New Designs for Voting Networks Based on the Boyer-Moore Majority-Finding Algorithm

Behrooz Parhami

Department of Electrical & Computer Engineering
University of California, Santa Barbara, USA
parhami@ece.ucsb.edu

***Abstract*—Replication and voting is a key redundancy scheme for obtaining highly-reliable computation results from unreliable components. Both hardware voters (voting networks) and software voters (voting algorithms) have been studied for many decades and trade-offs in their designs have been investigated. For voting networks, the circuit complexity grows superlinearly with the number of inputs, making the design of large voters quite challenging, given that failures in the voter hardware tend to be critical. In this paper, we show how to use the linear-cost, logarithmic-parallel-delay Boyer-Moore algorithm to design efficient and scalable voting networks.**

Keywords*— *fault-tolerant computing, majority, replication, supermajority, voter, voting algorithm, voting network

I. Introduction

Reliable computing has been of interest since the 1960s, when NASA started planning long-duration space missions, during which repair would be impossible for years, and the US defense establishment became interested in increasing the robustness of electronic devices used in command-and-control systems. NASA used teams of computer scientists/engineers to design hardware & software for spacecraft control systems and initiated a major reliable computing project at JPL [1]. The US Defense Department awarded, through ARPA, multiple R&D contracts for design and proof-of-concept projects [2] [3]. Interest in fault-tolerant computing has continued ever since.

Throughout the years, one strategy for improving computer system reliability has been by means of replicating the computational elements and combining their outputs through voting [4]. Early on, hardware was quite expensive, bulky, and power-hungry, so 2-out-of-3 voting was dominant. The thinking was that if 2 of the 3 computational elements produced the same result, with the third one being in disagreement, then the agreeing outputs are much more likely to be correct. Later, larger replication factors were contemplated, which made the probability of the majority result being incorrect even smaller.

Recently, the interest of researchers in fault-tolerant computing has expanded to voters with dozens of inputs, given highly unreliable nanoelectronic and biologically-inspired components needing larger replication factors for reliable operation [5] [6]. Some previously-studied voter designs have complexities that rise very quickly with the number of inputs, perhaps exponentially. It is thus imperative to revisit the problem of designing voting algorithms and networks in order to discover new designs with better asymptotic latencies, as well as more economical hardware realizations (including VLSI-friendliness) and energy economy.

Use of majority voters to build reliable computers from marginal components dates back to the 1950s [7]. More recently, majority voting has been used to protect against errors from radiation-caused bit-flips [8] and to design long-life and safety-critical computers based on replication [9]. As previously mentioned, the redundancy factor, limited to 3-6 in the past, is moving up, owing to variability & imperfections, and thus high error rates, in nanoelectronic circuits. Redundancy factors of several-dozen have already been proposed and even higher redundancy may be needed with further circuit shrinkage and the accompanying rise in variabilities and uncertainties [5].

We have previously devised several methods for designing arbitrarily large voting networks [10] and proposed recursive designs for majority bit-voters [11], including their realization via 2D arrays of multiplexers [12]. Supermajority voters (e.g., 4-out-of-5 or 6-out-of-9 voter), which can be designed through the same design strategies as simple majority voters, can offer significantly higher reliabilities.

II. Types of Voting Schemes

We are interested in finding the majority element among n inputs, x_1, x_2, … , x_n. We take n to be odd for clarity and simplicity of our presentation, although many of our results can be readily generalized to an even number of inputs.

A. Bit-voting

A bit-voter implements a symmetric function of its n input bits, with the output being 1 whenever at least $(n + 1)/2$ of the inputs are 1s. Thus, a straightforward way of implementing a bit voter is to use a parallel counter [13] to count the number of 1s among the inputs, followed by a threshold comparator. Any threshold voter can be implemented in the same way.

