Counting in the Presence of Memory Faults

Gerth Stølting Brodal[1], Allan Grønlund Jørgensen[1], Gabriel Moruz[2,⋆], and Thomas Mølhave[3,⋆⋆]

[1] MADALGO⋆⋆⋆, Department of Computer Science, Aarhus University
[2] MADALGO⋆⋆⋆, Institut für Informatik, Goethe University Frankfurt am Main.
[3] Department of Computer Science, Duke University

Abstract. The faulty memory RAM presented by Finocchi and Italiano [1] is a variant of the RAM model where the content of any memory cell can get corrupted at any time, and corrupted cells cannot be distinguished from uncorrupted cells. An upper bound, δ, on the number of corruptions and $O(1)$ reliable memory cells are provided. In this paper we investigate the fundamental problem of counting in faulty memory. Keeping many reliable counters in the faulty memory is easily done by replicating the value of each counter $\Theta(\delta)$ times and paying $\Theta(\delta)$ time every time a counter is queried or incremented. In this paper we decrease the expensive increment cost to $o(\delta)$ and present upper and lower bound tradeoffs decreasing the increment time at the cost of the accuracy of the counters.

1 Introduction

Modern memory chips are made from increasingly smaller and complicated circuits that work at low voltage levels and offer large storage capacities [2]. Unfortunately, these improvements have increased the likelihood of *soft memory errors*, where arbitrary bits flip, corrupting the contents of the affected memory cells [3]. Soft memory errors are triggered by phenomena such as power failures, cosmic rays, and manufacturing defects. Even though the occurrence rate of these errors in individual memories is quite low they are a serious concern in applications running on clusters, where the frequency of soft memory errors is much larger. The soft memory errors rate is predicted to increase in the future [4]. Since the amount of cosmic rays increases with altitude, soft memory errors are a serious concern in fields like avionics and space research [5].

Corrupted memory cells can have significant consequences for algorithms. For instance, a single corruption in a sorted array can force a standard binary search to end up $\Omega(n)$ cells away from the correct position. Soft memory errors can also be exploited to break the security of software systems. This has been demonstrated in works breaking Java Virtual Machines [6], cryptographic protocols [7, 8], and smart-cards [9].

Soft memory errors can be addressed by using replication and error correcting codes at the hardware level, but this approach is not always popular since the increased circuitry requirements is costly with respect to performance, storage capacity, and money.

[⋆] Partially supported by the DFG grant ME 3250/1-1.

[⋆⋆] Work done while at MADALGO.

[⋆⋆⋆] Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

In software, memory errors have been addressed in a variety of settings, with the main focus on ensuring that code runs as expected, anticipating critical errors caused by hardware errors and malicious attacks. Errors are detected using techniques such as algorithm based fault tolerance [10], assertions [11], control flow checking [12], procedure duplication [13], and automatically correcting heap-based memory errors [14].

Most algorithms and data structures assume a perfectly reliable storage, but algorithms dealing with unreliable data were also proposed. These include fault-tolerant pointer-based data structures [15], fault-tolerant sorting networks [16], fault-tolerant parallel models [17], the liar model [18], and locally mendable distributed networks [19].

Faulty Memory RAM. Recently, the *faulty-memory RAM* model was proposed in [1]. This model is a regular RAM with word size w where any memory cell can get corrupted at any time during the execution of an algorithm, and a cell containing corrupted data cannot be distinguished from a cell that does not. Algorithms are provided with an upper bound, δ, on the number of corruptions that may occur during execution. We let $\alpha \leq \delta$ denote the actual number of corruptions that have taken place during the computation. Given that registers in the processor are considered incorruptible, $O(1)$ safe memory locations are provided. It is assumed that reading a word from memory is an atomic operation. An algorithm is *resilient* if it works correctly for all uncorrupted data. For instance, a resilient sorting algorithm outputs a sequence where all uncorrupted elements appear in sorted order and corrupted elements can appear anywhere in the output. The correctness of algorithms is usually proved by assuming that an adaptive adversary (worst-case) performs up to δ corruptions during the execution of an algorithm.

