Pre-print version
L. Breveglieri, V. Piuri, "Digital Median Filters", in *The Journal of VLSI Signal Processing*, Springer, pp. 191-206, July, 2002. ISSN: 1939-8018. DOI: 10.1023/A:1015487418553

Digital Median Filters

LUCA BREVEGLIERI
Department of Electronics and Information, Politecnico di Milano, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy

VINCENZO PIURI
Department of Information Technologies, University of Milan, via Bramante 65, I-26013 Crema (CR), Italy

Received January 14, 1998; Revised February 8, 2001

Abstract. Several DSP algorithms need to remove high-frequency or impulsive noise while preserving edges, e.g., in speech and image processing applications: median filtering has been proved to be more effective for achieving this goal than other filtering techniques. Efficient architectural implementation for real-time applications involves a careful VLSI design, which takes into account modularity, regularity, adaptability, scalability, throughput, circuit complexity and fault tolerance.

Four new architectural approaches are presented and evaluated in this paper to deal with different application and implementation constraints. They are: the *serial-input polarizing median filter*, the *floating median filter*, the *pipelined polarizing median filter* and the *pipelined sorting median filter*. The 1st and the 2nd architectures are based on majority voting, while the 3rd and the 4th ones are based on sorting techniques. All of them are designed so as to exhibit high scalability and to be easily pipelined for higher working frequencies.

Keywords: median filter, VLSI, dedicated architecture, rank ordering filter

1. Introduction

In many signal processing applications it is often required to attenuate noise while certain signal properties should be preserved as far as possible. Differently from linear filtering, the median filter proved itself to be useful for removing high-frequency and impulsive noise, while effectively preserving edges in speech and image signals. The properties of the median filter have been extensively studied during the last decade [1–4] and several extensions have been proposed to allow more degrees of freedom and better performances.

There are two types of median filters [1, 2, 4]: non-recursive and recursive. In 1-D non-recursive median filtering, a window of size $2N + 1$ is moved across the entire signal sequence: the centre value of each window is replaced by the median of the samples in the window itself. The median of the values in the window can be in fact considered as the best guess for the centre value. Let W_i be a window centred on the i-th input value of the signal: $W_i = \{x_{i-N}, \ldots, x_i, \ldots, x_{i+N}\}$; the filter output is $y_i = median(W_i)$. In d-D (with $d > 1$) non-recursive median filtering, a window of size $2N + 1$ for each dimension is moved along the input data. Let $W_{i,j,\ldots,m}$ be a window centred on $x_{i,j,\ldots,m}$: the filter output $y_{i,j,\ldots,m}$ is the median of the values in $W_{i,j,\ldots,m}$.

In recursive median filtering, the window stores the most recent median values as well as the input values. The i-th window for 1-D recursive filtering is $W_i^r = \{y_{i-N}, \ldots, y_{i-1}, x_i, \ldots, x_{i+N}\}$, while the filter output is $y_i = median(W_i^r)$. Similar definitions are given for the d-dimensional case. By including the most recent median values in the evaluation of the median of a new window, we can extract the characteristics of the input signal faster than in non-recursive median filters.

Several architectures have been proposed by now in the literature to support real-time signal processing by means of high-performance integrated devices (see e.g. [5–18]). Array architectures [8, 13, 15, 16, 18] have a limited circuit complexity, but often have a large sampling period due to the sequential execution of several operations in the same sampling period. Sorting network-based architectures [9, 10, 17] are inherently pipelined (i.e., a higher throughput can be reached), but they contain a large number of compare-swap units. Stack filter-based architectures [11] can reach reasonable performances with a limited modularity. Some solutions are based on semi-analog devices [15], to bound the size of the architecture, requiring however suitable technologies [12] for implementation. In [4] a review of algorithms for median filtering is proposed, and their time/space complexities are analysed. We defer to the conclusion section for more detailed comparisons with specific architectures.

In this paper we consider the design of high-performances digital architectures for median filtering with a reasonable circuit complexity: one of the main goals is to obtain a considerable improvement of regularity and modularity to match the requirements imposed by the integration techniques, by the adaptability and by the scalability of the structure itself. Computational time and throughput must be carefully addressed to implement an architecture suited for real-time digital signal processing. At the same time physical realizability introduces additional constraints for the circuit complexity (i.e., for the silicon area used by active devices and interconnection paths). Regularity and modularity are considered to support a simple design approach for dedicated DSP architectures, since the actual filter can be obtained by customizing the number of operating cells in the architecture without re-designing the internal structure of the cells themselves. Regularity and modularity are also exploited to introduce fault tolerance capabilities in the basic structure for concurrent error detection and continuous output validation, suited for mission-critical applications, by using data coding. A special attention is posed to the possibility to pipeline the proposed architecture, in order to speed up the clock frequency.

In Section 2, we present the *serial-input polarizing median filter* capable to deal with bit-serial computing structures, while the case of modular architectures both for parallel and serial data transfer is considered in Section 3 with the *floating median filter*. The *pipelined polarizing median filter* is presented in Section 4, while the systolic structure of the *pipelined sorting median filter* is described in Section 5; the first pipelined solution allows to achieve a higher throughput, while the second one has higher modularity. In each of these sections, we give a detailed description of these architectures, their evaluation from the point of views of circuit complexity, computational time, throughput, and fault tolerance. For simplicity sake, we discuss in detail the architectures for non-recursive one-dimensional median filters. Finally, in Section 6, we give a comparative evaluation to provide guidelines for the system designer.

2. The Serial-Input Polarizing Median Filter

For the case of bit-serial inputs, we designed the *serial-input polarizing median filter*; its general structure is shown in Fig. 1. The polarizing median filter is composed of three main parts: the register file, the polarizing subsystem, and the polarization selector.

The register file provides the proper data presentation for median evaluation. It consists of $2N + 1$ shift registers (one for each input sample belonging to the window under examination): each of them is n bits long, where n is the number of bits for each input sample. Shift registers are concatenated so that the most-significant output bit of register i is the least-significant input bit of register $i + 1$. Input data are introduced serially into the concatenated register file (most-significant bit first): at each clock cycle, the data stored in the register file are shifted down by one position and are recirculated on the adjacent register. Recirculation guarantees

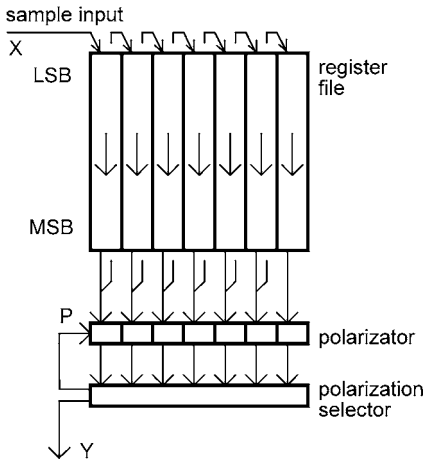

Figure 1. The serial-input polarizing median filter.

availability of the last $2N$ input signals for the subsequent input window. The analysis of signal values in a window terminates at each n-th clock cycle, i.e., after examining all bits of each sample in the window. The oldest sample is progressively discarded at the output of the $2N + 1$-th shift register, since it is not required for subsequent windows.

At each bit iteration, the bits presented at the bottom of the register file are extracted and properly "polarized" by the polarization subsystem by using the results computed in the previous iterations. Polarization allows for removing progressively the input data that differ too much from the expected median value from the computation of the median itself. Polarization consists of the substitution of actual data with a rearranged sequence of bits, which are median-equivalent to the original ones. In fact, the median value in a window is not changed if we substitute the value 0 (or, more in general, any value which is smaller than the median) to all the input samples which are smaller than the median value. Similarly, the median value is not modified if we substitute the top value that can be represented in a n-bit word (or, more in general, any value which is higher than the median) to all the input samples which are higher than the median value.