In bit-voting, determining the majority value is equivalent to median-finding. There are numerous methods for finding the median value in a set of n inputs in hardware. One is using a selection network [14], which tends to be simpler and faster than a sorting network. It is well-known that for software median-finding schemes, $O(n)$ comparisons suffice, whereas the sorting lower bound is $O(n \log n)$ comparisons. Of course, lower bounds don't always translate to practical designs. Pippenger's median-finding algorithm needs $3n$ comparisons and is practically realizable. For large values of n, the best practical sorting networks are of size $O(n \log^2 n)$ and of latency $O(\log^2 n)$. For example, two designs due to Batcher [15] [16] both result in fairly efficient sorting networks.

B. Exact word-voting

Word-voting refers to a process by which the majority value among n input words is determined. We first proceed with the assumption that the input words represent exact values, say, k-bit integers. Subsequently, we will also discuss inexact voting, as in the case of floating-point computation results which may turn out slightly different when performed on diverse hardware or with diverse algorithms. Even with identical resources, floating-point results may exhibit slight differences when compiler or hardware instruction-scheduling optimizations kick in.

Assume that the inputs are k-bit integers. A naïve voting scheme would use bit-voting on each of the k bit-positions. If a majority does exist among the n inputs, this naïve voting scheme will indeed produce the majority value. Let the majority value have a bit value m_i in bit-position i. Then, m_i will be the majority value in that bit-position. If a majority does not exist among the n inputs, then bit-voting will produce an arbitrary value that may have no relation to the inputs. Here is an example.

$x_1 = 110$
$x_2 = 000$
$x_3 = 110$
$x_4 = 101$
$x_5 = 001$

The input data above clearly lacks a majority value, as the most-repeated input value occurs twice. Bitwise voting produces 100, which isn't even one of the inputs. In other words, bitwise majority voting can yield a result that has zero support. To reiterate, bitwise voting on exact values produces a valid result only if a majority value does exist.

The Boyer-Moore algorithm can also be used to obtain the majority element and yields the correct majority if it exists; otherwise, it will yield one the inputs which may not even be one that appears frequently among the inputs. The algorithm scans the input list once ($O(n)$ operations), keeping a single candidate for the majority with its associated count. The count isn't the actual number of times that the element appears among the inputs but a modified count. If we encounter two unequal input elements as we scan the inputs, we discard both of them as candidates. Because a majority element has at least $(n + 1)/2$ occurrences of which at most $(n - 1)/2$ will be discarded when matched against non-majority elements, a representative of the majority element will survive to the end of the algorithm.

Here is a more detailed justification for Boyer-Moore algorithm's correctness. Let us put the n inputs $x_1, x_2, \ldots, x_n$ into two bins. Bin 1 contains the elements that are in the majority and Bin 2 contains all the other elements. If a majority does exist, Bin 1 will contain at least $(n + 1)/2$ elements. When we remove from consideration the two elements x_i and x_j, with $x_i \neq x_j$, these two unequal elements are either in the minority bin or in different bins. In either case, the majority bin will still hold the majority element among the remaining inputs. Repeating this process will discard all the minority elements, leaving one or more elements in the majority bin.

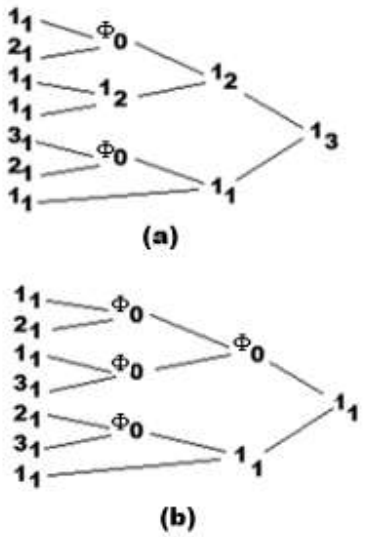

Fig. 1. Examples of applying the Boyer-Moore algorithm to 7 inputs in two cases where there is a majority and when there is no majority.

