Transactions Briefs

Algorithms for Max and Min Filters with Improved Worst-Case Performance

Mike Brookes

***Abstract*—This brief presents three algorithms for implementing a running-max/min filter of arbitrary order K, in which the average computation time per sample is asymptotically independent of K when the input data samples are statistically independent and identically distributed. The algorithms differ in their worst-case performance when acting on correlated input signals: for one of the algorithms, the computational complexity is of order K, while for the other two it is of order $\log(K)$. This brief gives the theoretical and experimental performance for a number of real and synthetic input signals.**

***Index Terms*—Algorithms, nonlinear filters, rank filters, tree data structures, sorting.**

I. Introduction

This brief is concerned with algorithms for implementing a running-max filter of order K, where the output is equal to the largest of the previous K input samples. If the input and output of the filter are $x(n)$ and $y(n)$ respectively, we have

$$y(n) = \max_{k=0}^{K-1} \left(x(n-k) \right). \tag{1}$$

The same algorithms can be used to implement a running-min filter by reversing all the data comparisons. Running-max and -min filters are widely used in the processing of speech and image signals because of their robust behavior in the presence of noise and signal nonstationarities [1]–[3].

Three different algorithms, MAXLINE2, MAXTREE, and MAXTREE2, are described below, and their performance on a number of input signals is evaluated. The following notation is used in the algorithm descriptions:1) $b[j]$ denotes the contents of buffer location $j = 0, 1, \ldots$; 2) $i\%j$ denotes the remainder when i is divided by j; 3) $i \oplus j$ denotes the bitwise exclusive-or of the integers i and j; and 4) floor (z) denotes the largest integer not exceeding z. The algebraically largest and smallest values that can be taken by $x(n)$ are denoted by $+\infty$ and $-\infty$, respectively.

II. MAXLINE2 Algorithm

Pitas has presented the MAXLINE algorithm [4] for implementing a running-max filter. He shows that when the $x(n)$ are independent and identically distributed (i.i.d.), the expected number of data comparisons per input value is independent of the filter order K. For large K, however, the computation time of his algorithm is dominated by data movements within the storage buffer and is asymptotically proportional to K. The data movements can be eliminated and the performance of the algorithm improved by using a circular buffer to store the input values. If the number of distinct values that can be taken by $x(n)$ is small compared to K, the performance can be further improved by keeping track of the most recent input value that attains the current maximum. The revised algorithm, MAXLINE2, is shown in Fig. 1.

The algorithm stores the input samples in a circular buffer $b[^*]$ of length K, whose contents are initialized to $-\infty$. In step A, a new input sample $x(n)$ is stored in the buffer at $b[j]$, overwriting the now obsolete value $x(n-K)$. The subsequent steps of the algorithm then update the pointer p, such that $b[p]$ is the maximum value within the buffer. In step B, we distinguish between three possible situations: 1) if $x(n)$ equals or exceeds the previous maximum, $y(n-1)$, then $x(n)$ becomes the new maximum; 2) otherwise, if $p = j$, then since the now discarded $x(n-K)$ was the previous maximum, we must go to step C to search for a new maximum; 3) if neither 1) nor 2) applies, we can retain the previous maximum, since it exceeds $x(n)$ and is still within the buffer. Only for case 2) do we perform step C, in which we search the entire buffer for its maximum value. We perform this search by examining the elements of the buffer in reverse chronological order beginning with the current sample $b[j]$. By performing the search in this order, we ensure that if the maximum value should arise several times, we will set p to correspond to its most recent occurrence. In the final step of the algorithm, step D, we output $b[p]$ as the current maximum and return to step A to process the next input sample.

We only perform step C when the previous maximum $y(n-1)$ corresponded to the oldest sample within the buffer $x(n-K)$. We will, therefore, minimize the number of searches required by ensuring that p always corresponds to the most recent instance of the current maximum. We achieve this by setting p in step C to the most recent instance of the maximum value and by updating p in step B whenever a new input sample is equal to the current maximum.

III. MAXTREE Algorithm

The MAXTREE algorithm, shown in Fig. 3, is similar in structure to an algorithm introduced in [4] and developed in [5]. Unlike those algorithms, however, it has an average computation cost that is asymptotically independent of K when the $x(n)$ are independent.

The algorithm requires $2K$ buffer locations logically arranged as a binary tree. The numbering of the tree nodes for $K = 5$ is illustrated in Fig. 2. Each node $j > 1$ has a unique *parent* numbered floor $(j/2)$ and a unique *sibling* numbered $j \oplus 1$. Nodes 0 and 1 are siblings but do not have any parent. The *ancestors* of node j are all those nodes that lie on the direct path between node j and node 1. The K leaf nodes are numbered K to $2K-1$ and form a circular buffer that contains the K most recent input samples: thus, sample $x(n)$ is stored in buffer location $b[K + n\%K]$. The algorithm operates by storing at each of the nonleaf nodes a value equal to the maximum of its two children. Whenever we change the value of a leaf node, we need to update the tree. Thus, in Fig. 2, if a new value is stored in node 8 we must set node 4 to the maximum of nodes 8 and 9, then set node 2 to the maximum of nodes 4 and 5, and finally set node 1 to the maximum of nodes 2 and 3. It is useful to think of each input value ascending the tree until it reaches a node where it is less than or equal to the value stored in its sibling node. Node 1 will always contain the maximum of all K leaf nodes and is therefore the desired output signal, $y(n)$.