At each bit iteration, all input signals that have been recognized to be smaller than the expected median value are polarized toward the current minimum value within the window. Conversely, all input samples that appear to be higher than the expected median value are polarized to the current maximum window sample. At the end of the polarization process, the median value is the one that has never been polarized within the current window. To simplify these operations, each actual datum that is recognized to be smaller than the expected median value during the bit iteration j is replaced with a sample having the $j - 1$ most significant bits equal to the original datum, while the remaining $n - j + 1$ bits are null. Similarly, each actual datum higher than the expected median value is substituted by a sample having the $j - 1$ most significant bits equal to the original datum, while the remaining $n - j + 1$ bits are equal to 1. The circuit implementing the polarization operation is shown in Fig. 2.

Due to the presence of the register file and to the strict separation between sample holding and the polarization operation, polarization is confined within each individual window and does not affect polarization in the subsequent windows.

The polarization selector generates the new polarization P for the subsequent bit iteration from the polarized input signals available during the current iteration. At the j-th bit iteration, if the number of 1's in the set of the polarized j-th bits of all samples is over N, all polarized samples having the j-th bit equal to 0 are smaller than the expected median value, independently of the value of their bits of weight smaller than j. Similarly, if the number of 0's is over N, the polarized samples having the j-th bit equal to 1 are higher than the expected median value. Therefore, the majority value of the j-th bits gives the polarizing value at iteration j. The circuit for generating the polarization value P is shown in Fig. 3; P is 0 if the number of 1's is over N, otherwise it is 1.

The samples having the j-th bit equal to the minority value at the j-th iteration must be polarized: they are smaller than the expected median value if the minority value is 0; otherwise, they are higher than the expected median value. Polarization is achieved by setting all bits having position from j down to 0 equal to the complement of the polarizing value.

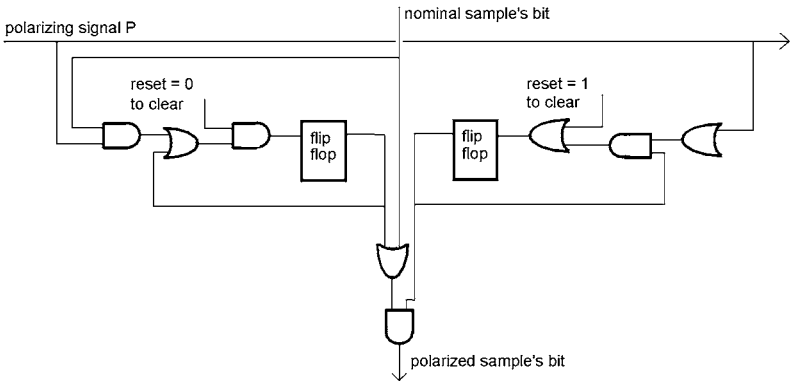

Figure 2. The basic cell for the polarization subsystem.

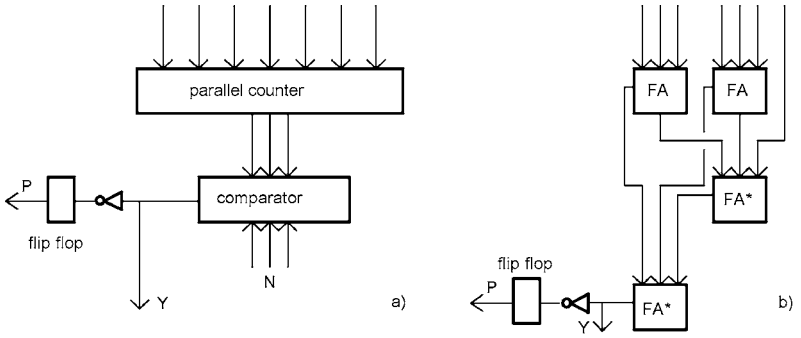

Figure 3. The polarization selector. The general structure (a). The implementation case for $N = 3$ (b): FA are full adders, FA* are full adders which generate only the carry bits.

From the previous definitions, the j-th value of the polarization signal P is equal to the complement of the j-th bit of the median value. The median is generated serially, starting from the most-significant bit.

To choose the optimal architectural solution for a specific application, it is necessary to provide some design guidelines. Circuit complexity C_1 is strictly related to the area occupied by the integrated implementation. Since silicon area is roughly proportional to the number of active devices (also assuming that the area for signal routing obeys the same proportional law), we can adopt the number of gates as first approximation of this figure of merit and, in turn, of the circuit complexity. The actual value of this characteristic is related to the specific implementation adopted for the basic components: the typical shape has been derived by adopting a traditional design for the basic units [19]. The area used by shift registers is given by $(2N + 1)nA_{ff} = 7n(2N + 1)$, where A_{ff} is the number of gates for each master-slave flip-flop [19]. The polarizator area is approximately given by $22(2N + 1)$ (see Fig. 2). Since the detailed design of the polarization selector is strictly related to the number of inputs, i.e., to N, there is no simple relationship which gives the number of gates as function of N [20]; for example, the number of gates is 28, 38 and 63 for N equal to 2, 3 and 4, respectively. In general, parallel counters can be designed as Dadda's or Wallace's addition trees [20]. The total area is therefore proportional to the sum of the previous components.

To evaluate the clock period and the throughput T_1, we adopt the same approximation introduced above: the clock period is proportional to the number of gate levels. In the following time is therefore measured in number of gate levels. The clock period τ_b to generate each individual bit of the median can be easily computed by analyzing the filter structure. Since this architecture is based on data recirculation, the evaluation of the throughput can be performed by considering the system in a steady state, i.e., by assuming that the $2N + 1$ samples have been already loaded into the register file. In fact, loading of the sample x_{i+N} is completely overlapped with the computation of the median value y_{i-1}. The time τ_b is given by the following summands: the time τ_{ff} for the one-bit shifting in the register file, the delay τ_p for polarization, and the time τ_{ps} for polarization selection. τ_{ps} is strongly related to the specific value of N, while the other components are completely independent of N. The summand independent of the internal structure of the polarization selector is $6 + 3\tau_{ff} \approx 30$, where τ_{ff} is the computational time of a flip-flop [19]. The clock period τ_b is approximately equal to 46, 46, and 52, for N equal to 2, 3, and 4, respectively.

The computational time required to generate a complete n-bits output median value is thus equal to $n\tau_b$, while the throughput T_1 of the median filter is $1/n\tau_b$.

Error detection is the basic step of any fault tolerance policy. Concurrent error detection becomes important for mission-critical applications in which output correctness must be guaranteed continuously, or when the use of periodic testing induces a fault detection latency which is too long (i.e., too many outputs may be erroneous before the fault is detected). In our architecture, concurrent error detection can be implemented by applying different techniques: however, to minimize the additional circuit complexity, while still providing a good detection capability, we consider only single stuck-at errors. On the other hand, their occurrence probability is usually much higher than the one of multiple-bit errors.

The register file can be protected by using a parity code [21]: odd parity is suited when n is odd; otherwise, even parity allows to detect single stuck-at faults for n even. Any stuck-at fault occurred before the beginning of the computation for a given filtering window can be detected with probability 1. In some cases, it can be detected also if it occurs *during* the computation of the considered window; otherwise, it is latent for one window. Hardware redundancy to support this approach consists of an additional memory bit in each sample register and of a parity checker for each sample register. The computational delay introduced by this technique is τ_b, due only to the additional parity bit that must be recirculated (checking time is shorter than τ_b). The cost of this technique is very limited for large values of n.

The polarizing subsystem and the polarizing selector cannot be protected by the above code. The simplest and most effective detection technique is in this case modular redundancy (namely, duplication). Hardware redundancy is due to the complete duplication of these two components and to the equality checker for the P signal. Time delay is due only to the equality checker. Since these two components are relatively small, duplication has a limted cost. Occurrence of disagreement is signalled to invalidate the computation.

The high regularity of the register file and the polarizator allows to adopt dynamic redundancy with reconfiguration to provide system survival after fault occurrence. Additional spare shift registers and polarizer cells are added to the nominal structure: a suited, dynamically-configurable, switched interconnection network creates the required data paths. Whenever one of the nominal components is faulty, it can be replaced by one of the spare components through switch positioning. As many faulty components as the spare ones can be tolerated in the structure. Faults in the polarization selector are more critical from the point of view of survival and more area consuming. For example, modular redundancy with voting could be used to protect this subsystem and to guarantee correct outputs even in the presence of some multiple faults. Dynamic redundancy has a higher cost; a detailed costs/benefits tradeoff is necessary, based on the evaluation of an effective implementation.