Figure 1 shows two examples of applying the Boyer-Moore algorithm to the input lists (1, 2, 1, 1, 3, 2, 1), which has the majority element 1, and (1, 2, 1, 3, 2, 3, 1), which contains no majority element. In the case where a majority does not exist, the order of comparisons will dictate which of the n input elements emerges as output, whereas when a majority value does exist, the order and groupings of comparisons will have no effect on the result obtained. The notation x_h in Fig. 1 means that the value x has h occurrences remaining, given the comparisons thus far. The symbol Φ stands for the null value that is not equal to any input value.

C. Approximate voting

When the inputs are approximate values, such as floating-point computation results, we need to redefine the notion of majority. Let us say that two approximate values agree if their difference is no greater than ε. In the following example, five approximate values are shown and we assume ε = 0.01.

$x_1 = 2.010$
$x_2 = 2.102$
$x_3 = 2.020$
$x_4 = 2.122$
$x_5 = 2.022$

Note that approximate equality is not transitive: $2.010 \cong 2.020$ and $2.020 \cong 2.022$ according to the threshold $\varepsilon = 0.01$, but 2.010 and 2.022 are not approximately equal according to the same threshold, because their difference 0.012 exceeds ε.

Different approximate voting algorithms have been proposed [17] [18]. For example, if we take the median of the five values above as the voting result, we get 2.022. On the other hand, if we consider 2.102 and 2.122 as outliers (compared to the other three values) and discard them, then the voting result is obtained based on the three remaining inputs 2.010, 2.020, and 2.022. Choosing the median of the latter three values yields 2.020 as the result, whereas averaging the values produces the result 2.017.

Any comparison-based exact word-voter can be converted to an approximate voter once we define the notion of approximate equality and replace all the comparators in the exact voter with approximate comparators. For example, defining approximate equality by choosing 0.02 as our ε, the values 2.010, 2.020, and 2.022 are deemed approximately equal. A comparison-based exact word-voter would produce one of the three or more equal values as the output. Here too, any of the three values just listed can emerge as the voter output. If this scheme is acceptable, then for designing approximate word-voters, we need to worry only about the design of an approximate comparator with a suitably chosen comparison threshold.

III. Design of Bit-Voters

A. Prior designs

Early bit-voter designs for multi-channel computation were limited to $n = 3$ or $n = 5$, given the prohibitive cost of hardware replication. The first study of voting network design for larger values of n considered four different design strategies: Gate-level design from logic expressions, arithmetic-based design as outlined at the beginning of Subsection II.A, multiplexer-based design, and selection-network-based design [10]. The study compared the gate-count complexity of the four design approaches for n up to 16, concluding that (see Fig. 2):

- Gate-level design is competitive only for $n < 5$
- Multiplexer-based design is best for $n \leq 7$
- Selection-network-based design is best for $n \geq 7$

Arithmetic-based design, though almost never competitive in terms of gate complexity and thus in custom VLSI realization, does have features that make it attractive for being mapped onto FPGAs and for certain application domains. Figure 3 depicts an example, which is a 6-out-of-8 supermajority voter. The 8 inputs are added to form a 4-bit number z in [0, 8], which is then compared against 6: z is greater than or equal to 6 if its most-significant bit z_4 is 1 or else both of its middle bits z_3 & z_2 are 1s.

Numerous researchers have proposed ad-hoc voting network designs to satisfy the needs of particular applications. Because they are optimized for a particular application context, such designs tend to be quite efficient, but they lack generalizability and scalability.

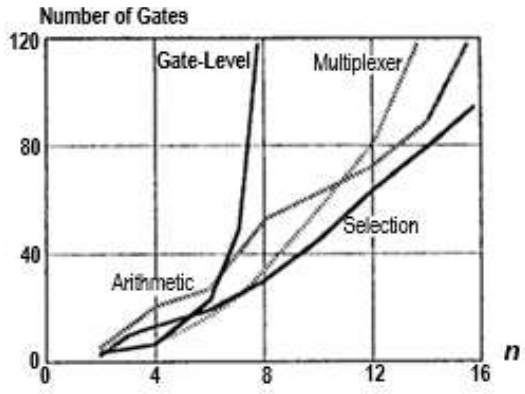

Fig. 2. Results of an early study [10] of voting network designs.