Pseudocode for the MAXTREE algorithm is given in Fig. 3. All buffer locations are initialized to $-\infty$ except for $b[0]$, which is permanently set to $+\infty$. In step A of the algorithm, we save the new input sample $x(n)$ in the circular buffer at position $b[j]$, replacing the previous contents $x(n-K)$, which are retained as z for use in steps B

Manuscript received April 1998; revised April 2000. This paper was recommended by Associate Editor N. Ranganathan.

The author is with the Electrical and Electronic Engineering Department, Imperial College of Science, Technology and Medicine, London SW7 2BT, U.K.

Publisher Item Identifier S 1057-7130(00)07753-3.

```
1  Initialise:  set n = 0, p = K–1, y(–1) = –∞ and foreach j = 0 to K–1 set b[j] = –∞ .
2    Step A:    set j = n % K, b[j] = x(n).
3    Step B:    if x(n) ≥ y(n–1) then set p = j and go to step D.
4               else if p = j then go to step C.
5               else go to step D.
6    Step C:    foreach m = j–1 downto 0 { if b[m] > b[p] then set p = m }
7               foreach m = K–1 downto j+1 { if b[m] > b[p] then set p = m }
9    Step D:    set y(n) = b[p], n = n+1 and return to step A.
```

Fig. 1. MAXLINE2 algorithm.

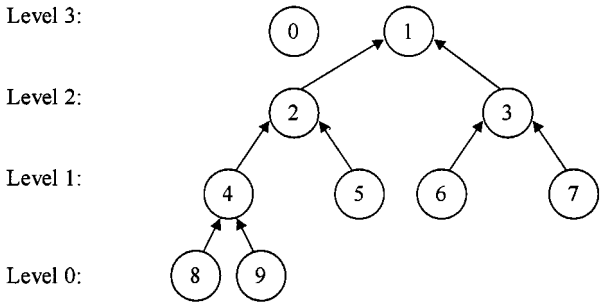

Fig. 2. Logical arrangement of buffer locations for MAXTREE algorithm for $K = 5$.

```
1  Initialise:  set n = 0, b[0] = +∞ and foreach j = 1 to 2K–1 set b[j] = –∞ .
2    Step A:    set j = K + (n % K), z=b[j] and b[j] = x(n).
3    Step B:    If x(n) > z then go to step C else go to step D
4    Step C:    If x(n) ≤ b[j⊕1] then go to step E
5               else set j = floor(j/2), b[j] = x(n) and return to step C.
6    Step D:    If z ≤ b[j⊕1] then go to step E
7               else set j= floor(j/2), b[j]=max(b[2j],b[2j+1]) and return to step D.
8    Step E:    set y(n) = b[1], n = n+1 and return to step A.
```

Fig. 3. MAXTREE algorithm.

and D. The procedure for updating the tree depends on whether the contents of $b[j]$ have been increased or decreased. Accordingly, in step B, we compare $x(n)$ to $x(n-K)$ and proceed either to step C or to step D. The value $x(n-K)$ needed for this comparison was saved in step A as z.

If $x(n) > x(n-K)$, then the new value will rise at least as far up the tree as the previous position of $x(n-K)$. We must, therefore, ascend the tree and store the value $x(n)$ at each node until we reach a point where it is less than or equal to it sibling; this procedure is performed by step C, which we repeat for as long as $x(n) > b[j \oplus 1]$, the sibling of node j. In line 5, we ascend the tree by setting $j = \text{floor}(j/2)$, set the new $b[j]$ to $x(n)$, and then loop back to line 4 to check if we have ascended far enough. Since we initialized $b[0]$ to $+\infty$, the loop will, at the latest, terminate when $j = 1$.

If, on the other hand, $x(n) \leq x(n-K)$ in step B, then the new value $x(n)$ cannot ascend the tree any higher than the previous position of $x(n-K)$. We, therefore, follow the tree up to the level that was attained by $x(n-K)$ and check that each node is correctly set to the maximum of its two children; this procedure is performed by step D, which we repeat for as long as $x(n-K) > b[j \oplus 1]$. In line 7, we ascend the tree by setting $j = \text{floor}(j/2)$, set the new $b[j]$ to the maximum of its children and then loop back to line 6 to check if we have ascended far enough.

In the final step of the algorithm, step E, we output $b[1]$ as the current maximum and then return to step A for the next input sample.