3. The Floating Median Filter

The second architecture we propose is the *floating median filter*. It has been designed to cope in particular with parallel inputs, but it can be used also for bit-serial inputs, provided that suited structures are adopted to store and to recirculate data.

In Fig. 4 we show the overall array architecture, while in Fig. 5 we give the structure of the basic cell. Such an architecture is based on a linear semi-systolic array of $2N+1$ cells. Inputs enter the filter from the left side and are broadcast through the array. The basic idea is to allow the samples values of a given window "to float" within the array until they reach the final steady distribution in which samples are totally ordered by increasing values. At each new window, the oldest sample of the previous filtering window is discarded and a new sample enters the array: again, data float until they reach the new totally-ordered distribution.

Each cell stores a sample value in the register S: at each iteration, it is compared with the new input sample X, or with the sample S_{left} stored in the cell on the left side, or with the value S_{right} in the cell on the right side, according to the selection signal G generated by a suited circuit. If the new input sample is higher than the value stored in the cell, it must be stored in a cell on the right side of the current one; otherwise, it must be placed in a position on the left side of the considered cell.

Therefore, each input sample floats within the array until it reaches the correct position in the total increasing ordering. Such a position may be already free if it was occupied by the oldest sample of the previous window. Otherwise, it is in use for another sample of the current window; in this case, it must be prepared to receive the new sample by moving the sample already

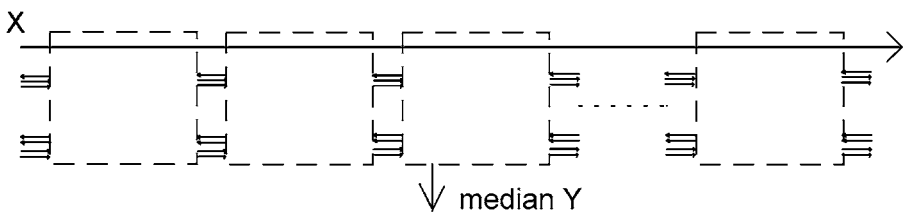

Figure 4. The array architecture of the floating median filter.

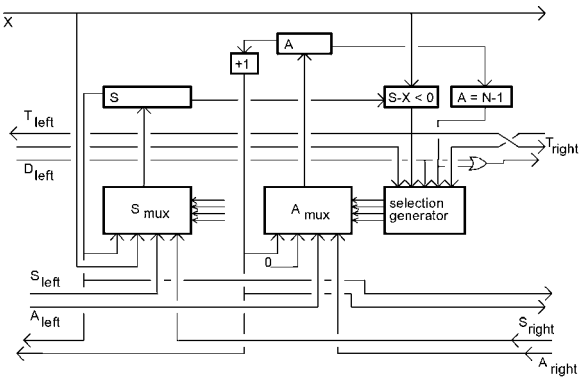

Figure 5. The basic cell of the floating median filter.

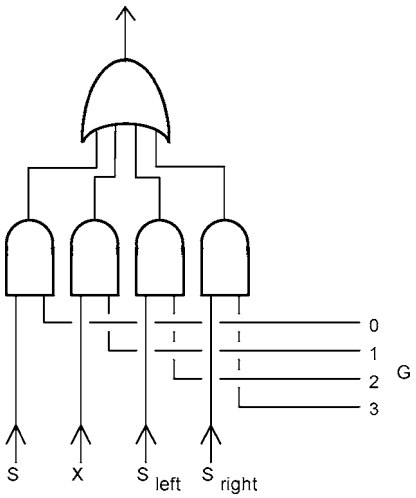

Figure 6. The sample multiplexer: The basic cell for one bit selection.

stored in it. This can be achieved by moving all the samples having a value higher than the new input sample towards the right side of the array, while samples with a smaller value are placed onto the left side. The cell separating the two groups of samples will therefore remain empty; the new input sample must be placed in such a cell to obtain the totally ordered set of sample values in the current window.

Samples aging is automatically maintained in the register A by discarding the old samples and freeing space for the incoming ones. The age of a new input sample is set equal to 0. At each iteration, the register A containing the age of the sample S is increased by one time unit. The sample having age equal to $2N$ at the beginning of the iteration is the oldest of the window and must be discarded when the new sample X enters the array.

To control data floating in the whole array architecture, we introduce a modular distributed control structure. It is based on the sample multiplexer S_{MUX}, on the age multiplexer A_{MUX}, and on the selection generator.

The sample multiplexer is used to select the new value that must be stored in the sample register S among the four possible sources: the cell on the left side (S_{left}), the cell on the right side (S_{right}), the input sample (X), or the cell itself (S). In the first case, the values stored on the left side must be moved to the right side to free a cell for an input sample smaller than the right-moving samples. In the second case, the values stored on the right side must be moved to the left side due to an input sample higher than the left-moving samples. In the third case, the cell is free and able to receive the incoming new input sample X. In the last case, there is no empty cell on the left (right) side that must be filled in by the sample stored in the considered cell, in order to free some space on the right (left) side to store the new input sample; samples smaller than the new input are already packed on the left (right) side. The sample multiplexer can be implemented by using the circuit shown in Fig. 6.

The age multiplexer has a role similar to that of the sample multiplexer. It selects the age of the sample that must be stored in the cell from one of the four possible sources: the cell on the left side (A_{left}), the cell on the right side (A_{right}), the age of the input sample (0), or the cell itself ($A + 1$). It is controlled in the same way as the sample multiplexer.

The selection generator provides the suited control signals to perform data floating; its structure is shown in Fig. 7. Selection generators cooperate to evaluate the conditions which specify whether the contents of adjacent cells must be moved to the left side or to the right one, whether the new input sample must be stored in a given cell, or whether the cells contents must be hold without any moving. In particular, each selection generator computes a signal G stating which source of new data must be used for the cell under consideration. For example, G is equal to 0 if there is no data movement affecting the cell, while G is 1 if the new input sample X must be stored in the cell. We set G equal to 2 if sample S and its age A must be moved from the adjacent cell on the right side into the considered cell. Finally, G is 3 if the reverse movement must be executed.

The control signal T of the selection generator gives information about the relative ordering between the sample S stored in the cell and the current input X, while the control signal D states the availability of a

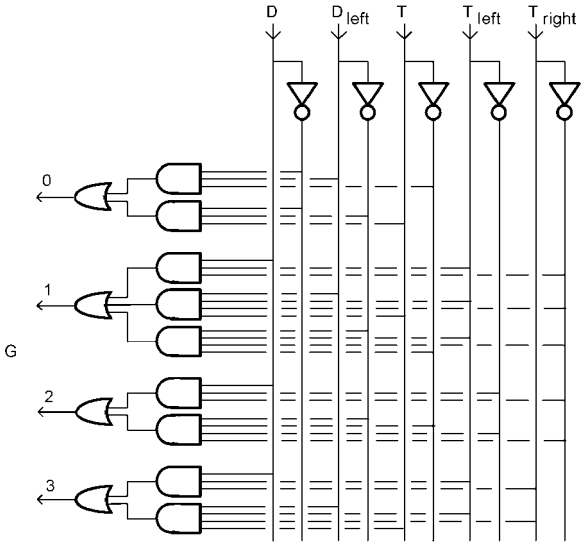

Figure 7. The selection generator.

free place on the left side or in the current cell. These signals can be obtained by using traditional comparators: the first one is generated by a majority comparator, while the second one is computed by an equality comparator (a custom design can be adopted for each specific value of N).

At the end of the propagation (floating) of each input signal, i.e., when the array contains the $2N + 1$ totally ordered samples of the current window, the median value is extracted from the cell in the middle of the array. In fact, since samples have been ordered by increasing value from left to right during the previous $2N + 1$ iterations, so the $N + 1$-th cell automatically stores the median value.