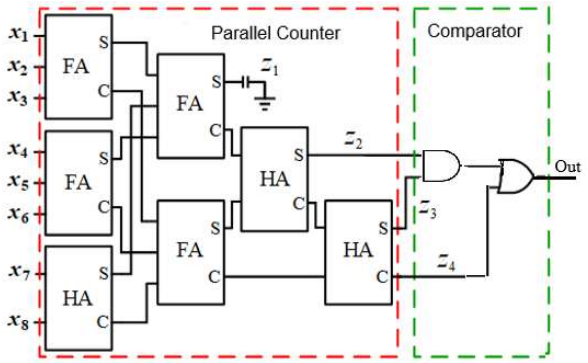

Fig. 3. Arithmetic-based design for a 6-out-of-8 supermajority voter. The parallel counter part is composed of full-adders (FAs) and half-adders (HAs).

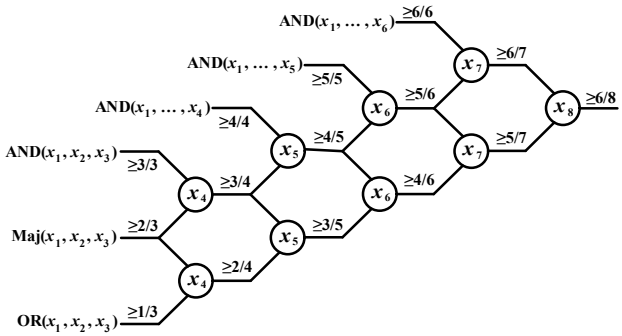

Fig. 4. Recursive mux-based design for a 6-out-of-8 supermajority voter.

B. Modern recursive (mux-based) designs

Mux-based design was introduced in [10] but was more systematically studied much later in the framework of a recursive design strategy [11]. Figure 4 depicts a 6-out-of-8 supermajority voter built out of 2-way multiplexers (shown as circles in the diagram) and a number of peripheral gates. The gates can be realized by multiplexers, making the circuit a regular 2D array of multiplexers, which, given the simplicity and efficiency of CMOS multiplexers, is quite efficient as well as VLSI-friendly in today's prevailing technology [12].

To understand the design shown in Fig. 4, focus on the right side of the diagram and the mux controlled by the input x_8, which is written inside the circle. That part of the design can be interpreted as follows: To have at least 6 of the 8 inputs equal to 1, we need 6 of the remaining 7 inputs to be 1s when $x_8 = 0$ or 5 of the remaining 7 inputs to be 1s when $x_8 = 0$. The recursion is then unrolled all the way until we get expressions such as 3-out-of-3 (3-input AND gate), 1-out-of-3 (3-input OR gate), and other basic building blocks.

It is easy to see that such recursive designs have $O(n^2)$ circuit complexity and $O(n)$ delay, when fully unrolled [11]. We can end the recursion early by using larger building blocks, such as pre-designed small voters, but the asymptotic cost and delay formulas will not change as a result. Even though these designs are not competitive with some earlier designs in an asymptotic sense, their suitability for VLSI realization and for pipelined operation make them of interest for practical values of n. The interest in these designs is akin to our interest in array multipliers, which tend to be slower than tree multipliers, but win on account of regularity and other VLSI attributes.

C. Designs based on the Boyer-Moore algorithm

We discussed the Boyer-Moore majority algorithm in Subsection II.B. Intuitively, bit-voters designed based on this algorithm will not be competitive with the best prior designs (mux-based and selection-based, as discussed in connection with Fig. 2). More on this conclusion later.

Take the example in Fig. 5, which shows the correct identification of 1 as the majority value among 7 inputs (conventions and notation were previously described in connection with Fig. 1). Three of the four 1s at input are cancelled out by the three 0s, with the fourth 1 prevailing at output. A majority always exists in bit-voting, so we don't need to worry about verification of the result.