Several problems have been addressed in the faulty-memory RAM, see a recent survey [20] for more information. For instance optimal comparison based sorting algorithms and (static and dynamic) dictionaries [1, 21–23], and priority queues [24] have been proposed. In [25] it is shown that resilient sorting algorithms are of practical interest. Motivated by the increased soft memory errors frequency on clusters operating with massive data sets, in [26] resilient algorithms are linked to external-memory algorithms, providing the first external-memory algorithms resilient to memory faults.

Our results. We investigate maintaining many counters in the faulty memory RAM:

Definition 1. *A resilient counter with* additive error γ *is a data structure with an* increment *operation and a* query *operation. The query operation returns an integer between* $v - \gamma$ *and* $v + \gamma$ *where* v *is the number of increment operations preceding the query.*

We investigate upper and lower bound tradeoffs between the time needed for n increase operations and the additive error of the counter. We only consider data structures where no information is stored in safe memory between operations, therefore the counters are stored completely in unreliable memory. Our results are summarized in Figure 1.

In Section 2 we prove that any resilient counter with non-trivial additive error must use $\Omega(\delta)$ space, and that a deterministic query operation requires $\Omega(\delta)$ time. Furthermore, we prove a lower bound tradeoff between the increment time and the additive error, stating that if an increment operation takes $t \leq \delta$ time, the additive error is at least $\lfloor \delta/t \rfloor$ in the worst case, i.e. (increment time) × (additive error) $\geq \delta$. The lower

Time (n increments)	Query time	Additive error γ	Space	Section
$O(\delta n)$	$O(\delta)$	0	$O(\delta)$	-
$O(nt\log(\delta/t)+\alpha\log(\alpha/t))$	$O(\delta)$	α/t	$O(\delta)$	3.1
$O(n+\alpha\log\alpha)$	$O(\delta)$	$\alpha\log\delta$	$O(\delta)$	3.1
$O(n)$	$O(\delta^2)$	$O(\alpha^2)$	$O(\delta)$	3.2
$O(n+\alpha\sqrt{\delta})$	$O(\delta)$	α	$O(\delta)$	3.3
Expected $O(n)$	$O(\delta)$	α	$O(\delta)$	3.4

Fig. 1. Overview of our upper bounds.

bounds suggest that an optimal resilient counting data structure is characterized by an $O(\delta)$ space bound, $O(t)$ increment time, $O(\alpha/t)$ additive error and $O(\delta)$ query time.

In Section 3.1 and 3.3 we provide deterministic data structures where both the increment time and the additive error depend on α. The first result in Section 3.1 provides a tradeoff between the increment time and the additive error that does not blow up the space used by the data structure nor the query time. Given any $t \geq 1$ the data structure has additive error α/t and supports n increments in $O(nt\log(\delta/t)+\alpha\log(\alpha/t))$ time. A small change to this data structure gives a data structure with additive error $\alpha\log\delta$ that supports n increments in $O(n+\alpha\log\alpha)$ time. In Section 3.3 we describe a data structure with additive error α that supports n increments in $O(n+\alpha\sqrt{\delta})$ time. This is optimal for $n=\Omega(\alpha\sqrt{\delta})$.

In Section 3.2 we describe a deterministic data structure where the time used by an increment is independent of the number of possible corruptions. The data structure supports increments in $O(1)$ time in the worst case. The additive error of the data structure is $O(\alpha^2)$ and queries are supported in $O(\delta^2)$ time.

Finally, in Section 3.4 we present a randomized data structure with additive error α, that supports n increments in $O(n)$ time in expectation and supports queries in $O(\delta)$ time in the worst case. This is optimal up to constant factors.

The additive error of any of our resilient counters can be reduced by a factor of t by using t counters. Each increment operation increments all t counters and the query operation returns the sum of all t counters divided by t. However, this produces a new tradeoff by increasing the increment time and space by a factor of t. Similarly, any of our resilient counters can be used to create a new counter that supports both decrement and increment operations with the same additive error. This is achieved by using two counters; one to count the number of increment operations and one to count the number of decrement operations.

