work develops a very simple decision rule that allows computing the median for a running window of size m as fast as predecessor/successor operations can be processed on the previous window median. The contribution here results in accelerations, when the window is kept sorted by construction. Practical implementations of the algorithm given here can exploit any existing integer data structure with fast successor/predecessor operations. Indeed data structures such as van Emde Boas [3] or y-fast tries [4] have successor/predecessor operations with execution time $O(\lg q)$ when mapping distinct q-bit integers. For a window of m q-bit integers with no repeats, successor/predecessor operations are performed in constant time using van Emde Boas. This paper gives a fast solution for the median of a running window with repeats. Possibly the best known simple and generic solution to this problem for embedded processors is the use of double heaps [5] and skip lists [6] with $O(\lg m)$ time at best to find the median per iteration. The implementation reported here gives an advantage for cases where $m > q$; this is corroborated in the results. We report speedups in execution time of at least a factor of six against a skip list implementation for various q and m sizes when programmed in embedded processors.

The classical application of the median is as a filter for the removal of the so called salt and pepper noise in images [7]; its usefulness far extends into the rich area of image processing such as detecting cut-and-paste forgeries in images [8]. The online median is used in data mining as in the k-median problem used for optimizing placement of facilities [2], and in clinical applications and health care through quantile regression [9].

The running median algorithm proposed here exhibits features of great practical interest: it can be applied for any other rank-order statistics, it has a small cost for a window size m, it can easily be implemented in software as addressed or as a hardware architecture solution (not addressed).

II. Preliminaries

A window W is a collection of copies of m integers (unordered) that are all not necessarily distinct; a window may have repeated integers. If the window has not repeated integers we will refer to it as a set window W_S; $\forall x, y \in W_S \rightarrow x \neq y$. Each integer is taken from a set of integers S where x_i of S satisfies $x_i \leq 2^q \forall i \in 1,\ldots,m$; that is, each integer is a q-bit word. If W is sorted, the successor/predecessor of an integer at position j is the integer at position j+1/j-1 respectively (if it exists). For a self-contained reading we include a few standard definitions from textbooks.

Submitted for review on December 19th 2017.

O. Cadenas is with the School of Engineering, London South Bank University, London, UK (e-mail: cadenaso@ lsbu.ac.uk).

G. M. Megson is with the Department of Computer Science, University of Westminster, London, UK (e-mail: g.megson@westminster.ac.uk).

Definition 1. An integer element $x \in W$ has rank k if there are no more than $k-1$ integers in W that are less than x and there exist at least k integers in W that are less or equal to x.

Definition 2. The median of W is an integer of rank $k+1$ in W when $m = 2k+1$. We make $m = 2k+1$ for convenience, so that if W is sorted in ascending order the median is the integer at position $k+1$.

Definition 3. Let $x[j]$, $j = 1,2,\ldots,n$ be a sequence or time series. The *running median* is defined as the sequence $y[j] = median(\{x[j], x[j+1], \ldots, x[j+m-1]\}), j = 1,\ldots,n-m+1$. A median is computed on the last m windowed samples of the time series, where at each time step, a new integer arrives and an old integer leaves the window.

Theorem 1. *For a window W with $2k+1$ integer elements x_i $i = 1,2,\ldots,2k+1$ sorted in an array A with median M, assuming one old integer is removed from W while a new integer is inserted into W forming a new window W' so that W' still holds $2k+1$ integer elements, the median M' of W' is*:
if (*old* ≥ *M*) **and** (*new* < *M*)
  M' = maximum(predecessor(M), new)
else if (*old* ≤ *M*) **and** (*new* > *M*)
  M' = minimum(successor(M), new)
else
  M' = M

Proof. First consider a window set W_S; notice W_S is a set of (unordered) integers that can be arranged with a "less than relation", ≤. Let an array A be a sorted version of W_S thus the median always resides at position $A[k+1]$. For ease of reference let the old integer removed from the window be at position $W_S[2k+1]$ and the new inserted integer be at position $W_S[1]$. Observe that the median M' upon insertion and deletion into the set window is either:

(i) unchanged
(ii) $A[k]$ or the predecessor of M
(iii) $A[k+2]$ or the successor of M
(iv) the new element $W_S[1]$