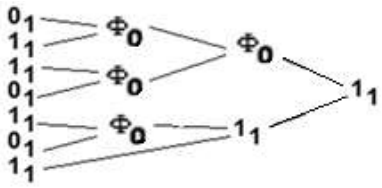

Fig. 5. The Boyer-Moore majority algorithm applied to 8 input bits.

If we are interested in super-majority voting, then we first find the majority element and then use a parallel counter to verify its super-majority status.

The hardware realization of Fig. 5 will need bit comparators (XOR gates) and adder/subtractors for combining weights. It is the need for these adder/subtractors that makes the design non-competitive. In the case of Fig. 5, the weights can grow up to 7, but we need to keep track of weights only up to 4, requiring 3-bit weights and 3-bit adder/subtractors. The detailed design of a Boyer-Moore hardware voter will be discussed in Subsection IV.B (please look ahead to Fig. 7).

We won't discuss Boyer-Moore-based voters in connection with bit-voting any further, except to note in passing that in the case of weighted bit-voting [19] [20], the Boyer-Moore algorithm may become competitive, given the need for keeping track of the weights.

IV. Exact Word-Voters

A. Prior designs

The first systematic designs for exact word-voters were based on the use of sorting networks, followed by auxiliary circuitry [10]. Sorting the inputs brings equal values next to each other, making it easier to determine the number of repetitions. Given the $O(n \log^2 n)$ cost and $O(\log^2 n)$ latency of the best practical sorting networks, and significant additional circuitry needed in such voters, extra-large voters of this kind may not be practical. For small to moderate values of n, repositories of efficient sorting networks, derived by trial-and-error or by applying transformations to existing networks, are available [21], which can be used in voter design.

Over the years, many ad-hoc designs have been proposed for specific application contexts and with the assumption of various domains for input values. As we will see in the next subsection, here the Boyer-Moore algorithm offers significant advantages in cost and delay.

B. Boyer-Moore-based exact word-voters

Hardware voters designed based on the Boyer-Moore algorithm have $O(\log n)$ latency and $O(n)$ cost. Such voters are thus quite scalable to large numbers of input. In this subsection, we proceed with the assumption that a majority value exists among the n inputs. In Subsection V.B we will present a design framework that allows us to deal with cases when a majority value does not exist.

Consider the example in Fig. 1(a), which essentially shows the structure of a voting network. Each of the 6 nodes in the diagram, arranged in 3 levels, represents a hardware unit that compares the two weighted inputs (a, i) and (b, j) presented to it and generates an output c and a corresponding weight k based on the algorithm given in the pseudo-code of Fig. 6. The intuition behind the algorithm is that with equal input values, the weights add up, whereas with unequal values the difference of the two weights prevails.

Here is a brief description of the operation of the circuit in Fig. 7. If $a = b$, the "Sub" control signal to the adder/subtractor will be 0 and the weights i and j are added; otherwise, $i - j$ is computed, which yields a signed result. In the case of a negative value for $i - j$, its sign is flipped to find $j - i$. So, the output weight k is one of the three values $i + j$, $i - j$, or $j - i$, as prescribed by the algorithm in Fig. 6. The multiplexers on the top right of the diagram of Fig. 7 select one of the two input values a or b, or the null value Φ, as the circuit's output c.

```
Majority circuit hardware cell
Inputs (a, i) and (b, j)
Output (c, k)
if a = b
then (c, k) := (b, i + j)
else if i = j
     then (c, k) := (Φ, 0)
     else  if i > j
           then (c, k) := (a, i – j)
           else (c, k) := (b, j – i)
           endif
     endif
endif
```

Fig. 6. Pseudo-code for hardware cells from which voting networks based on the Boyer-Moore majority algorithm are built.