Preliminaries. Throughout the paper we denote by *reliable value* a value stored in unreliable memory that can be retrieved reliably despite possible corruptions. This is achieved by replicating the given value in $2\delta+1$ consecutive cells. Since at most δ of the copies can be corrupted, the majority of the $2\delta+1$ elements are uncorrupted. The value can be retrieved in $O(\delta)$ time with the majority algorithm in [27], which scans the $2\delta+1$ values keeping a single majority candidate and a counter in safe memory.

2 Lower Bounds and Tradeoffs

We present some simple lower bounds on space and time for resilient counters.

Space. Any resilient counter data structure with non-trivial additive error must use more than δ space. If the data structure uses δ space or less, the adversary can corrupt the entire structure and force a query operation to return any arbitrary value.
Deterministic Query. Any deterministic algorithm uses at least δ probes in the worst case for a query. If a query algorithm reads at most δ memory cells the adversary can simulate any value by corrupting δ cells. This means that the adversary can completely control the value returned by a query, making it impossible to get a non-trivial bound on the additive error.
Deterministic Increment. If an increment takes k time the adversary can roll back the changes to the data structure done by the last $\lfloor \delta/k \rfloor$ increments, or do the changes to the data structure corresponding to $\lfloor \delta/k \rfloor$ increments. Thus, the counter has additive error at least $\lfloor \delta/k \rfloor$ in the worst case.

3 Data Structures

3.1 Replicating Bits

In this section we describe a data structure that is parameterized with an integer t, $1 \leq t \leq \delta$. The data structure uses $O(\delta)$ space and has additive error $\lfloor \alpha/t \rfloor$. The time used for n increments is $O(nt\log(\delta/t) + \alpha\log(\alpha/t))$, and queries take $O(\delta)$ time.
Structure. The data structure maintains the bits of the binary representation of the counter value separately, each bit replicated depending on its significance as follows. For $i = 0, \ldots, \lfloor \log(\delta/t) \rfloor$ the i'th least significant bit is replicated $t2^{i+1}$ times in $t2^{i+1}$ different memory cells. The value of the remaining $w - \lfloor \log(\delta/t) \rfloor$ most significant bits are stored in a reliable variable v. The memory cells are stored in one array of size $O(\delta)$.
Increment. Increments are implemented as binary addition, where we consider the i'th bit to be one if at least $t2^i$ of the $t2^{i+1}$ copies of it are non-zero. The i'th bit is set by writing the value of the bit in all of the $t2^{i+1}$ copies.
Query. The query algorithm reliably retrieves the value of the $w - \lfloor \log(\delta/t) \rfloor$ bits stored in v. For the lower order bits, we add 2^i to the sum, for $i = 0, \ldots, \lfloor \log(\delta/t) \rfloor$, if at least $t2^i$ of the $t2^{i+1}$ copies of the i'th least significant bit are non-zero.
Additive Error. Since the value of the i'th bit is given by the majority value of $t2^{i+1}$ copies, the adversary must use $t2^i$ corruptions to alter the i'th bit. Changing the i'th bit changes the value stored in the data structure by 2^i, yielding an additive error of $\lfloor \alpha/t \rfloor$.
Complexity. If no corruptions occur, we update the i'bit of the counter every 2^i increments, taking $O(t2^i)$ time. Similarly, we update v after $\Theta(\delta/t)$ increments in $O(\delta)$ time. Therefore, if we ignore corruptions, the time used for n increments is $O(nt\log(\delta/t))$.

The only way corruptions can influence the running time of increment operations is by changing the value of a bit. Assume the adversary corrupts the i'th bit, using $t2^i$ corruptions. After a number of increments a cascading carry affects this (corrupted) bit and the increment operation writes the $t2^{i+1}$ copies of the $i+1$'th bit. We charge the work needed to move the $t2^i$ corrupted bits to the corruptions that caused them. These corrupted bits can be charged in $\log(\delta/t) - i$ such cascading carries. However, when k increments have been performed, where $kt > \alpha$, the time used by the increments alone is $O(kt\log\delta/t)$ dwarfing the time needed to deal with corruptions. Otherwise, the number stored in the data structure is at most $k + \alpha/t \leq 2\alpha/t$. Thus, the most significant bit written in an increment operation is the $\lceil \log(\alpha/t) \rceil$ least significant bit. We conclude that the extra time needed to deal with corruptions is $O(\alpha \log(\alpha/t))$.