The procedure is illustrated in Table I for $K = 5$. Each row of the table shows the state of the buffer after processing the new input sample, $x(n)$, given in its second column. All buffer locations other than $b[0]$ are initialized to $-\infty$ and successive input samples are placed into the circular buffer formed by $b[5]$ to $b[9]$.

TABLE I
BUFFER CONTENTS DURING THE OPERATION OF MAXTREE ALGORITHM; ALTERED CELLS ARE HIGHLIGHTED

		Buffer Contents: $b[*]$									
n	x(n)	9	8	7	6	5	4	3	2	1	0
Init		–∞	–∞	–∞	–∞	–∞	–∞	–∞	–∞	–∞	+∞
0	3	–∞	–∞	–∞	–∞	3	–∞	–∞	3	3	+∞
1	2	–∞	–∞	–∞	2	3	–∞	2	3	3	+∞
2	3	–∞	–∞	3	2	3	–∞	3	3	3	+∞
3	7	–∞	7	3	2	3	7	3	7	7	+∞
4	4	4	7	3	2	3	7	3	7	7	+∞
5	1	4	7	3	2	1	7	3	7	7	+∞
6	2	4	7	3	2	1	7	3	7	7	+∞
7	6	4	7	6	2	1	7	6	7	7	+∞
8	2	4	2	6	2	1	4	6	4	6	+∞
9	8	5	2	6	2	1	5	6	5	6	+∞
10	9	5	2	6	2	9	5	6	9	9	+∞

An example of a sample for which $x(n) \leq x(n-K)$ arises when $n = 8$ and the value of $b[8]$ is decreased from 7 to 2. The comparison in step B is false and we update the tree using step D with $z = x(n-K) = 7$. We execute step D four times as follows:

1) $7 > b[9]$ so we set $j = 4$ and $b[4] = \max(b[8], b[9]) = 4$;
2) $7 > b[5]$ so we set $j = 2$ and $b[2] = \max(b[4], b[5]) = 4$;
3) $7 > b[3]$ so we set $j = 1$ and $b[1] = \max(b[2], b[3]) = 6$;
4) $7 \leq b[0]$ so the loop terminates and we proceed to step E.

For the following sample $n = 9$, we have $x(n) > x(n-K)$. The value of $b[9]$ is increased from 4 to 5, and we update the tree using step C. This step is repeated three times as follows with j initially set to 9:

1) $5 > b[8]$ so we set $j = 4$ and $b[4] = 5$;
2) $5 > b[5]$ so we set $j = 2$ and $b[2] = 5$;
3) $5 \leq b[3]$ so we exit the loop and proceed to step E.

Note that when, as here, the value of a leaf node is *increased*, i.e., $x(n) > x(n-K)$, all the nodes that are updated take $x(n)$ as their new value. In contrast, when the value of a leaf node is *decreased*, i.e., $x(n) < x(n-K)$, the nodes that are updated all previously had the value $x(n-K)$.

IV. MAXTREE2 ALGORITHM

It is possible to modify the algorithm so that an auxiliary array $d[j]$ stores the index of the leaf node whose value is stored in $b[j]$. The use of this auxiliary array allows a number of data comparisons to be eliminated from the algorithm since whenever a new value is stored at leaf node m, we know that all other nodes having $d[j] = m$ must be updated.

Pseudocode for the revised algorithm MAXTREE2 is shown in Fig. 4. In the initialization step, lines 3 and 4 ensure that the array $d[*]$ is initialized to consistent values with $d[j] = j$ for each leaf node and $d[j] = d[2j]$ for each nonleaf node. Steps A, B, and E are the

```
1  Initialise:  set n = 0, d[0] = 0, b[0] = +∞
2               foreach j = 1 to 2K–1 set b[j] = –∞ .
3               foreach j = K to 2K–1 set d[j] = j.
4               foreach j = K–1 downto 1 set d[j] = d[2j]
5    Step A:    set j = K + (n % K), m = j, z=b[j] and b[j] = x(n).
6    Step B:    if x(n) >= z then go to step C1 else go to step C2
7   Step C1:    if d[j/2] ≠ m then go to step C2
8               else set j=floor(j/2), b[j]=x(n) and return to step C1
9   Step C2:    if x(n) ≤ b[j⊕1] then go to step E
10              else set j=floor(j/2), b[j]=x(n), d[j]=m and return to step C2
11 Step D1:     if d[j/2] ≠ m then go to step E else set j=floor(j/2), and go to step D2
12 Step D2:     if b[2j] ≥ b[2j+1] then set b[j] = b[2j], d[j] = d[2j] and return to step D1
13              else set b[j] = b[2j+1], d[j] = d[2j+1] and return to step D1
14   Step E:    set y(n) = b[1], n = n+1 and return to step A.
```

Fig. 4. MAXTREE2 algorithm.

same as for the MAXTREE algorithm, except that a new variable m is introduced in step A to remember the node in which $x(n)$ has been stored. As before, the value $x(n-K)$ is saved as z in step A.