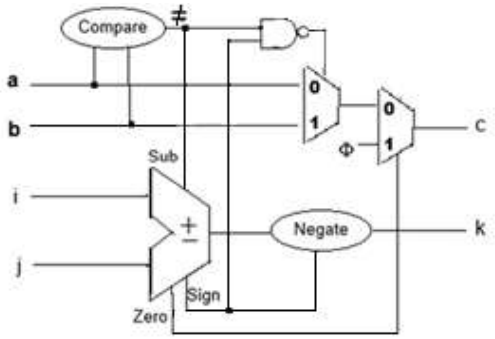

Fig. 7. Hardware cell realizing the algorithm described in Fig. 6

The weights i and j can theoretically go as high as n, requiring a $(\log_2 n)$-bit adder/subtractor. However, by adopting a saturation value of $(n + 1)/2$, we can reduce the width by using a saturating adder/subtractor.

V. Evaluation and Discussion

A. The principle of optimizing the common case

As we will show in the analysis that follows, in nearly all cases when we use voting networks, input values contain a majority value. We noted in Subsection II.B that when a majority does not exist, some voting algorithms may produce nonsensical results. However, given that such cases are rare, relegating their detection to hardware mechanisms that are off the critical path will allow the computation to proceed at full speed. The critical path contains hardware for the common case and works correctly almost all the time. If not, then the computation is rolled back and rerun. If we can show that the probability of rollback is extremely small, then performance is dictated by the performance in the common case.

The principle of optimizing the common case is now an important maxim in the field of computer architecture and design. It achieved prominence with the introduction of reduced instruction-set computer (RISC) architectures, whose design focused on running simple, frequently-used instructions extremely fast within a main instruction pipeline, while relegating complex instructions to programmed or microprogrammed implementation. Quantitative evaluations, both analytical and experimental, have shown that optimizing the common case achieves higher overall performance in a wide variety of applications.

B. A design framework

The Boyer-Moore algorithm always produces an output, even if majority agreement does not exist. But given that the probability of the latter event is extremely small (see the analysis in Subsection V.C), with channels that are reasonably reliable (error probability of 0.01 or smaller), we can proceed by assuming that the algorithm's output is indeed the majority, verifying this fact off the critical path and invalidating (rolling back) the computation in the unlikely event of no majority. Given that such a rollback is extremely unlikely, its impact on performance is negligible.

Figure 8 depicts the aforementioned 2-stage voting process. The verification box compares the n inputs with the candidate majority value and uses a bit-voter to confirm that a majority value does in fact exist. Let the tiny probability of not having a majority value be δ. The effective voting time, given below will be nearly identical to voting time in the common case.

(common-case voting time) + $\delta \times$ (rollback time)

Rollback can be accomplished in a number of different ways. One way is to provide buffers at the input to continued computation to save the inputs supplied to that block. If an invalidate signal is generated, the output of the continued computation is ignored and a recovery process is initiated. In cases when δ is expected to be super-small, we can avoid having a special recovery process by aborting the entire computation and restarting it from the beginning.

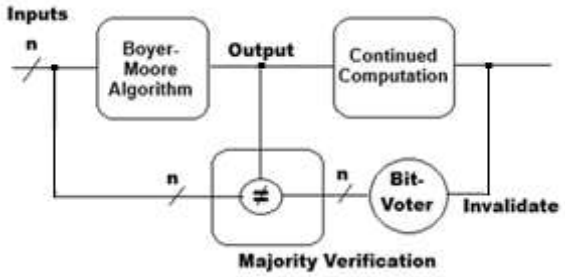

Fig. 8. The proposed 2-stage majority-voting scheme first finds a candidate result via the Boyer-Moore algorithm and then verifies the correctness of the majority result off the critical path.

In fact, the framework discussed above allows us to design not just majority voters, but also supermajority voters. For example, if we are interested in a 6-out-of-9 supermajority voter, we use the Boyer-Moore algorithm to find the majority value, but in the verification process, we replace the ordinary 5-out-of-9 bit-voter with a 6-out-of-9 supermajority bit voter. For the same reasons that it is quite unlikely not to have a majority among n fairly reliable inputs, not having the sought-after supermajority is also quite unlikely, making the described framework applicable.