For this second architecture, the circuit complexity C_2 and the clock period are strictly related to the data transfer approach adopted between adjacent cells. We consider here the case of parallel data transfer, but the discussion can be directly extended to the bit-serial case. The area A_c of each cell is given by the sum of the areas of the following components: the sample register, the age register, the age incrementer, the sample comparator, the age comparator, the selection generator, the sample multiplexer, and the age multiplexer. It is therefore given by $A_c = 14n + 16 + 22\,ceil(log_2 N)$ [19]. The total area C_2 is $(2N + 1)A_c$.

In this semi-systolic architecture, all cells operate in parallel: thus, the total computational time of the filter, i.e., the time required to produce the median value, is equal to the computational time of the individual cell (see Fig. 5). From the structure, it is possible to obtain [19] that the computational time is approximately equal to $12 + N + 2\,max(n, 3 + 3\,ceil(log_2 N))$. Throughput T_2 is given by the inverse of the total computational time.

Concurrent error detection can be afforded as in the previous architecture by using data coding for registers and arithmetic operations, while duplication protects the control sections. In particular, we adopt the parity code for the sample register and the $3N$ code (a well-known arithmetic code [21]) for the age register. Error checking on sample data can be performed within each filter cell or only in the final filter output (i.e., on the median value). In the first case, the error is detected as soon as possible. In the second case, it may remain latent for an unpredictable time interval, since filter outputs may be codewords even if they are not the correct median values; an error may in fact modify the expected data ordering of samples moving a correctly-coded sample in the central cell. The error will be detected only if a non-codeword is moved into the central cell.

The $3N$ code protects the age register and the adder for automatic sample aging. Checking consists of verifying the divisibility by the code generator 3; to guarantee correct aging, it must be performed within each cell.

This simple structure of comparators, multiplexers, and selection generators is not suited to the application of any data coding technique for error detection. To achieve concurrent detection, we adopt modular redundancy by duplicating each unit and by comparing the outputs within each cell. If the hardware redundancy is too expensive, robust design techniques should be applied for fault avoidance: in this case, errors should be prevented, but they are not detected whenever they occur.

The floating median filter is well suited to support dynamic redundancy with reconfiguration to guarantee system survival in the presence of a limited number of faulty cells. To achieve this goal, we must introduce a switched interconnection structure among the cells so that faulty cells can be excluded from the active computation and their workload moved onto the neighbouring fault-free cells. We can tolerate as many faulty cells as the spare ones.

4. The Pipelined Polarizing Median Filter

The general structure of the *pipelined polarizing median filter* is shown in Fig. 8; it consists of three main components: the polarization generators, the

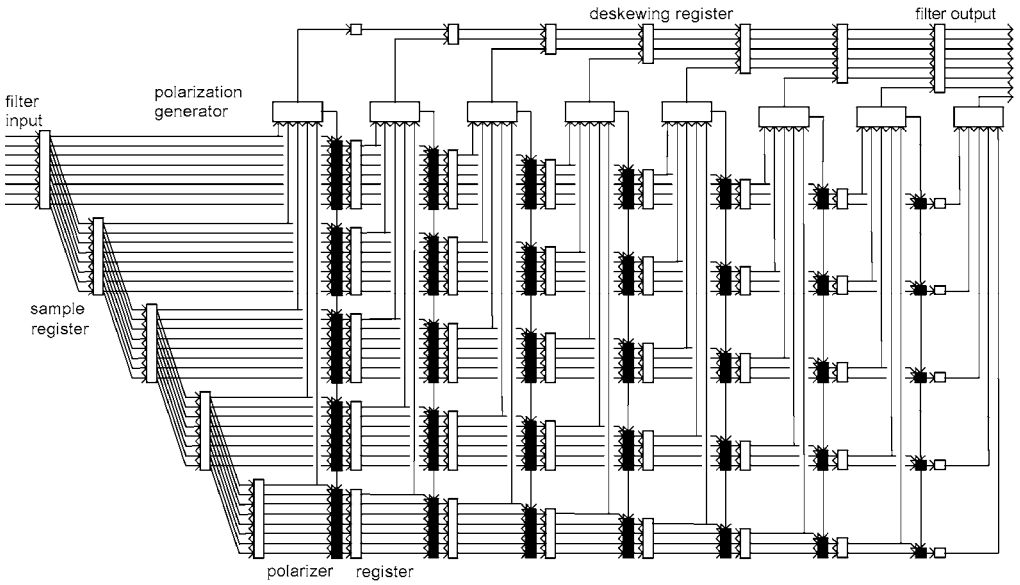

Figure 8. The pipelined polarizing median filter.

polarizers, and the registers providing correct pipelining. Input data are presented sequentially at the filter input: all the bits of the individual sample are introduced in parallel. The registers on the left side of Fig. 8 form a register file storing all the samples to be processed. It consists of $2N + 1$ master-slave registers; each of them is n bits long, where n is the number of bits of the individual sample. At each clock cycle, the data contained in the register file are shifted down by one position to free space for a new sample, i.e., to prepare the set of samples of the next window; the oldest sample is discarded at the bottom of the register file. In the structure of Fig. 8, the analysis of the samples in a window is progressively carried out in a pipelined way from the left side to the right one. A different window in the filtering algorithm is associated to each stage of the pipeline; during each clock cycle, there are n active windows within the architecture. The oldest window is on the right side of the architecture.

At each pipeline stage, the evaluation of a new bit of the expected median value corresponding to the window associated to such stage is performed. Median bits are generated starting from the most significant one. Only the sample bits, which are required by the evaluation of the subsequent least-significant median bits, are propagated. Within each stage, master-slave registers are introduced to support pipelining by holding the bit values that must be propagated: each register in the j-th stage stores $n - j$ bits.

Since only one median bit is generated in each pipeline stage, master-slave registers are introduced to provide the proper deskewing of the output values (see the top side of Fig. 8). The deskeweing register in the j-th stage has $n - j$ bits.

The inputs of the filter are progressively modified through the pipeline by applying a "polarizing" transformation. Such polarization is directed to mask the inputs whose value is far from the expected median value. At each iteration in the j-th pipeline stage, the $n - j + 1$ least-significant bits of each sample are analyzed. Operations are performed in parallel on all samples of the window; each sample is manipulated in the corresponding row in Fig. 8. At stage j, the most-significant bit of the propagated subset of sample bits (i.e., the $(n - j + 1)$-th bit of the sample) is used to evaluate the $(n - j + 1)$-th bit of the median value corresponding to the window samples currently associated with the stage j. The $(n - j + 1)$-th bit of the median value is equal to the majority value in the set of the $(n - j + 1)$-th bits of the samples.

The remaining $n - j$ least-significant bits of each sample in the j-th stage must be properly "polarized" by using the results of the median evaluation achieved till the current stage. Polarization consists of replacing actual data with a rearranged set of values, median-equivalent to the original ones: the median value is not changed if we substitute to all input samples, which are smaller than the median, the value 0 or any value

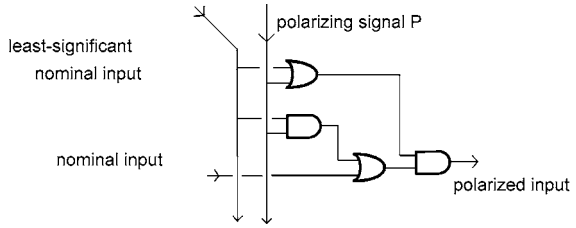

Figure 9. The one-bit polarizer cell.

which is smaller than the median itself. Similarly, it is not modified if we substitute to all input samples, which are higher than the median, the top value that can be represented with n bits or any value higher than the median itself.

In each stage, all input signals that have been recognized to be smaller than the expected median value are polarized towards the current minimum value within the window. Conversely, all input samples that appear to be higher than the expected median value are polarized to the current maximum window sample. Each actual datum which appears to be smaller than the expected median in the j-th stage is substituted by a sample having the $j-1$ most-significant bits equal to the original ones, while the remaining $n-j+1$ bits are reset to 0; then it is propagated to the input pipeline register of the $j+1$-th stage. Similarly, each actual datum higher than the expected median is replaced with a sample having the $j-1$ most significant bits equal to the original ones, while the remaining $n-j+1$ bits are set to 1.