Consider the insertion handling of *new* and *old* integers into A to maintain a sorted array. If the *new* and *old* integers are on opposite sides of the location $A[k+1]$ then the integers in A are shuffled left or right between the location of insertion and deletion. That is, the old integer is replaced with the successor (or predecessor) rippling back to create a place where to insert the new integer. This implies that the median M' is either case (ii) (deletion on right) or case (iii) (deletion on left). If the *new* and *old* integer are on the same side with respect to the location $A[k+1]$ then the shuffle has no effect on the median which implies case (i). Of course, if the place for the new integer occurs at $A[k+1]$ then case (iv) applies. Secondly, for a window W composed of a bag of $2k+1$ integers (elements may be repeated) the result holds since the integers in the window is a totally ordered set with a relation "less or equal to", ≤. ■

Corollary 1. *Theorem 1 holds for any rank statistics r.*

Proof. In Theorem 1, the result is shown for rank $r = k+1$, or the median. Replace $k+1$ in the reasoning of the proof for Theorem 1 for the rank statistics r of interest. ■

Corollary 2. *Theorem 1 has running time O(1).*

Proof. Theorem 1 assumes a sorted array of $2k+1$ integers, so knowing median M is of time $O(1)$ by Definition 2. Similarly, *predecessor*(M) and *successor*(M) are of time $O(1)$. As *maximum* and *minimum* operations are of time $O(1)$, all three cases of Theorem 1 are of running time $O(1)$. ■

Theoretically, the new window median can be found in constant time; this constant is essentially how fast a *predecessor* or *successor* operation is performed when the window is kept sorted. Implementing an algorithm based on this result alone is challenging. The implementation here for keeping a generic window sorted after one deletion and one insertion has a small cost of $O(\lg \lg m)$ time, as we will see shortly.

A. Running Median Algorithm

Procedure 1. *Running_Median*(x, k)
Input: Sequence x of length n, parameter k
Output: Sequence y of length $n-(2k+1)+1$ (Definition 3)

1. Process $m=2k+1$ integers to form window W
2. Find median M for window W; $j = 1$, $y[1] = M$
3. Make $old = W[2k+1]$ and $new = x[2k+2]$
4. Make $W = \{new, W[1:2k]\}$ as the new window
5. Apply Theorem 1; $j = j+1$, $y[j] = M$'
6. Repeat step 3

The algorithm is this simple; fill in the first window after processing m integers from the input sequence $x[j]$ (Step 1). Find the median M for this first window (Step 2) using any median finding method; and then slide through the window all the remaining n-1 integers (Step 6) from sequence $x[j]$ forming a new window for each new integer. It places *new* integer into position $W[1]$ while shifting window positions 1,…, $2k$ to the right thus evicting $W[2k+1]$ from the window as the *old* integer (Steps 3, 4). The median for the newly formed window is found by applying Theorem 1 and added to the output sequence $y[j]$. The loop Step 2 through Step 6 is repeated n-1 times while Step 5 takes constant time as proved. However, as previously noted our implementation to keep the conditions for applying Theorem 1 in Step 5 after a new window in Step 4 has a small cost. This paper examines implementations based on bit-vectors.

B. Related work

In the running median algorithm of Procedure 1, trivially, the window can be initialized with all entries to a constant M_0 so that $M = M_0$ in Step 2 is found in $O(1)$ time without the need to call any known finding median method. Table 1 shows a summary of known solutions to the running median problem as compared to the work in this paper.

For a useful comparison of popular methods used in practice for this problem see [10]. For the common case of integers, special data structures lend well towards finding the median such as

double heaps [5] and skip lists [6]. Although Theorem 1 is of $O(1)$ time, when implemented using a van Emde Boas tree [3] Procedure 1 is of time $O(n \lg \lg m)$ since predecessor/successor operations incur a cost of ($\lg \lg m + \lg q$) time per window (see III-B). Similar results are obtained using a flat array of integers with a stack of smaller arrays to act as summaries of the array below (as with a van Emde Boas tree). For practical purposes, our work finds the median for each running window as fast as predecessor/successor operations can be performed within that window. Theorem 1/Procedure 1 stand as a base for future alternative faster implementations.