If $x(n) > x(n-K)$, we update the tree using steps C1 and C2. We know that $x(n)$ will ascend at least as far as $x(n-K)$, so in step C1 we ascend the tree setting $b[j] = x(n)$ for each node where $d[j] = m$. We then proceed to step C2 which, as in the MAXTREE algorithm, propagates $x(n)$ up the tree until it no longer exceeds the value of its sibling. Whenever we set $b[j] = x(n)$, we must also update the auxiliary array by setting $d[j] = m$.

If $x(n) \leq x(n-K)$, we update the tree using steps D1 and D2. Since $x(n)$ cannot ascend higher than $x(n-K)$, the nodes that need updating are precisely those nodes with $d[j] = m$. For as long as this condition holds, we ascend the tree by setting $j = \mathrm{floor}(j/2)$ in step D1 and setting $b[j] = \max(b[2j],\, b[2j+1])$ in step D2. The $\max()$ operation is here performed using an explicit comparison because as well as updating $b[j]$, we must set $d[j]$ to the corresponding leaf-node index that is stored in either $d[2j]$ or $d[2j+1]$.

Although the algorithm is somewhat more complex than MAXTREE, the number of data comparisons is significantly reduced because in steps C1 and D1 we instead use index comparisons to determine how far up the tree to ascend.

V. Computational Complexity

For each algorithm, we will calculate M, the expected number of data comparisons per input sample for the case when the input values are i.i.d. samples drawn from a continuous distribution. We assume that the probability of two identical values within the buffer is negligible.

The MAXLINE2 algorithm normally entails only one data comparison, which is made in step B. The only exception arises when the value being discarded from the buffer $x(n-K)$ is in fact the current maximum. In this case, searching the buffer for a new maximum requires a further $K-1$ data comparisons. This situation occurs when $x(n-K)$ is the largest of the $K+1$ values $x(n-K), \ldots, x(n)$, and for i.i.d. input samples has a probability of $(K+1)^{-1}$. The expected number of data comparisons per input sample is thus given by

$$M = 1 + \frac{K-1}{K+1} = 2 - \frac{2}{K+1} \xrightarrow{K\to\infty} 2.$$

If the input data samples are correlated, the above analysis no longer holds. The worst possible input signal for this algorithm is one that falls monotonically: for this signal $M = K$, since the largest value in the buffer is always the oldest sample and a search is required at every sample. The best possible input signal is one that is nondecreasing, and in this case no searches are required, so $M = 1$.

For simplicity, we restrict our analysis of the tree-based algorithms to the case when K is an exact power of two. For this case, all nodes at level R of the tree have exactly 2^R leaf-node descendants. We define the "R-cousins" of the input sample that is stored at a particular leaf node to be the 2^R input samples (including itself) whose leaf-nodes share with it an ancestor at level R of the tree. It is useful to define the following quantities:

L	$\log_2(K)$, the highest level in the tree, as shown in Fig. 2;
$p(h)$	probability that $x(n) > x(n-K)$ and that $x(n)$ rises to precisely level h in the tree;
$q(g;\, h)$	conditional probability that $x(n-K)$ had risen to precisely level g in the tree at the time of sample n, given the condition of $p(h)$ above;
δ_{gh}	equal to one if $g = h$, and zero otherwise.

We can derive an expression for $p(h)$ by noting that $x(n)$ will rise to a level $\geq h$ in the tree if, and only if, it is the largest of its h-cousins, i.e. the largest of 2^h values. If, in addition, we require that $x(n) > x(n-K)$, then $x(n)$ must be the largest of $(1+2^h)$values, and since all values are i.i.d., this has probability $(1+2^h)^{-1}$. To calculate the probability that it rises to precisely level h, we subtract the probability that it rises to a level $\geq(h+1)$ from the probability that it rises to a level $\geq h$. Thus

$$p(h) = (1+2^h)^{-1} - (1-\delta_{hL})(1+2^{h+1})^{-1}.$$

The factor $(1-\delta_{hL})$ is needed because when $h = L$, the probability that $x(n)$ rises to a level $\geq(h+1)$ is zero.

If the condition of $p(h)$ holds, then $x(n-K)$ cannot have been higher in the tree than level h since $x(n) > x(n-K)$. The probability that it was at a level $\geq g$ equals 2^{-g}, since it is just the probability that it was the greatest of its g-cousins. Following the previous argument, we deduce that

$$q(g;\, h) = 2^{-g} - (1-\delta_{gh})2^{-(g+1)}.$$

We note that, by symmetry, the probabilities $p(h)$ and $q(g;\, h)$ remain unchanged if the quantities $x(n)$ and $x(n-K)$ in their definitions are interchanged.