The circuit implementing the polarization operation is shown in Fig. 9. Each polarization generator produces the new polarization P to prepare the input data for the subsequent stage. At the j-th stage, if the number of 1's in the set of the j-th bits of the polarized signals is over N, all polarized samples having the j-th bit equal to 0 are smaller than the expected median, independently of the value of their bits in position of weight smaller than j. Similarly, if the number of 0's is over N, the polarized samples having the j-th bit equal to 1 are higher than the expected median value.

The majority value of the j-th bits gives the polarizing value at iteration j. The circuit for generating the polarization value P is shown in Fig. 10; P is 0 if the number of 1's is over N, otherwise it is 1. The samples having the j-th bit equal to the minority value at the j-th iteration must be polarized: they are smaller than the expected median value if the minority value is 0; otherwise, they are higher than the median value. Polarization is achieved by setting all the bits having weight from j down to 0 equal to the complement of the polarizing value. From the previous definitions, the j-th value of the polarization signal P is equal to the complement of the j-th bit of the median value.

Circuit complexity C_3 is related to the silicon area of the components introduced above. The area used by data registers is $(2N+1)A_{ff}\, n(n+1)/2 = 7(2N+1)\, n(n+1)/2$, while the area used by deskewing registers is $A_{ff}\, n(n-1)/2 = 7n(n-1)/2$ [19]. The polarizator area is approximately given by $2n(n-1)$. Since the detailed design of the polarization selector is strictly related to the number of inputs, i.e., to N, there is no simple relationship that gives the number of gates as function of N; for example, the number of gates is 28, 38, and 63, for N equal to 2, 3, and 4, respectively. The total area C_3 is the sum of the previous components.

The computational time required to generate one median value is $n\tau_s$, where τ_s is the computational time of the individual stage; this last time is given by the sum of the time τ_{ff} for operation of a master-slave register, the delay τ_p for polarization, and the time τ_g for polarization selection [19]. τ_g is strongly related to the specific value of N, while the other components are completely independent of N. The part of τ_s which is independent of the internal structure of the polarization selector is $3+\tau_{ff} \approx 11$. The total computational time τ_s is approximately equal to 27, 27, and 33, for N equal

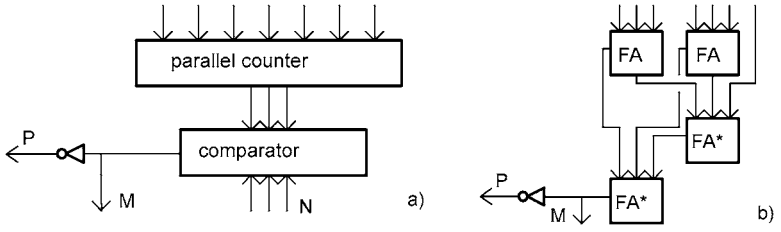

Figure 10. Polarization generator. The general structure (a). The implementation case for $N = 3$ (b): FA are full adders, FA* are full adders that generate only the carry bits.

to 2, 3, and 4, respectively. Due to the continuous sample loading during the computation, the throughput T_3 is given by $1/\tau_s$.

Concurrent error detection can be achieved by adopting a parity code as in the case of serial-input polarizing median filter, since the main structure is composed by registers [21]. Data checking is performed on the filter ouput. The parity bits are propagated towards the final output and polarized similarly to the other bits: the final parity bit coincides with the parity bit of the median value. To achieve a higher detection capability, each row of the structure can be checked separately; this provides also information for fault localization.

The high regularity of the rows is well suited to support directly the dynamic redundancy with reconfiguration to exclude the faulty row from the computation. Graceful degradation of filtering (i.e., width reduction of the filtering window) can be adopted when the hardware redundancy has been fully exploited.

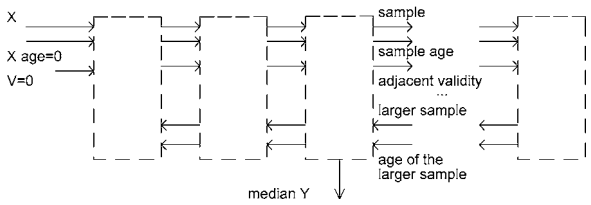

Figure 11. The array architecture of the pipelined sorting median filter.

5. The Pipelined Sorting Median Filter

The last architecture we propose is the *pipelined sorting median filter*. This architecture is based on a linear systolic array of $2N+1$ cells: Fig. 11 shows the overall array structure, while Fig. 12 gives the internal structure of the basic cell. In some sense, it is the pipelined version of the floating median filter. The j-th cell consists of an input register X holding the sample coming from the cells on the left side, a sample register S holding the j-th sample of the previous window in increasing order, an age register A to store the age of the sample currently stored in the register S, an aging circuit for updating the age of the sample S automatically, a selection generator G performing the sorting operations of samples, and the multiplexers S_{MUX}, X_{MUX}, A_{MUX}, and A^t_{MUX} to propagate the sorted samples through the array.

Input samples enter the filter from the left side of the linear array. The basic idea is to allow a wavefront propagation of window samples from the left to the right side of the array so that, at the end of propagation, all samples are rearranged in increasing order. Propagation is viewed as a sprint race among athletes starting from the left side of the array: speedier athletes have higher values. The speediest one is the first one that arrives at the rightmost cell; a photograph of the athletes, taken exactly when the speediest one arrives at the right side, gives the actual distribution of athletes according to their relative capabilities.

In our array, samples are compared one pair at a time. The sample of the pair having higher value "runs"

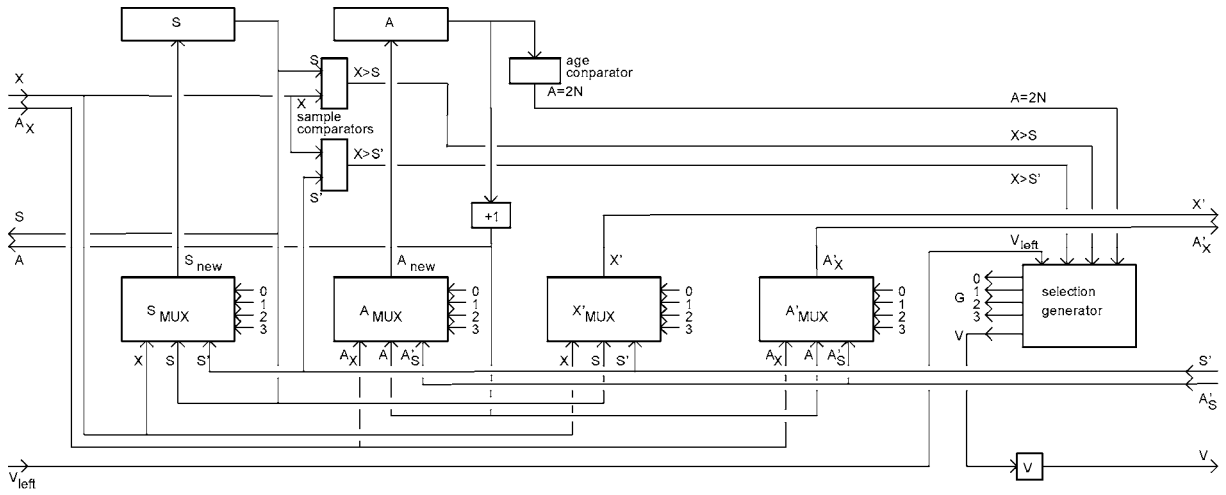

Figure 12. The basic cell for the pipelined sorting median filter.

faster and thus arrives before the other one at the adjacent cell on the right side. The race terminates when all comparisons have been performed among the samples in the same window. The winner is the runner who scores the midpoint arrival classification. This algorithm is similar to the classical "bubble sort" algorithm, where movement of samples is always in the same direction.

Pipelining can be applied since comparison are performed from left to right within the linear structure, and since the $2N$ samples inherited from the previous window have been already ordered. At each new window, the oldest sample of the previous filtering window is discarded by applying an automatic aging mechanism and a new sample enters the array.

Each cell holds a sample value in the register S: at each iteration, it is compared with the new input datum X. If X is higher than the value stored in the cell, it is moved to the adjacent cell on the right side; otherwise, it is stored in the considered cell. After $2N$ comparisons, the window samples are placed in the proper increasing order.