Theorem 1. *The counter structure uses $O(\delta)$ space and has additive error $\lfloor \alpha/t \rfloor$. The time used for n increments is $O(nt\log(\delta/t) + \alpha\log(\alpha/t))$ and queries take $O(\delta)$ time.*

Trading off Additive Error for Increment Time. We can reduce the time for n increments to $O(n + \alpha\log\alpha)$ by storing the $\lfloor \log\log\delta \rfloor$ least significant bits in the same memory cell. For $i = \lfloor \log\log\delta \rfloor + 1, \ldots, \log\delta$ the i'th least significant bit is replicated in $2^{i+1}/\lfloor \log\delta \rfloor$ memory cells. The remaining bits are stored in a reliable value v as before. One corruption can change the $\lfloor \log\log\delta \rfloor$ least significant bits causing an additive error of at most $\lfloor \log\delta \rfloor$, and $2^i/\lfloor \log\delta \rfloor$ corruptions are needed to corrupt the i'th bit. The increment and the query are basically the same.

Corollary 1. *The counter structure uses $O(\delta)$ space and has additive error $\alpha\log\delta$. The time used for n increments is $O(n + \alpha\log\alpha)$ and queries use $O(\delta)$ time.*

3.2 Round-Robin Counting

In this section we describe a data structure that uses $O(\delta)$ space and has $O(\alpha^2)$ additive error. Increments are supported in constant time, and queries use $O(\delta^2)$ time.
Structure. The data structure consists of an array A of $k = 2\delta + 3$ integers $C_1, \ldots, C_k$ used as counters, and a round-robin index i. The structure is initialized by setting all counters to zero and i to one. We denote by *corrupted counter* a counter that has been changed directly by the adversary.
Increment. If i is not in the range $1, \ldots, k$, it has been corrupted and we reset it to one. Next, we increment first C_i and then i. If i becomes $k + 1$ we set it to one. Note that i could have been corrupted to a value in $1, \ldots, k$, but we do not check if this happened.

Let v_j be the number of times the increment algorithm has incremented C_j, and let $v = \sum_{j=1}^{k} v_j$ denote the correct value of the counter. If no corruption has taken place, then $C_1 = \cdots = C_r = d + 1$ and $C_{r+1} = \cdots = C_k = d$, where $d = \lfloor v/k \rfloor$ and $r = v \bmod k$. Furthermore, if no counter has been corrupted, $v = \sum_{j=1}^{k} C_j$, regardless of corruptions of the round robin index i.
Query. Let α_i be the number of times i has been corrupted. The key observation for the query algorithm is that for any two uncorrupted counters, C_a and C_b, we have $|v_a - v_b| \leq \alpha_i + 1$, which means that $|v/k - v_a| \leq \alpha_i + 1$.

First, we compute a value m larger than or equal to at least one uncorrupted counter, and smaller than or equal to at least one uncorrupted counter. Since the difference between two uncorrupted counters is at most $\alpha_i + 1$, $m \in \{\frac{v}{k} - \alpha_i - 1, \frac{v}{k} + \alpha_i + 1\}$. After computing m, simply returning mk yields an additive error of $O((\alpha + 1)k) = O((\alpha + 1)\delta)$. To improve the additive error we locate $O(\alpha)$ counters which are too far from m and ignore them.

We store m in safe memory and compute it as in [21] as follows. Initially, we set m to $-\infty$. The k counters are scanned $\lceil k/2 \rceil$ times. In each iteration we update m to the minimum counter larger than the current m. Since $k = 2\delta + 3$, after $\lceil k/2 \rceil$ iterations there exist two uncorrupted counters, such that one is smaller and one is larger than m.

Next, we find a bound, x, on the number of the counters that are too far away from m as follows. Initially, we set x to one. Then, the number of counters c outside the range $\{m-x,\dots,m+x\}$ is counted in a scan. If $c \geq x$ we increment x and recompute c. This process ends when x becomes larger than c. Finally, we scan the k counters maintaining a sum, initially zero, in safe memory. If a counter stores a value in the range $\{m-x, m+x\}$ we add it to the sum. If a counter is outside the range, it is far from m, and we add m to the sum. Finally, we return the computed sum.