In the MAXTREE algorithm, one data comparison is always made in step B, and further comparisons are made in either step C or step D according to whether or not $x(n) > x(n-K)$. In the first case, if $x(n)$ ends up at level h in the tree then step C will be executed $h+1$ times and will entail $h+1$ data comparisons. In the second case, if $x(n-K)$ was previously at level h in the tree, then step D will be executed $h+1$ times for a total of $2h+1$ data comparisons: $h+1$ from line 6 and h

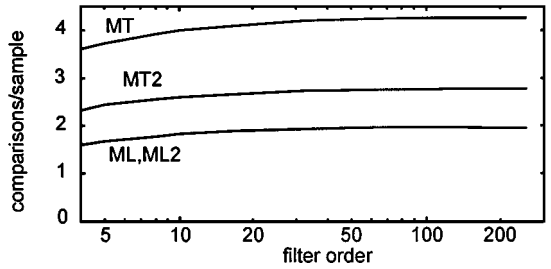

Fig. 5. Number of data comparisons per sample for the MAXLINE (ML), MAXLINE2 (ML2), MAXTREE (MT) and MAXTREE2 (MT2) algorithms as a function of filter of order for uniform random input data.

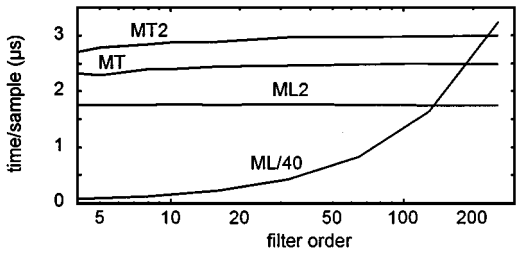

Fig. 6. Execution time per sample for the MAXLINE (ML), MAXLINE2 (ML2), MAXTREE (MT) and MAXTREE2 (MT2) algorithms as a function of filter of order for uniform random input data. Results for the ML algorithm have been divided by 40.

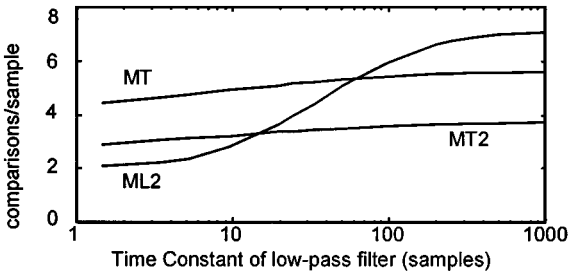

Fig. 7. Number of data comparisons per sample for the MAXLINE2 (ML2), MAXTREE (MT) and MAXTREE2 (MT2) algorithms for a filter of order 128 for low-pass filtered uniform random input data.

from line 7. Both cases arise with probability $p(h)$, so we can express the average number of data comparisons as

$$\begin{aligned} M &= 1+\sum_{h=0}^{L}(h+1)p(h)+\sum_{h=0}^{L}(2h+1)p(h) \\ &= 1+\sum_{h=0}^{L}(3h+2)p(h) \\ &= 1+\sum_{h=0}^{L}(3h+2)(1+2^h)^{-1}-\sum_{h=0}^{L-1}(3h+2)(1+2^{h+1})^{-1} \\ &= 1+\sum_{h=0}^{L}(3h+2)(1+2^h)^{-1}-\sum_{h=1}^{L}(3h-1)(1+2^h)^{-1} \\ &= 2+3\sum_{h=1}^{L}(1+2^h)^{-1} \xrightarrow{L\to\infty} 4.2935. \end{aligned}$$

If the input data are not independent but are monotonically increasing, step B is executed once per sample and step C is executed $L+1$ times, giving a total of $M = 2+L$. For monotonically decreasing data, this increases to $M = 2+2L$, since step D is now executed and entails two comparisons.

In the MAXTREE2 algorithm, one data comparison is always made in step B and further comparisons are made in either step C or step D according to whether or not $x(n) > x(n-K)$. In the first case, if $x(n)$ ends up at level h in the tree and $x(n-K)$ was previously at level g, then steps C1 and C2 are executed $g+1$ and $h-g+1$ times, respectively. In the second case, if $x(n-K)$ was previously at level h in the tree, then steps D1 and D2 will be executed $h+1$ and h times respectively. Only steps C2 and D2 entail data comparisons, so the number of data comparisons in these two cases is $h-g+1$ and h respectively. We can express the average number of data comparisons as

$$\begin{aligned} M &= 1+\sum_{h=0}^{L}\sum_{g=0}^{h}(h-g+1)p(h)q(g;\,h)+\sum_{h=0}^{L}hp(h) \\ &= 1+\sum_{h=0}^{L}p(h)\left(2h+1-\sum_{g=0}^{h}gq(g;\,h)\right) \\ &= 1+\sum_{h=0}^{L}p(h)\left(2h+1-\sum_{g=0}^{h}g2^{-g}+\sum_{g=0}^{h-1}g2^{-(g+1)}\right) \\ &= 1+\sum_{h=0}^{L}p(h)\left(2h+1-\sum_{g=1}^{h}2^{-g}\right) \\ &= 1+\sum_{h=0}^{L}p(h)\left(2h+2^{-h}\right) \\ &= 1+\sum_{h=0}^{L}\left(2h+2^{-h}\right)\left(1+2^h\right)^{-1} \\ &\quad -\sum_{h=0}^{L-1}\left(2h+2^{-h}\right)\left(1+2^{h+1}\right)^{-1} \\ &= 1.5+\sum_{h=1}^{L}\left(2-2^{-h}\right)\left(1+2^h\right)^{-1} \xrightarrow{L\to\infty} 2.7935. \end{aligned}$$