The above mechanism could operate correctly if the $2N$ previous samples were placed in the $2N$ leftmost cells of the array without gaps, i.e., if the oldest sample of the previous iteration is stored in the rightmost cell.

Unfortunately, this is not always the case. The oldest sample of the previous window (the one that must be discarded in the current window) may be in any position within the array. Thus, at the beginning of the computation for the current window, there may be an empty cell in any position. When a sample reaches such a cell, there is no sample stored in the cell itself to perform the comparison and propagation steps described above. We cannot store the input sample in the free cell since such a sample has not yet been compared with the ones placed in the subsequent cells; besides, the input sample cannot be compared with the one stored in the adjacent cell on the right side since the comparison with the input sample is still in progress.

To solve this timing problem correctly, we adopt a two-phases clocking management for the whole array architecture. During odd phases, the cells at odd position $(1, 3, \ldots, 2N + 1)$ execute their nominal operations, while the cells at even position $(2, 4, \ldots, 2N)$ are standing by, and vice versa. Filter inputs must be presented at the leftmost cell of the array only during odd cycles. The alternate operation of odd and even cells appears also in the X registers: each of them stores a valid value only at the beginning of its operational cycle. To guarantee a correct propagation of data, the input registers X and the sample registers S are master-slave.

If the sample stored in the register S is not the oldest one of the previous window, the cell compares the input sample X with the one stored in the register S. If X is higher than S, it is moved to the next cell, while S is still held in the cell itself. Otherwise, S is moved to the right cell with its age stored in the register A; X and its age are stored in the cell.

When a cell is free due to over-aging of the stored sample, the cell on the right side is not operating and its input register X (from now on, it will be called X^t to distinguish it from the register X of the considered cell) is empty. Moreover, the sample stored in register S of the adjacent cell on the right side (from now on, it will be called S^t) is correctly ordered. Therefore, the sample X can be compared directly with S^t.

Samples aging is automatically maintained in the register A in order to discard old samples and to free space for the incoming ones. The age of a new input sample is set equal to 0; an age equal to $2N$ identifies the oldest sample of the window. To guarantee a correct propagation of data, the age registers A are master-slave. Aging is performed by using an increment circuit, implemented as a ripple-carry adder (composed by half adders) with least-significant input carry set to 1.

In each cell, the flip-flop V is used to validate the sample stored in the register S of the adjacent cell on the right side; it is equal to 0 if S is valid, otherwise it is 1. It is used to invalidate the content of the register S in the cell on the right side after that its value has been used to fill in the register S in the current cell.

To control the data flow, we use a modular distributed structure. It is based on the sample multiplexer S_{MUX}, on the age multiplexer A_{MUX}, on the propagated sample multiplexer X_{MUX}, on the propagated age multiplexer A^t_{MUX}, and on the selection generator G. These multiplexers can be implemented, for example, by using the circuits shown in Fig. 13.

The sample multiplexer S_{MUX} selects the new value that must be stored in the register S among the three possible sources according to the selection signal $G(X, S,$ or $S^t)$. The age multiplexer A_{MUX} selects the new value for the age register according to the signal G: the new value is the one associated with the new sample for S (the age A_X of the input sample, the age A of

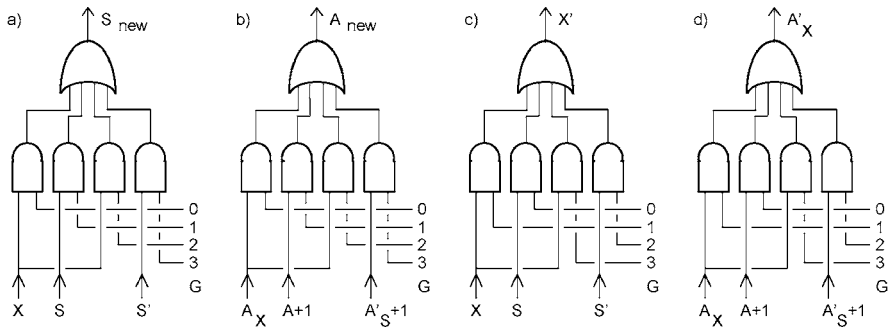

Figure 13. The basic cells of the multiplexers for one bit selection: S_{MUX} (a), X_{MUX} (b), X^t_{MUX} (c), and A^t_{MUX} (d).

the sample S previously stored in the cell, or the age $_S$ of S). The propagated sample multiplexer X_{MUX} A^t ^{t} selects the sample value X^t that must be propagated to the cell on the right side (X, S, or S^t). The propagated age multiplexer A^t_{MUX} selects the age value A^t_X that must be associated with X^t(A_X, A, or A^t_S).

The selection generator G provides the control signals to execute data sorting and propagation (see Fig. 14). It computes a signal G, stating which datum must be stored and which one must propagated, and the validity signal V. If the incoming validity signal V_{left} is 0 and if A is over $2N$, G is equal to 0 if X is not higher than S: X is stored in the cell, A_X is stored in the age register, S and $A + 1$ are propagated to the cell on the right side, and V is set to 0. Under the same conditions for V_{left} and A, G is equal to 1 if X is higher than S: S is stored again in the cell, $A + 1$ is stored in the age register, X and A_X are propagated to the cell on the right side, and V is 0.

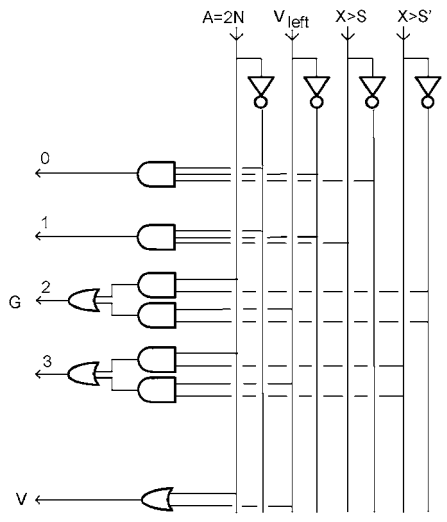

Figure 14. The selection generator.

If V_{left} is 1 or if A is equal to $2N$, the sample value stored in the cell is not valid. If X is not higher than S^t, G is set equal to 2: X is stored in the cell, A_X is stored in the age register, S^t and $A^t_S + 1$ are propagated to the cell on the right side, and V is set to 1. If X is higher than S^t, G is set equal to 3: S^t is stored in the cell, $A^t_S + 1$ is stored in the age register, X and A_X are propagated to the cell on the right side, and V is set to 1.

At the end of the propagation of each input signal, the median value is extracted from the cell in the middle of the array. Since samples are totally ordered, the $N + 1$-th cell stores the median value. The extraction of the result can be performed at the end of the N-th odd clock cycle, i.e., just before the N-th even cycle.

This architecture can be greatly simplified. To identify the median value, we do not need to know exactly the complete ordered sequence of samples; we need only to know which samples are smaller than the median and which ones are higher. Besides, we only need to know the total ordering of samples in one of these two subsets (e.g., in samples smaller than the median value). After identifying of the samples which are smaller than the median and after detecting the median, sorting of the remaining higher-valued samples can be avoided. In fact, assume first that no sample smaller than the median value has reached the maximum age: if the new sample is greater than the median value of the previous window, it will run over the position in the middle of the array (the median value is identical to the previous one). If the new sample is not higher than the previous median value, it will be placed in the proper ordering in the first $N + 1$ positions: all values higher than the median one in the previous window are not required to

generate the new median value. Consider now the case of a "dead" sample due to the aging mechanism in the first $N+1$ positions (i.e., a free place in the left part of the array); only the new sample and the samples still living in the $N+2$ leftmost cells will be involved in the process of total ordering to identify the new median value: samples in the $N-1$ rightmost cells do not influence the generation of the new median value since they are obviously higher than any other sample in the system.

Therefore, the systolic array can be truncated to the $N+2$ leftmost cells: the N leftmost cells store the ordered subset of samples having values smaller than the median, while the $N+1$-th cell stores the median value. The $N+2$-th cell may contain the smallest value which is higher than the median of the previous window; it is required for the correct operation of the $N+1$-th cell.