TABLE 1
KNOWN SOLUTIONS TO THE RUNNING MEDIAN PROBLEM

Technique	Time Complexity	Comments
Brute for Merge sort	$O(nm \lg m)$ [1]	Simple, slow
Brute force linear sort	$O(nm)$ [1] [10]	Simple, still slow
Sorted linked lists	$O(nm^{1/2})$ [11]	Requires pointers
Histogram-based	$O(nk)$ [12]	Applicable to images
Double heap	$O(n \lg m)$ [5]	Applicable to images
Huang's optimizations	$O(n)$ [7]	Complex, for images
Skip list	$O(n \lg m)$ [6]	Generic, simple
This implementation	$O(n \lg \lg m)$	Generic, simple

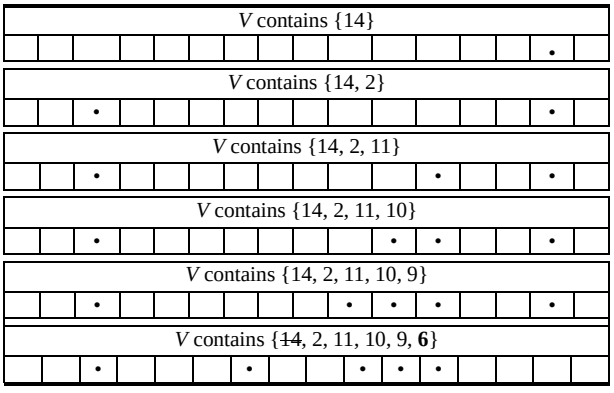
III. IMPLEMENTATION GUIDELINE

Note that the result of Theorem 1 requires every integer in the window to have its own position, including repetitions. This, although trivial in principle, has to be handled when implementing a programmed solution. We frame and examine implementations under the RAM mode of computation [1].

A. Principle of Operation

Let's assume we have a window W_S, that is, all integers are distinct. Consider five 4-bit integers with the set W_S = {14, 2, 11, 10, 9} (with order of arrival left to right). We arrange a 16-bit long bit-vector V to represent them; Table 2 shows one way to populate V from empty, one integer at a time, until all five integers are inserted. Say integer 14 now leaves the window and integer 6 gets into the window; removing bit 14 in V while inserting bit 6 in V keeps five integers in V and they are sorted. That is the key principle to maintain the conditions to apply Theorem 1 between Step 4 and Step 5 in Procedure 1. The sorting of integers in a window is performed by construction with a suitable data structure rather than relying on an explicit call to a sorting routine.

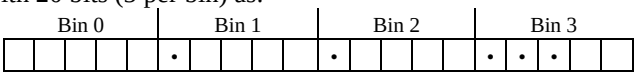
TABLE 2
Walk through Set {14, 2, 11, 6, 9}



V is composed of 16 bins (bits from 0 to 15 as boxes at the bottom) all initially empty (0). Upon integer with value v arrives the bin v is filled in with a one (•). In the last snapshot integer 14 was removed, and 6 inserted; V still contains five integers.

B. Generic Window

Any integer may appear at most $2k+1$ times in a window. We can associate a bin to each integer value, with each bin composed of $2k+1$ bits to account for repetitions. For instance, a window of five integers with values {2, 3, 3, 1, 3} are mapped with 20 bits (5 per bin) as:



This is similar to a histogram where each bin count is maintained in a base-1 number system or unary. Thus, $2^q(2k+1)$ bits are required to account for repetitions of integers within a window. In general, we can define a bin B_v as a bin allocated to integer value v such that the number of repetitions, R_v is given by the sum of all the unary digits of the bit count. We keep $p \geq 2k+1$ unary digits per bin so that a bin has a more convenient bit length to match bytes or words. In the example above, each bin is of five base-1 digits; Bin 3 count is 00111_1 (or 11100_1) to represent that the integer value 3 appears three times in the window or $R_3 = 3$. We can perform increment and decrement operations on these base-1 counts in a very simple way. Note the base-1 digits in B_v can be read as a decimal D_v; in other words, $D_v = 2^{R_v} - 1$. Therefore an increment operation on a bin count B_v is performed by $2D_v+1$. A bin decrement on a bin count B_v is performed by $B_v >> 1$ with ">>" understood as the logical right shift operation on bits. Thus, each integer in a window, including repeated ones, maps as a unique single bit within a $m2^q$ bit string so that for every new window of m integers, exactly m bits are set across the bit string of size $U = m2^q$. We use van Emde Boas to build a tree for this bit string. This has an implementation cost of $O(\lg U)$ or $O(\lg \lg m + \lg q)$ time per window or $O(\lg \lg m)$ time as q is independent of m.