Additive Error. Let α_c be the number of times a counter was corrupted by the adversary. By definition, $\alpha_i + \alpha_c = \alpha \leq \delta$. First we recall that for any two uncorrupted counters, C_a and C_b, we have $|v_b - v_a| \leq 1 + \alpha_i$, and that the value of m is in the range $\{\frac{v}{k} - \alpha_i - 1, \frac{v}{k} + \alpha_i + 1\}$. Therefore, if $x \geq \alpha_i + 1$ in the above algorithm, then c, the number of counters that are not in the range $\{m-x, m+x\}$, is at most α_c, the number of counter corruptions. At most α_c corrupted counters can be counted by c, and we conclude that when the algorithm terminates, then $x \leq \alpha_i + \alpha_c + 1$.

Let S be the set of counters not counted by c, i.e. all counters in the range $\{m-x, m+x\}$. All uncorrupted counters in S are unchanged and do not contribute to the error. Let C_j be a corrupted counter in S. By definition of m and x we know that $|v_j - C_j| \leq |v_j - m| + |m - C_j| \leq \alpha_i + 1 + x \leq 2\alpha + 1$. Therefore, each corrupted counter in S can affect the additive error by $O(\alpha)$. We add m to the result for all counters outside the range $\{m-x, m+x\}$. By definition of m, the value for uncorrupted counters not in S differs from m by at most $\alpha_i + 1$. Similarly, for any corrupted counter C_j not in S the difference between m and v_j is at most $\alpha_i + 1$. There are at most $x = O(\alpha)$ counters not in S, and at most α_c corrupted counters in S, leading to an additive error of $O(\alpha^2)$.

Complexity. The increment operation uses $O(1)$ time to update a counter and the round robin index. The query time is given by the time used to compute m and x, that is $O(\delta^2)$.

Theorem 2. *The counter data structure described uses $O(\delta)$ space and has an additive error of $O(\alpha^2)$. Increments are supported in $O(1)$ time and queries in $O(\delta^2)$ time.*

3.3 Counting by Scanning Bits

We describe a counter data structure that uses $O(\delta)$ space with additive error α. It performs n increments in $O(n + \alpha\sqrt{\delta})$ time, and answers queries in $O(\delta)$ time. First, we describe a simpler data structure with an additive error of α that supports n increments in $O(n + \alpha\delta)$ time. Subsequently, we reduce the cost for n increments to $O(n + \alpha\sqrt{\delta})$.

Structure. The data structure stores an array A of δ memory cells, a reliable variable v, and a round-robin index i. Each cell of A is used to store a single bit. We initialize all values in A to zero, v to zero, and i to one.

Increment. If $A[i] = 0$ we set $A[i] = 1$ and set $i = 1 + (i + 1 \mod \delta)$. Otherwise, we count the number of non-zero entries in A. We add this number plus one (for the current increment) to v and set all entries in A to zero.

Query. We count the number v' of non-zero entries in A, retrieve v, and return $v + v'$.

Additive Error. Every time we add a value, k, to the reliable value v in an increment we have seen $k - 1$ non-zero entries in A. The only way a cell in A can be non-zero is if it was set to one by an earlier increment operation, or the adversary corrupted it. Conversely, a cell is set to zero either after updating the reliable value or by a corruption.

Thus, the number returned by a query differs by at most α from the actual number of increments performed.

Complexity. If no corruptions occur, the increment operation takes $O(1)$ amortized time, since setting a value in A to one takes $O(1)$ time and updating v takes $O(\delta)$ time and occurs every $\delta+1$ increments. Every corruption to the round robin index i or an element of A can force us to scan A and reliably add a value to v, and this takes $O(|A|+\delta) = O(\delta)$ time. Therefore, n increments take $O(n+\alpha\delta)$ time.

Improving Increment Time by Packing. We improve the time used for n increments to $O(n+\alpha\sqrt{\delta})$ by packing elements in A to an auxiliary array. In addition to the reliable value v and the array A of size δ, we store an array P of size δ, which is logically divided into $\Theta(\sqrt{\delta})$ blocks of $\sqrt{\delta}$ consecutive memory cells.