The circuit complexity C_4 and the computational time can be derived by analyzing the structure of the single cell. The area A_c of each cell is given by the sum of the area of the following components: the input register X, the input age register A_X, the sample register S, the age register A, the validity flip-flop V, the age comparator C_A, the sample comparator C_{XS}, the right sample comparator C_{XS^r}, the cell age incrementer, the selection generator G, the sample multiplexer S_{MUX}, the age multiplexer A_{MUX}, the propagated sample multiplexer X_{MUX}, and the propagated age multiplexer A^r_{MUX}. The area A_c is given by $47n+34+14\ ceil\ (log_2 N)$ [11]. The circuit complexity C_4 is thus proportional to $(N+2)A_c$.

Cells operate in parallel according to the two-phase clock discussed above: the total computational time of the filter is equal to $(N+1)\tau_c$, where τ_c is the computational time of the individual cell. From the structure, the computational time τ_c of the cell is approximately equal to $5n+7$ [19].

The clock period for input presentation and the total throughput T_4 can be derived from the analysis of the array computation in the truncated version: the first one is $2\tau_c$, while the second one is $1/2\tau_c$.

As in the case of the floating median filter, concurrent error detection can be achieved by using data coding (namely, the $3N$ code [21]) for registers and arithmetic operations, while duplication protects the control sections. Error checking is performed within the individual cell so that direct fault localization is possible. The high modularity of the array structure is well suited to support dynamic redundancy in order to guarantee system survival even in the presence of a limited number of faulty cells.

6. Comparative Evaluation and Concluding Remarks

In this paper, we have presented four partially new architectures for digital median filtering: their characteristics have been evaluated from the point of view of computational capabilities (latency and throughput) and circuit complexity. Fault tolerance features have also been addressed by considering, in particular, data coding and concurrent error detection for continuous output validation. A detailed evaluation of the costs of the proposed architectures is given, thus giving a more refined comparison than the one presented for instance in [4].

In [14] a median filter architecture is presented, containing the basic principle used by the polarizing median filter proposed in this paper: to set or reset the sample bits if the samples they belong to are retained/discarded as median value candidates. It seems to be the architecture closest to our polarizing median structure. This architecture is serial at the bit level and could be used in a way similar to our polarizing median filter structure. There are differences, anyway: the number of cells in the array of architecture [14] equals n (the number of bits of the sample), whereas it equals N (the width of the filter window) for our polarizing median filter structure. In architecture [14] one sample of n bits can be input every clock cycle; in our structure inputting one sample requires n clock cycles. Therefore, in the former case the throughput (measured in output median values over time) equals the clock frequency, whereas in the latter case it equals the clock frequency divided by n.

Architecture [14] is systolic, whereas our structure is semi-systolic (though in a very limited way), since there is one signal (the polarization signal) that is broadcast to all the cells simultaneously. Two adjacent cells of architecture [14] exchange $2N$ bits, whereas two adjacent cells of our structure exchange a constant number of bits (2 bits, actually), independently of N. This also means that a cell of architecture [14] contains a number of storage elements (flip-flops) different from a cell of our structure: in fact, the cell of architecture [14] contains $3N$ storage elements, whereas the cell of our structure contains $n+2$ storage elements. Hence, the array of architecture [14] contains $3nN$ storage elements,

whereas our solution contains $N(n+2)$ storage elements. Of course, the cells of both structures contain combinatorial logic; however, in the cell of architecture [14] the size of this combinatorial logic is proportional to N, whereas in our structure it is constant, independently of N. Of course, our structure requires one parallel counter with N inputs, a type if combinatorial network for which very optimised solutions exist [20].

Being semi-systolic, our structure exhibits a higher fan-out for some signals than architecture [14]: as a matter of fact, however, this involves only the one-bit polarization signal, that is broadcast to N cells. By all the above considerations, we envision the polarizing median filter as an architecture expecially optimized for the cases not requiring extremely high throughput. Note also that if very high throughput is not required, broadcasting the polarization signal to all cells simultaneously is not a limiting factor.

Other solutions based on the usage of semi-analog majority gates [15] have been proposed as well, sharing some ideas with the polarizing structures described in this paper; however, these ones are fully digital, similarly to the sorting/floating ones, and therefore can be easily scaled and pipelined.

Some sorting-based algorithms, for instance among those illustrated in [4], are close to the floating/sorting architectures illustrated in this paper; however, we develop detailed VLSI architectures, giving a particular attention to the introduction of architectural properties like scalability and the possibility to pipeline the structure, in order to speed up the clock frequency.

In [17] an architecture is presented, based on the sorting of the samples, and seeming to be close to the the floating median filter proposed in this paper. It is based on the idea of maintaining an ordered window of N samples, each one stored in one array cell; at every clock cycle the oldest sample is removed and a new sample is inserted in the window. To achieve this result, the oldest sample value is made available to all the cells in the array, and it is compared to the actual sample stored in each cell. In this way, the cells need not maintain the age of the samples to discard the oldest one (this method works because the sample window is ordered, as it is proved in [17]), as the floating median filter presented in this paper does. However, the architecture developed to full detail in [17] is systolic, whereas the floating median filter presented in this paper is semi-systolic, since the incoming sample is broadcast to all the cells of the array simultaneously. Hence, a direct comparison seems hard to be done.

In Fig. 15, we present a comparative evaluation of the basic characteristics discussed in the previous sections; in particular, we give the typical shapes of circuit complexity and throughput for a traditional design of the basic units [19]. The circuit complexity is measured in a first approximation as the number of gates of the circuit; the actual value could be obtained by multiplying such measure by the typical gate area (including the routing area). The throughput is the inverse of the time interval between two subsequent median values. The time interval is estimated as the number of gate levels, while its actual value could be derived by multiplying this measure by the gate switching time. Similarly, the actual throughput could be obtained by multiplying the values given in Fig. 15 by the inverse of the gate switching time, or by the gate switching frequency.

The pipelined architectures have higher throughput with respect to the first two solutions, but they are rather expensive from the point of view of circuit complexity. In each pair of homogeneous structures, one solution has smaller circuit complexity but smaller throughput than the other one. Therefore, a careful selection of the system architecture must be taken into account for the specific application envisioned: the analysis of Fig. 15 provides the basic guidelines to design the optimum solution.

Results achieved in this paper for the case of non-recursive median filtering can be directly extended to the case of recursive median filtering [1, 2]. In the latter case, the computed median value is not only delivered as external output of the filter, but is also recirculated in the middle of the window. Recirculated samples are treated as the nominal ones. This architectural extension can be simply achieved by modifying the filter interconnections to support output recirculation.

Extension to the case of multidimensional filtering [1, 2] can be basically reduced to the presentation of d-dimensional windows of input data to the filter; on each set of data, the median filter must extract the median value associated with the center of the d-dimensional window. The basic architectures here presented can be extended to such filtering by adjusting the size of the working data set, i.e., by changing the number of the input signals and of the sample data on which the filter operates. The modularity of our architectures allows to deal with these cases as well.