C. The probability of not having a majority value

As we will show in the analysis that follows, in nearly all cases when we use voting networks, input values contain a majority. Let p be the probability of error in each of the n computation channels. To estimate the probability of not having a majority agreement we compute $1 - m$, where m is the probability of having a majority of correct results. The probability of not having a majority value is upper bounded by:

$$n = 3: \quad p^3 + 3p^2(1-p) = p^2(3-2p)$$

$$n = 5: \quad p^5 + 5p^4(1-p) + 10p^3(1-p)^2 = p^3(10 - 15p + 6p^2)$$

$$n = 7: \quad p^7 + 7p^6(1-p) + 21p^5(1-p)^2 + 35p^4(1-p)^3 = p^4(35 - 84p + 70p^2 - 20p^3)$$

Because the probability of not having a majority declines sharply for larger number of inputs (notice that the first term in each expression is p with an increasing power, so if p is small, the initial term declines quickly), we can draw appropriate conclusions from the cases considered above.

	$n = 3$	$n = 5$	$n = 7$
$p = 0.100$	0.028 000	0.008 560	0.003 097
$p = 0.050$	0.007 250	0.001 158	0.000 194
$p = 0.020$	0.001 184	0.00 0008	0.000 005
$p = 0.010$	0.000 030	0.000 001	0.000 000
$p = 0.005$	0.000 007	0.000 000	0.000 000
$p = 0.002$	0.000 001	0.000 000	0.000 000
$p = 0.001$	0.000 000	0.000 000	0.000 000

Fig. 9. Calculated upper bounds for the probability of not having a majority value among n inputs for different levels of input error p (reliability $1 - p$).

The values represented by the preceding equations and in Fig. 9 are upper bounds because they assume the worst case that any malfunctioning computation channel will necessarily disagree with the correct result. However, there do exist benign malfunctions that do not affect the result obtained, or perhaps affect it in such a way that the error can be detected before the voting stage, allowing us to isolate an incorrect value before it corrupts the voting process.

We see that even with fairly small values of n (7 or less) voting is extremely unlikely to produce an incorrect result or lead to a situation with a lack of majority.

VI. Conclusion and Future Work

In this paper, we reviewed the state of the art in designing voting networks of various kinds and considered whether the Boyer-Moore majority algorithm, a clever scheme originally intended for software voting, can be used as the basis for more-efficient hardware voters. The answer to the question above is a tentative "no" in the case of bit-voting networks and a definite "yes" for word-voting networks, both with exact (integer) and approximate (floating-point) input values.

The problem of finding a majority element in a data stream is a special case of finding frequent items [22], with the latter being significantly more complex in terms of the consumed time and/or the needed space. Given that plurality voting requires the determination of the most-frequently occurring value among the inputs, it is more complicated than threshold voting, a generalization of majority voting [23].

Hardware voters based on parallelizing the Boyer-Moore majority algorithm reduce circuit complexity from $O(n \log^2 n)$ to $O(n)$ and latency from $O(\log^2 n)$ to $O(\log n)$. These improvements are achieved at virtually no performance penalty, given the very small probability of computation rollback or restart in the event of no majority. At this stage of our work, we still do not have actual circuit implementations or simulation results. We are confident, however, that the mathematical analysis of Subsection V.C will prove consistent with observed performance.

The designs of our proposed voting networks intersect with and are impacted by certain arithmetic components [24]. For example, the adder/subtractor component in the cell design of Fig. 7 will likely be of saturating kind, necessitating further research in designing saturating add/subtract units optimized for this particular application domain. As another example, multiple options are available for the bit-voter block of Fig. 8, including new designs based on parallel counters that are based on sorting networks [25]. These and other design options constitute fruitful areas for further research.

Another aspect of this work we intend to pursue is considering the effect of replicated voters. In triplication, for example, we can use three voters to produce three independent copies of the final result, in a scheme known as triple-modular redundancy (TMR). If one voter's operation is affected by input errors or internal faults in such a way that its output is not a majority value, the other two voters may cover or mask the problem, possibly obviating the need for majority verification and further improving the overall performance.