Increment. First, we test if i is in the range $\{1,\ldots,\delta\}$. If not then i has been corrupted and we set it to one. Then, we test whether $A[i]=0$ and if so, we set $A[i]=1$ and increment i. If i becomes $\delta+1$ we set i to one. However, unlike the simpler data structure, if $A[i]\neq 0$, a *packing phase* is initiated. In the packing phase we scan A from left to right starting from $A[1]$ until we encounter a zero, or the end of A is reached. During the scan we count the amount, c, of non-zero entries read and set all these entries to zero. After the scan i is set to one. Then, we set c entries in P to one as follows. Let d_j be the index in P of the first element in the j'th logical block. We scan P from d_1. If we see an entry storing a zero, we set it to one, and decrement c. If we see something else we go to the start of the following logical block and continue. We stop the packing phase when c reaches zero or a non-zero element, or the boundary of the last block is found. If $c>0$ after the packing phase, we count the amount of non-zero elements in A and P in a scan and set all entries to zero. This count summed with c is added to v.

Query. The query operation returns the sum of v and the number of ones in A and P.

Additive Error. Similarly to the simpler data structure, each corruption can only change the value of the data structure by one. It follows that the additive error is α.

Complexity. We analyze the time used between two consecutive updates of v and this time-frame we denote a round. The array A consists of a number of sections of non-zero elements separated by zeros. Note that the packing phase removes at least one section. If no corruptions occur, increments can only extend sections. A corruption, of a cell in A or of the index i, may extend a section, connect two sections, create a section or split an existing section in two. The same things can happen in an increment following a corruption of the index i. Thus, the number of sections created during a round is bounded by one plus the number of corruptions, and a section is moved only once in P.

Moving t non-zero entries from A to P in a packing phase takes $O(t+\sqrt{\delta})$ time, and the clean ending the round takes $O(\delta)$ time. Let c_p be the number of increments and α_p be the number of corruptions in the p'th round. Since the packing phase is called at most α_p+1 times, the time used in the p'th round is $O(c_p+\alpha_p\sqrt{\delta}+\delta)$. We show that the $O(\delta)$ time used for the clean can be payed for by the c_p increments and the α_p corruptions, by charging $O(1)$ per increment and $O(\sqrt{\delta})$ per corruption.

If we copy elements to the i'th logical block in P in a packing phase and encounter a non-zero entry before filling all the $\sqrt{\delta}$ cells, at least one cell in the block is corrupted. Furthermore, we never put elements in the i'th block again unless a new corruption occur, setting a zero in the first entry of the block. This means that the only block that is changed by a packing phase that is not completely filled or has a cell that has been corrupted since the last time it was updated, is the last block considered in the phase.

When an increment performs a clean, ending the round, the first block of all logical blocks contained a non-zero entry during the packing phase. We categorize the $\sqrt{\delta}$ logical blocks as *filled* blocks, *corrupted* blocks, and *last* blocks. A filled block is a logical block which a packing phase has filled with $\sqrt{\delta}$ non-zero entries, a corrupted block contains a cell that has been corrupted during the round and which is not filled, and a last block is a block that does not contain a corrupted cell, but was not completely filled during the packing phase that put a one in the first entry of the block.

There are at most $\alpha_p + 1$ packing phases in a round, thus at most $\alpha_p + 1$ last blocks, and at most α_p corrupted blocks. If there are f filled blocks then we have performed at least $f\sqrt{\delta} - \alpha_p$ increments in the round. This means that there are $\sqrt{\delta} - f$ other blocks (corrupted, last) and since there are $O(\alpha_p)$ blocks that are not filled, $\sqrt{\delta} - f = O(\alpha_p)$. We have charged each increment $\Theta(1)$, which means that the increments have payed at least $f\sqrt{\delta} - \alpha_p$. It remains to charge $\delta - (f\sqrt{\delta} - \alpha_p) = \sqrt{\delta}(\sqrt{\delta} - f) + \alpha_p$ to the α_p corruptions. Since $\sqrt{\delta} - f = O(\alpha_p)$, we have charged enough if each corruption pays $\Theta(\sqrt{\delta})$. We conclude that n increments take $O(n + \alpha\sqrt{\delta})$ time.