If the input data are not independent but are monotonically increasing, step B is executed once per sample and step C2 is executed $1+L$ times, giving a total of $M = 2+L$. For monotonically decreasing data this decreases to $M = 1+L$, since step D2 is only executed L times.

VI. Results

The four algorithms discussed above were implemented in C and evaluated using an input signal consisting of 500 000 integer samples uniformly distributed in the range (0 100 000). The average number of comparisons per sample is shown in Fig. 5 as a function of filter order and matches the theoretical predictions very closely. The root-mean-square deviation from theory is 10^{-2} with a maximum discrepancy of 0.012. The program execution time per sample for each algorithm is shown in Fig. 6 when using a SUN SPARC 5 workstation. The execution times for the original MAXLINE algorithm have been divided by 40 and, as expected, are far greater than for the other algorithms. The original MAXLINE algorithm is not considered further in this paper as its revised version, MAXLINE2, is uniformly superior.

Figs. 7 and 8 show the results for fixed order $K = 128$ when the input data is passed through a low-pass filter with transfer function $H(z) = (1-e^{-1/\tau}z^{-1})^{-1}$ where τ denotes the time constant of the filter. It can be seen that this correlated input data degrades the performance of all the algorithms with MAXLINE2 affected much more than the others.

Table II gives the performance of the algorithms for the following six input signals.

1) Random: $x(n) =$ random number in range 0 to 100 000.
2) Binary: $x(n) =$ random number either 0 or 1.
3) Constant: $x(n) = 0$.
4) Increasing Sawtooth: $x(n) = n\%100\,000$.
5) Decreasing Sawtooth: $x(n) = -(n\%100\,000)$.
6) Speech sampled at 16kHz: TIMIT file test/dr1/faks0/sa1 [6].

The number of samples was 317 440 for the speech signal and 500 000 samples for all others. The table gives the average execution times in microseconds per input sample for $K = 10$ and $K = 128$. The table also gives, in parentheses, the average number of data comparisons per input sample. The computational costs of the MAXTREE and MAXTREE2 algorithms exceed that of the MAXLINE2 algorithm by 44% and 74%, respectively, for i.i.d. input data; for speech data, these numbers fall to 32% and 64%.

VII. Discussion

The results verify the theoretical predictions and confirm that for both speech and random i.i.d. data, the execution time and the number of data comparisons are largely independent of the filter order.

MAXLINE2 is based on the MAXLINE algorithm from [4], but because it uses a circular buffer to avoid data movements, it has a much lower computational cost as shown in Fig. 6. The two algorithms will normally involve identical data comparisons. However, whereas MAXLINE needs to search the entire buffer whenever $x(n - K)$ equals the current maximum, MAXLINE2 only does so when it is the sole instance of the current maximum. Thus, MAXLINE2 never requires more data comparisons than MAXLINE and will require fewer whenever the current maximum occurs more than once within the buffer.

The MAXTREE and MAXTREE2 algorithms use a binary tree structure that is similar to that of algorithms described by Pitas *et al.* in [4] and [5] but store their samples in a circular buffer rather than a shift register. The use of a circular buffer gives three advantages: 1) the K data movements per sample that dominate the computation requirements of the Pitas algorithms are eliminated; 2) throughout its time in the buffer an input sample stays in the same position within the tree, and hence retains the same ancestor nodes; 3) no algorithm modifications are required when $\log_2(K)$ is a noninteger. Advantage 2) is the most significant, and it is this that reduces the average number of data comparisons for i.i.d. data from $\log_2(K)$ for the Pitas algorithms to a constant that is independent of K.

In some applications, the comparison of two data values may involve many operations. The MAXTREE2 algorithm generally requires fewer data comparisons than the MAXTREE algorithm, and will therefore give improved performance in situations where data comparisons are expensive. The MAXLINE2 algorithm has the lowest number of data comparisons for i.i.d. data but, as Fig. 7 demonstrates, this does not remain true for strongly correlated input data. Even for the mildly correlated speech data, the average number of data comparisons for the MAXTREE2 algorithm is only 20% more than for the MAXLINE2 algorithm.

The worst-case execution time arises when the input signal is monotonically decreasing (MAXLINE2 and MAXTREE) or increasing (MAXTREE2). For these cases, the number of data comparisons matches theoretical predictions and the execution time is only of order $\log(K)$ for the tree-based algorithms, but is of order K for the MAXLINE2.