IV. EXPERIMENTS

A. Implementations

A skip list implementation is used as a base for comparisons [6]. A skip list has a linked list at level 1 where each of the m integers in a window reside in a node (so repetitions appear as different nodes). Thus, at level 1 all integers are sorted. A skip list superimposes, in a probabilistic way, another linked list at higher levels; these upper level lists can skip ahead during a search for an item. By adding an extra field to each node, for tracking the distance of a node from the head node, this skipping mechanism can be conveniently exploited for faster computation of the running median [13]; the median is the node at level 1 with a distance $k+1$ from the head and it takes $O(\lg m)$ time. Thus, our implementation is expected to be faster than a skip list when $\lg \lg m + \lg q < \lg m$, that is when $m > q$ since $m > 0$ with a speedup gain of $(\lg m)/(\lg \lg m)$. We have implemented a skip list with a double-linked list at level 1 for

making $O(1)$ processing time for the predecessor and the successor to the median; note the median is of time $O(\lg m)$. Two Procedure 1 implementations were evaluated (following the guidelines given in Section III): a full van Emde Boas tree (veb) and an array tree (referred to as array). The array tree uses arrays of packed integers as bit-vectors and follows the main idea of clusters and summaries as used in van Emde Boas trees. A leaf in a van Emde Boas tree has size 2; in the array tree used in our evaluations a leaf is pruned to the native size of a machine word, typically 32-bit or 64-bit accordingly.

B. *Experimental setup*

The implementations were compiled and ran in five different machines: a Raspberry Pi 2 running a 32-bit OS (pi32), a Raspberry Pi 3 running a 64-bit OS (pi64), a Nvidia Embedded Development Kit (32-bit, tk1), an Amazon virtual machine EC2 instance (64-bit, aws) and a x86-64 laptop (64-bit).

C. *Datasets*

Three different streaming datasets were used: one of radio frequency picked up by a Software Defined Radio (sdr) USB dongle: these are 8-bit samples; one of accelerometer data from mobile users with samples of 12-bit; and one of Winston Churchill speeches (in the public domain) with voice samples of 16-bit. Datasets were set to 10M samples each. Additionally uniform distributed random data of 8, 12 and 16-bit samples were generated. On each different machine, and for each implementation, running time to process all the 10M samples were recorded and annotated for each different dataset. Speedup in running time is then calculated over averages of 20 runs each.

D. *Results*

Fig. 1 shows one instance of speedup gain as a function of window size m while Fig. 2 shows speedup gain across five machines for a window size m of 25. For the embedded pi32 machine, the speedup factor is consistently above 6. An acceleration factor of at least two is seen consistently; it is well over a factor of eight in some instances. As expected, for $m > q$, our solution is always faster than the skip list. The array implementation is faster; this was observed across the three bit length of samples. The speedup for the streaming data is much higher due to the empirical observation that the nature of the data makes it more likely for the 'else' case of Theorem 1 to show up in the computation. The factor of acceleration will depend on the particular programming implementation; implementations here have avoided using specific optimizations native to the processor on a given machine. The simple skip list solution is also accelerated by adding a double linked list at level 1 and exploiting finding the median based on successor/predecessor operations as proposed here for the running median computation problem. On an embedded Raspberry Pi 2 it is possible to process over 2000 windowed medians per millisecond or streaming samples at a rate of over 10 MHz in an embedded Nvidia kit.

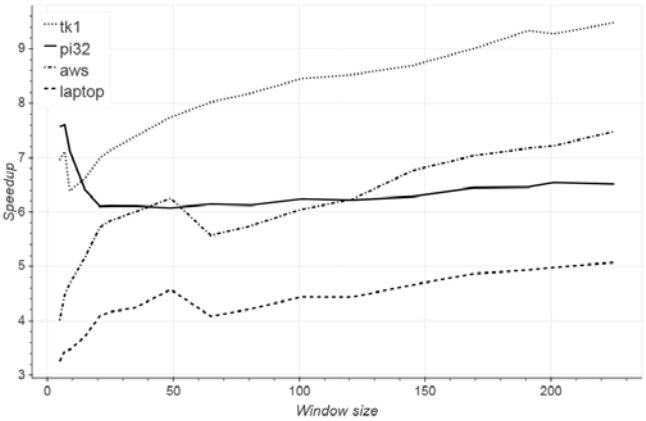


Fig. 1. Factor of speedup in execution time of the array implementation against a skip list implementation. The window size is varied from 5 to 225. The running times are averages of streaming 12-bit data samples and 12-bit uniform distributed random samples.