Theorem 3. *The counter data structure uses $O(\delta)$ space and has additive error α. The time used for n increments is $O(n + \alpha\sqrt{\delta})$ and queries are answered in $O(\delta)$ time.*

3.4 Using Randomization to Obtain Fast Increments

In this section we describe a randomized data structure that uses $O(\delta)$ space and has additive error α. The expected time used for n increments is $O(n)$, and queries are supported in $O(\delta)$ time in the worst case. The data structure is similar to the data structures in Section 3.3 but randomization is used to find an empty cell fast.

Structure. The data structure stores an array A of size $k = 3\delta$ and a resilient variable v. Initially, v and all entries in A are set to zero.

Increment. We pick a random index $r \in \{1, \dots, k\}$ and *probe* $A[r]$. If $A[r] = 0$, we set $A[r] = 1$ and return. Otherwise, the probe *failed* and we do one of two things: with probability $\frac{k-1}{k}$ we restart the increment operation and with probability $\frac{1}{k}$ we *clean* the array. The clean operation counts the number of non-zero entries in A and adds this plus one (the current increment) to the reliable value v, then it sets all entries in A to zero.

Query. The query operation is the same as the one in Section 3.3, it simply counts the number of non-zero entries in A and returns the sum of this number and v.

Additive Error. As in Section 3.3 the additive error is α since each unreliable array entry contributes at most one to the result.

Complexity. The query operation simply scans A and retrieves v in $O(\delta)$ time. The expected time analysis of the increment operation is more involved. The sequence of n increments is logically divided into $\lceil n/t \rceil$ rounds of $t = \lceil \delta/2 \rceil$ increments. We prove that the expected cost of each round is $O(t)$, and then the bounds follow from linearity of expectation. We split each full round in two parts, the first part consists of the increments performed before the first clean in the round, and the remaining increments are the second part. If a round does not do a clean, we additionally charge for repeatedly doing failed probes until a clean would be performed. When the first part starts, the state of the array A could be anything. When the second part starts, the array stores only zero values. We divide the cost of the t increments into three.

The cost of successful probes, the cost of failed probes and the cost of doing cleans. The cost of the successful probes is $O(t)$. The cost of failed probes, is divided into two, a cost for the failed probes in the first part and a cost for the failed probes in the second part. The first part ends when the first clean is performed. We charge the first failed probe in each increment to the increment itself. The remaining number of failed probes is upper bounded by the number of times we restart the increment operation before we clean, and a clean is performed with probability $\frac{k-1}{k}$. Thus, the probability of doing exactly f additional failed probes is $(\frac{k-1}{k})^f \frac{1}{k}$. This means that the expected cost of failed probes in the first part is bounded by $t + \sum_{f=0}^{\infty} f(\frac{k-1}{k})^f(\frac{1}{k}) = O(t)$. In the second part we place at most t ones in A and the adversary can at most introduce δ non-zero entries. Therefore, during each increment in the second part, half of the entries in A contains a zero. This means that for each increment in the second part we expect to do one failed probe implying that the expected cost of failed probes in the second part is linear in the number of increments. Each round makes one clean in the first part, and for each increment in the second part, the probability of doing a clean is at most $\frac{1}{2}\sum_{f=1}^{\infty} \frac{1}{2^f}(\frac{k-1}{k})^{f-1}\frac{1}{k} \leq 2/k$. Thus, the expected cost for doing cleans in the second part is $O(1)$ per increment, we conclude that the expected cost of a full round is $O(t)$.

Only the last round remains. If this is the first round, it has no first part, and by the analysis above the cost of this round is linear in the number of increments. If the last round is not the first round, the expected cost is $O(\delta)$ even if zero increments has been performed. We charge this cost to the second to last round.