Comparing the first three rows of the table, we see that the performance of all algorithms improves uniformly as the number of distinct data values is reduced. The improvement is least with the MAXTREE2 algorithm as this uses index comparisons rather than data comparisons to control the number of loop iterations.

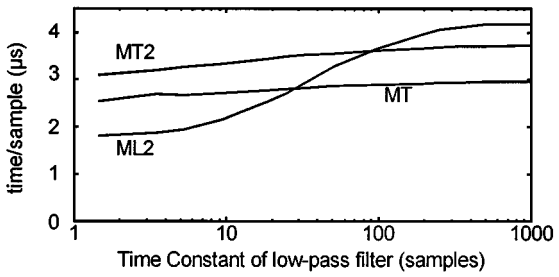

Fig. 8. Execution time per sample for the MAXLINE2 (ML2), MAXTREE (MT) and MAXTREE2 (MT2) algorithms for a filter of order 128 for low-pass filtered uniform random input data.

TABLE II
Execution Timer Per Sample (in μs) and, in Parentheses, Average Number of Data Comparisons Per Unit Sample

	MAXLINE2		MAXTREE		MAXTREE2	
	K=10	K=128	K=10	K=128	K=10	K=128
Random	1.7 (1.8)	1.7 (2.0)	2.4 (4.0)	2.5 (4.3)	2.9 (2.6)	3.0 (2.8)
Binary	1.3 (1.0)	1.3 (1.0)	2.0 (2.8)	2.0 (2.8)	2.7 (2.3)	2.7 (2.3)
Constant	1.3 (1.0)	1.3 (1.0)	1.7 (2.0)	1.7 (2.0)	2.4 (2.0)	2.4 (2.0)
Increasing	1.3 (1.0)	1.3 (1.0)	3.2 (5.4)	4.7 (9.0)	4.0 (5.4)	6.0 (9.0)
Decreasing	7.3 (10.0)	78.4 (128)	3.9 (8.8)	6.1 (16.0)	5.2 (4.4)	8.9 (8.0)
Speech	2.3 (2.8)	2.0 (2.7)	2.7 (4.7)	2.7 (4.9)	3.3 (3.1)	3.4 (3.2)

The algorithms differ in their storage requirements: the MAXLINE2 algorithm only requires a data buffer of length K whereas the tree-based algorithms require a buffer of length $2K$. In addition, the MAXTREE2 algorithm requires an auxiliary index array of length $2K$. The MAXLINE2 algorithm has the smallest code size but since all three algorithms are compact and this is unlikely to be a significant consideration.

VIII. Conclusion

Three algorithms for a running max filter have been presented each of which has as execution time that is independent of the filter order for i.i.d. data samples. The MAXLINE2 algorithm is an improved version of the algorithm in [4] and offers very good average performance on i.i.d. data. For real-time applications, however, latency restrictions and data buffering requirements are determined by the worst-case computation time and, even for i.i.d. data, this is proportional to K. For correlated input data, such as arises in speech or image processing, the performance is less good and ultimately becomes worse than the tree-based algorithms.

For i.i.d input data, the MAXTREE and MAXTREE2 have greater computational requirements than the MAXLINE2 algorithm on average, but their worst-case requirements are proportional to $\log(K)$ rather than K. Of the two algorithms, MAXTREE2 has a higher overhead but entails fewer data comparisons, and will therefore have a lower computational cost in applications where data comparisons are expensive. These algorithms require twice as much data memory as the MAXLINE2 algorithm and MAXTREE2 require, in addition an auxiliary index array of size $2K$.

In summary, the MAXLINE2 algorithm is the clear winner in terms of data buffer requirements and average execution time on i.i.d. data. However, for applications involving correlated input data or for real-time applications where worst-case execution time is important, the MAXTREE and MAXTREE2 algorithms are the preferred choice. Finally, it should be noted that although the Pitas algorithm from [5] is, as noted above, uncompetitive as a software algorithm, it remains a good choice for a parallel hardware implementation since it requires only K registers and $\log_2(K)$ comparators.

ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers and Dr. J. Chambers for their helpful comments which substantially improved this paper.

REFERENCES

[1] G. R. Arce and M. P. McLoughlin, "Theoretical analysis of the max-median filter," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, pp. 60–69, Jan. 1987.
[2] P. A. Maragas and R. W. Schafer, "Morphological filters—Part I: Their set theoretic analysis and relations to linear shift invariant filters," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, pp. 1153–1184, Aug. 1987.
[3] R. Martin, "Spectral subtraction based on minimum statistics," in *Proc. EUSIPCO-94*, Edinburgh, Scotland, Sept. 1994, pp. 1182–1185.
[4] I. Pitas, "Fast algorithms for running ordering and max/min calculation," *IEEE Trans. Circuits Syst.*, vol. 36, pp. 795–804, 1989.
[5] D. Coltuc and I. Pitas, "On fast running max-min filtering," *IEEE Trans. Circuits Syst. II*, vol. 44, pp. 660–664, Aug. 1997.
[6] J. Garofolo *et al.*, "DARPA TIMIT acoustic-phonetic continuous speech corpus (CD-ROM)," National Institute of Standards and Technology, 1990.