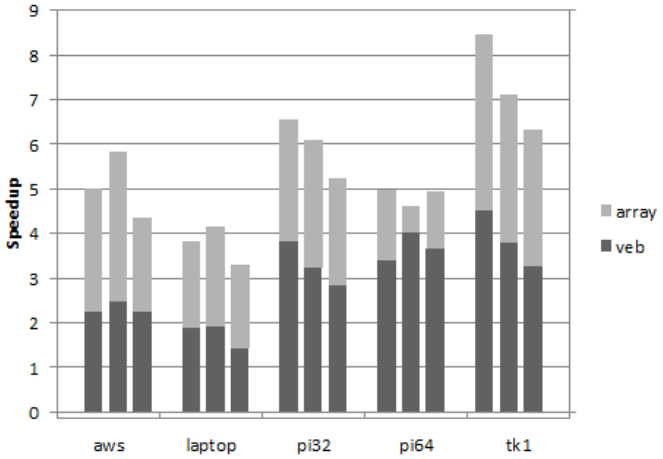


Fig. 2. Speedup factor in execution time over five different machines. For each machine a group of three bars correspond to 8, 12 and 16 bit samples respectively. On each bar, the top value is the speedup achieved by the array tree (array) while the darker grey gives the speedup for a van Emde Boas tree implementation (veb).

V. Conclusion

This paper gives a theoretical result for computing the running median on the last m integers of a set of n integers as fast as predecessor/successor operations can be performed on a data structure containing the last m integers. With a van Emde Boas implementation, the cost of predecessor/successor per iteration is of $O(\lg \lg m)$; a small cost in terms of m with a performance gain of $(\lg m)/(\lg \lg m)$ over a skip list implementation. The theoretical result stands for yet faster future novel implementations. Not only the median but also any rank statistics is supported. If the operations for creating a window of samples as presented here are preserved the result can be extended to higher dimensions.

References

[1] T.H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein "Introduction to Algorithms", The MIT Press, 2nd Ed. 2003.

[2] R.R. Mettu, and C.G. Plaxton, "The Online Median Problem," *SIAM J. Comput.*, vol. 32, no. 3, pp. 816-832, 2003.

[3] P. van Emde Boas, "Preserving Order in a Forest in Less than Logarithmic Time," *Proceedings of the Annual Symposium on Foundations of Computer Science FOCS.*, pp. 75-84, 1975.

[4] D.E. Willard, "Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(n)$," *Inf. Process. Lett.*, vol. 17, no. 2, pp. 81-84, 1983, doi:10.1016/0020-0190(83)90075-3.

[5] J.T. Astola, and T.G. Campbell, "On Computation of the Running Median," *IEEE Trans. On Signal Processing*, vol. 37, no. 4, pp. 572-574, 1989.

[6] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM*, vol. 33, no. 6, pp. 668-116, 1990, doi: 10.1145/78973.78977.

[7] S. Perreault, and P. Hébert, "Median Filtering in Constant Time," *IEEE*

Trans. on Image Processing, vol. 16, no. 9, pp. 2389-2394, 2007, doi: 10.1109/TIP.2007.902329.

[8] J. Chen., X. Kang, Y. Liu, and Z. J. Wang, "Median Filtering Forensics Based on Convolutional Neural Networks", *IEEE Signal Processing Letters*, vol. 22, no. 11, pp. 1849-1853, 2015.

[9] LJSMI, Editor, "Application of Quantile Regression in Clinical Research: An Overview with the Help of R and SAS Statistical Package," *Int. J. of Statistics and Medical Informatics*, vol. 2, no. 1, 2017, doi:10.3000/ijsmi.v2i1.5

[10] M. Juhola, J. Katajainen, and T. Raita, "Comparison of Algorithms for Standard Median Filtering," *IEEE Trans. On Signal Processing*, vol. 39, no. 1, pp. 204-208, 1991.

[11] S.D. Mohanty, "Efficient Algorithm for Computing a Running Median" Technical Note LIGO-T-030168-00 D, August 13, 2003, https://ddc.ligo.org/T030168/public

[12] Q. Zhang, L. Xu, and J. Jia, "100+ Times Faster Weighted Median Filter (WMF)," *IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 2830-2837, 2014. doi: 10.1109/CVPR.2014.362

[13] R. Hettinger, "Deep thoughts by Raymond Hettinger: Running median", http://rhettinger.wordpress.com/tag/running-median. Access on: August 31 2017.