Theorem 4. *The counter data structure described uses $O(\delta)$ space and has additive error α. The expected time used for n increments is $O(n)$, and queries use $O(\delta)$ time.*

4 Open Problems

The main open problem is whether there exists a data structure that given any $t \geq 1$ has additive error $O(\alpha/t)$, supports increments in $O(t)$ time and queries in $O(\delta)$ time. One resilient counter needs $\Omega(\delta)$ space. It would be interesting too see if one can store k counters using $o(k\delta)$ space with each counter having a non-trivial bound on the additive error. Most of the counters presented in this paper require $\Theta(\delta)$ space for a reliable variable which seems hard to share among several counters. It may be interesting to see if one can use the safe memory to store some state to achieve this and possibly circumventing the lower bound tradeoff between increments and additive error.

References

1. Finocchi, I., Italiano, G.F.: Sorting and searching in the presence of memory faults (without redundancy). In: Proc. 36th Annual ACM Symposium on Theory of Computing. (2004) 101–110
2. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE micro **23**(4) (2003) 14–19

3. Tezzaron Semiconductor: Soft errors in electronic memory - a white paper. http://www.tezzaron.com/about/papers/papers.html (2004)
4. Baumann, R.: Soft errors in advanced computer systems. IEEE Design and Test of Computers **22**(3) (2005) 258–266
5. Taber, A., Normand, E.: Single event upset in avionics. IEEE Transactions on Nuclear Science **40**(2) (1993) 120–126
6. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: IEEE Symposium on Security and Privacy. (2003) 154–165
7. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's guide to fault attacks. Proceedings of the IEEE **94**(2) (2006) 370–382
8. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Eurocrypt. (1997) 37–51
9. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems. (2002) 2–12
10. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers **33** (1984) 518–528
11. Rela, M.Z., Madeira, H., Silva, J.G.: Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In: Proc. 26th Annual International Symposium on Fault-Tolerant Computing. (1996) 394–403
12. Abadi, M., Budiu, M., Úlfar Erlingsson, Ligatti, J.: Control-flow integrity. In: Proc. 12th ACM conference on Computer and communications security. (2005) 340–353
13. Pradhan, D.K.: Fault-tolerant computer system design. Prentice-Hall, Inc. (1996)
14. Novark, G., Berger, E.D., Zorn, B.G.: Exterminator: Automatically correcting memory errors with high probability. Communications of the ACM **51**(12) (2008) 87–95
15. Aumann, Y., Bender, M.A.: Fault tolerant data structures. In: Proc. 37th Annual Symposium on Foundations of Computer Science. (1996) 580–589
16. Leighton, T., Ma, Y.: Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. SIAM Journal on Computing **29**(1) (2000) 258–273
17. Chlebus, B.S., Gasieniec, L., Pelc, A.: Deterministic computations on a pram with static processor and memory faults. Fundamenta Informaticae **55**(3-4) (2003) 285–306
18. Ravikumar, B.: A fault-tolerant merge sorting algorithm. In: Proc. 8th Annual International Conference on Computing and Combinatorics. (2002) 440–447
19. Kutten, S., Peleg, D.: Tight fault locality. SIAM Journal on Computing **30**(1) (2000) 247–268
20. Finocchi, I., Grandoni, F., Italiano, G.F.: Designing reliable algorithms in unreliable memories. Computer Science Review **1**(2) (2007) 77–87
21. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G.F., Jørgensen, A.G., Moruz, G., Mølhave, T.: Optimal resilient dynamic dictionaries. In: Proc. 15th Annual European Symposium on Algorithms. (2007) 347–358
22. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: Proc. 18th ACM-SIAM Symposium on Discrete Algorithms. (2007) 547–553
23. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal resilient sorting and searching in the presence of dynamic memory faults. Theoretical Computer Science (2009) To appear.
24. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority queues resilient to memory faults. In: Proc. 10th International Workshop on Algorithms and Data Structures. (2007) 127–138
25. Petrillo, U.F., Finocchi, I., Italiano, G.F.: The price of resiliency: a case study on sorting with memory faults. In: Proc. 14th Annual European Symposium on Algorithms. (2006) 768–779
26. Brodal, G.S., Jørgensen, A.G., Mølhave, T.: Fault tolerant external memory algorithms. In: Proc. 11th Algorithms and Data Structures Symposium. (2009) 411–422
27. Boyer, R.S., Moore, J.S.: MJRTY: A fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. (1991) 105–118