A CMOS Buffer Without Short-Circuit Power Consumption

Changsik Yoo

***Abstract*—A new CMOS buffer without short-circuit power consumption is proposed. The gate- driving signal of the output pull-up (pull-down) transistor is fed back to the output pull-down (pull-up) transistor to get tri-state output momentarily, eliminating the short-circuit power consumption. The HSPICE simulation results verified the operation of the proposed buffer and showed the power-delay product is about 15% smaller than conventional tapered CMOS buffer.**

***Index Terms*—CMOS buffer, short-circuit power consumption.**

I. INTRODUCTION

With the high integration level of CMOS very large scale integration (VLSI), the capacitive load of periodic signals such as clock has become very large. With such a large capacitive load, driving circuits consume a relatively large portion of the total power of a VLSI. The power consumption of a CMOS buffer driving a capacitive load consists of dynamic switching power and short-circuit power. While the switching-power consumption is unavoidable to drive a capacitive load, short-circuit power is a waste of current and should be minimized or even eliminated for low-power operation.

A conventional tapered CMOS buffer, shown in Fig. 1(a), consumes both the dynamic switching power and short-circuit power due to simultaneous turn-on of the pull-up/pull-down transistors, as illustrated in Fig. 1(b) [1]. Short-circuit power consumption can be eliminated by tri-stating the output node momentarily before every output signal transition. In [2], asymmetric inverters were used as waveform shaper to get momentary tri-state output period, but the propagation delay is increased by the asymmetric inverters. As an alternative, a feedback-controlled split-path (FS) CMOS buffer was proposed, where the output signal is fed back to control the output pull-up and pull-down transistors, as shown in Fig. 2, tri-stating the output momentarily and thereby eliminating the short-circuit power consumption [3]. But, in the FS CMOS buffer, the logic states of the split output stage drivers change

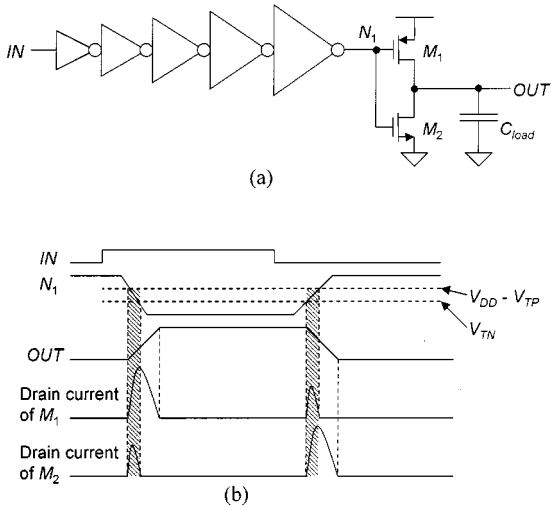

Fig. 1. (a) Tapered CMOS buffer and (b) its timing diagram.

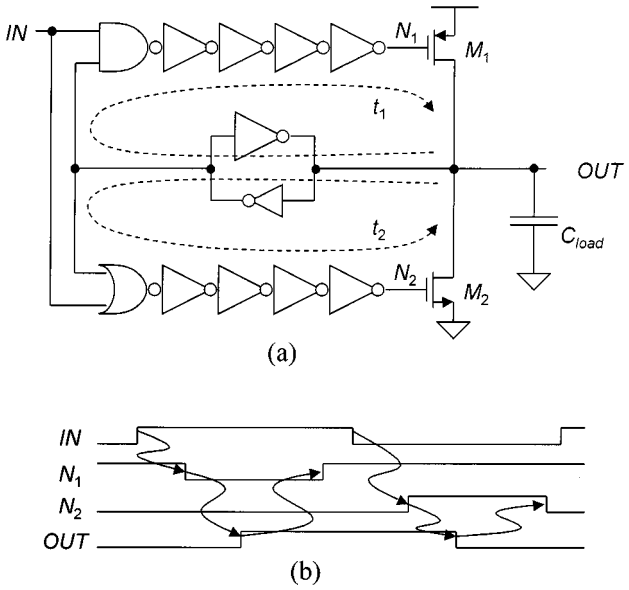

Fig. 2. (a) Feedback-controlled split-path CMOS buffer and (b) its timing diagram.

Manuscript received June 1999; revised June 2000. This paper was recommended by Associate Editor M. Bayoumi.

The author was with Integrated Systems Laboratory (IIS), Swiss Federal Institute of Technology, Zurich, Switzerland. He is now with Samsung Electronics, Kiheung, Korea.

Publisher Item Identifier S 1057-7130(00)07752-1.