References

[1] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr and D. K. Rubin, "The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Trans. Computers*, vol. C-20, no. 11, pp. 1312-1321, Nov. 1971.

[2] A. L. Hopkins, "A fault-tolerant information processing concept for space vehicles," *IEEE Trans. Computers*, vol. C-20, pp. 1394-1403, Nov. 1971.

[3] Wensley J. H., "SIFT: Software implemented fault tolerance, *Proc. Fall Joint Computer Conf.*, part I, pp. 243-253, 1972.

[4] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Research & Development*, vol. 6, no. 2, pp. 200-209, 1962.

[5] M. Stanisavljevic, A. Schmid, and Y. Leblebici, *Reliability of Nanoscale Circuits and Systems*, Springer, 2010.

[6] W. Zhang, et al., "Neuro-inspired computing chips," *Nature Electronics*, vol. 3, no. 7, pp. 371-382, 2020.

[7] A. Svoboda, "From mechanical linkages to electronic computers: recollections from Czechoslovakia," In: *A History of Computing in the Twentieth Century*, Academic Press, pp. 579-586, 1980.

[8] S. Sayil, "A survey of circuit-level soft error mitigation methodologies," *Analog Integrated Circuits and Signal Processing*, vol. 99, no. 1, pp. 63-70, 2019.

[9] B. Parhami, *Dependable Computing: A Multilevel Approach*, draft of unpublished book, available online (accessed Sep. 30, 2025) https://web.ece.ucsb.edu/~parhami/text_dep_comp.htm

[10] B. Parhami, "Voting networks," *IEEE Trans. Reliability*, vol. 40, no. 3, pp. 380-394, Aug. 1991.

[11] B. Parhami, "Recursive implementation of voting networks," *Proc. IEEE 14th Annual Computing and Communication Workshop & Conf.*, Jan. 2024.

[12] B. Parhami, "A 2D array of multiplexers can efficiently realize a variety of counting networks." *Proc. IEEE International Midwest Symposium on Circuits and Systems*, Aug. 2025.

[13] E. E. Swartzlander, "Parallel counters," *IEEE Trans. Computers*, vol. C-22, no. 11, pp. 1021-1024, 2006.

[14] N. Pippenger, "Selection networks," *SIAM J. Computing*, vol. 20, pp. 878-887, 1991.

[15] K. E. Batcher, "Sorting networks and their applications," *Proc. AFIPS Spring Joint Computer Conf.*, pp. 307-314, 1968.

[16] K. E. Batcher, "On bitonic sorting networks," *Proc. Intl. Conf. Parallel Processing*, pp. 376-379, 1990.

[17] G. Latif-Shabgahi, J. M. Bass, and S. Bennett, "A taxonomy for software voting algorithms used in safety-critical systems," *IEEE Trans. Reliability*, vol. 53, no. 3, pp. 319-328, 2004.

[18] B. Parhami, "A taxonomy of voting schemes for data fusion and dependable computation," *Reliability Engineering and System Safety*, vol. 52, pp. 139-151, 1996.

[19] B. Parhami, "Voting algorithms," *IEEE Trans. Reliability*, vol. 43, no. 4, pp. 617-629, Dec. 1994.

[20] B. Parhami, "The Parallel Complexity of Weighted Voting," Proc. *International Conf. Parallel and Distributed Computing and Systems*, pp. 382-385, Oct. 1991.

[21] S. Hunter, "Smallest and fastest sorting networks for a given number of inputs," GitHub repository, accessed on Sep. 30, 2025. https://bertdobbelaere.github.io/sorting_networks.html

[22] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in streams of data," *Communications of the ACM*, vol. 52, no. 10, pp. 97-105, 2009.

[23] B. Parhami, "Threshold Voting is Fundamentally Simpler than Plurality Voting," *International J. Reliability, Quality, and Safety Engineering*, vol. 1, no. 1, pp. 95-102, Mar. 1994.

[24] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2nd ed., 2010.

[25] Y. He and B. Parhami, "Implementing parallel counters based on sorting networks," this conference, Oct. 2025.