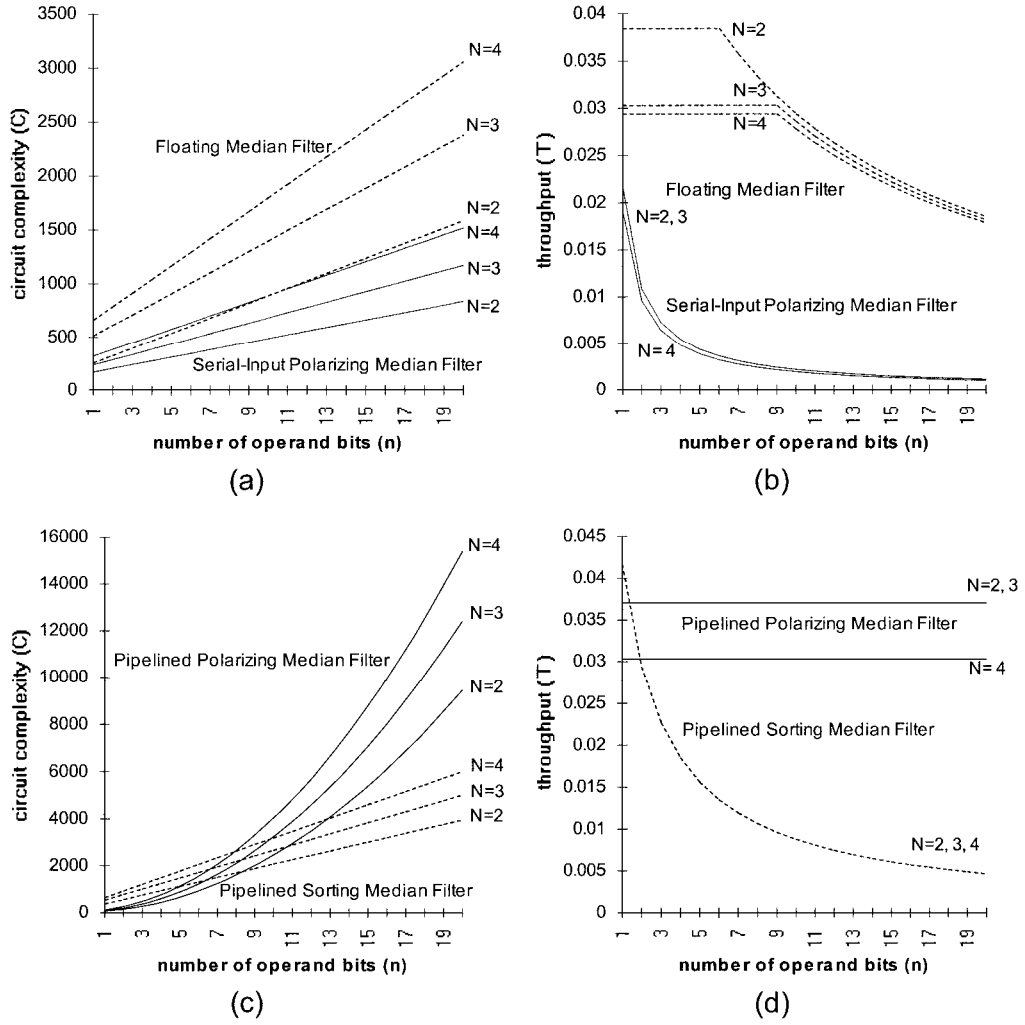

Figure 15. System evaluation: Circuit complexity (C_1 and C_2) of the non-pipelined architectures vs. the number of operand bits (a), throughput (T_1 and T_2) of the non-pipelined architectures (b), circuit complexity (C_3 and C_4) of the pipelined architectures (c), throughput (T_3 and T_4) of the pipelined architectures (d). Characteristics are given for windows containing 5, 7, and 9 samples, respectively (i.e., for $N = 2, 3, 4$).

Acknowledgment

The authors are grateful to the anonymous referees for providing comments and suggestions that greatly helped in improving this paper.

References

1. W. Pratt, *Digital Image Processing*, New York, USA: John Wiley & Son, 1978.
2. T.S. Huang (ed.), *Two-Dimensional Digital Signal Processing II: Transforms and Median Filters*, New York: Springer Verlag, 1981.
3. N.C. Gallagher and G.L. Wise, "A Theoretical Analysis of the Properties of Median Filters," IEEE *Transactions on Acoustic, Speech and Signal Processing*, vol. ASSP-29, 1981, pp. 1136–1141.
4. D.S. Richards, "VLSI Median Filters," *IEEE Transactions on Acoustic, Speed and Signal Processing*, vol. 38, no. 1, Jan. 1990, pp. 145–153.
5. *Proceedings of the 1992 IEEE International Symposium on Circuits and Systems*, San Diego, CA, USA, May, 1992.
6. *Proceedings of the 1992 IEEE International Workshop on VLSI Signal Processing*, Napa Valley, CA, USA, Oct. 1992.
7. S.Y. Kung, *VLSI Array Processors*, Englewood Cliffs, NJ, USA: Prentice Hall, 1989.
8. L.E. Lucke and K.K. Parhi, "A New VLSI Architecture for Rank Order and Stack Filters," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1992.
9. L.E. Lucke and K.K. Parhi, "Parallel Structures for Rank Order and Stack Filters," in *Proceedings of the IEEE International*

Conference on Acoustic, Speech and Signal Processing, 1992.
10. M. Karaman, L. Onural, and A. Atalar, "Design and Implementation of a General Purpose Median Filter in VLSI," in *VLSI Signal Processing III*, IEEE Press, 1988.
11. P.D. Wendt, E.J. Coyle, and N.C. Gallager, "Stack Filters," in *IEEE Transactions on Signal Processing*, 1986.
12. G.R. Arce and P.J. Warter, "A Median Filter Architecture Suitable for VLSI Implementation," in *Proceedings of the 23rd Annual Allerton Conference of Communication Control Computing*, Oct. 1984, pp. 172–181.
13. A.L. Fisher, "Systolic Algorithms for Running Order Statistics in Signal and Image Processing," *Journal of Digital Systems*, vol. 4, no. 2/3, 1982, pp. 251–264.
14. L.W. Chang and J.H. Lin, "A Bit-Level Systolic Array for Median Filter," *IEEE Transactions on Signal Processing*, vol. 40, no. 8, 1992, pp. 2079–2083.
15. C.L. Lee and C.W. Jen, "Bit-Sliced Median Filter Design based on Majority Gate," *IEE Proceedings on Circuits, Devices and Systems*, vol. 1391, 1992, pp. 63–71.
16. E.H. Lu, J.Y. Lee, and Y. Yang, "VLSI Architecture for Median Filters with Linear Complexity," *TECNON '96, Proceedings on Digital Signal Processing Applications*, vol. 1, 1996, pp. 358–362.
17. R. Roncella, R. Salemi, and P. Terreni, "70-MHz 2μm CMOS Bit-Level Systolic Array Median Filter," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 5, 1993, pp. 530-536.
18. E. Ataman, V.K. Aatre, and K.M. Wong, "A Fast Method for Real-Time Median Filtering," *IEEE transactions on Acoustic, Speech and Signal Processing*, vol. ASSP-28, no. 4, 1980, pp. 415–420.
19. *CMOS Macrocell Array Design Manual*, SGS-Thomson Microelectronics, Italy, 1990.
20. K. Hwang, *Computer Arithmetic*, New York, USA: John Wiley & Sons, 1979.
21. T.R.N. Rao, *Error Coding for Arithmetic Processors*, New York, USA: Academic Press, 1988.

Luca Breveglieri received the Dr. Eng. degree in electronic engineering and the Ph.D. degree in electronic engineering of information technology and systems from Politecnico di Milano, Italy, in 1986 and 1992, respectively.

From 1991 to 1998 he has been a Computer Technician and a part-time Researcher at the Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano (PdM). Since 1998 he is Associate Professor of Computer Science at Politecnico di Milano.

While a computer technician he has worked as a computer systems and networks manager, administering the computers and the local area network of DEI. His research interests include architectures of computing systems and networks, application specific VLSI synthesis for digital signal/image processing and cryptography, computer arithmetic, and formal languages theory.

Luca.Breveglieri@Elet.PoliMI.IT

Vincenzo Piuri obtained the Ph.D. in Computer Engineering in 1989, at Politecnico di Milano, Italy. From 1992 to September 2000, he was Associate Professor in Operating Systems at Politecnico di Milano. Since October 2000 he is Full Professor in Computer Engineering at the University of Milano, Italy. He was Visiting Professor at the University of Texas at Austin during the summers from 1993 to 1999.

His research interests include distributed and parallel computing systems, computer arithmetic, application-specific processing architectures, digital signal processing architectures, fault tolerance, neural network architectures, theory and industrial applications of neural techniques for identification, prediction, control, signal and image processing. Original results have been published in more than 150 papers in book chapters, international journals, and proceedings of international conferences.

He is Fellow Member of the IEEE and member of ACM, INNS, AEI. He is Associate Editor of the IEEE Transactions on Neural Networks, the IEEE Transactions on Instrumentation and Measurement, and the Journal of Systems Architecture.

piuri@dti.unimi.it