Technical Report: TR-96-07, March 28, 1996

Combinatorial algorithms in the presence of random errors - II

Pedrotti,Alberto

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

In the present study, we consider a computational model similar to the classical RAM machine, except that each instruction performed by the machine may fail with probability $q < 1/6e$. The failures are transient and independent of each other. The definition of this model (which we call ERRAM) follows a previous work citelp, where we did not allow for addressing errors. We show how an arbitrary RAM program may be transformed into an ERRAM program, working with assigned probability $1 - delta$, provided that we know the number T of instructions that are executed by the original program, or an upper bound to it. One major tool that we use is redundancy, namely, each operation is repeated a certain number of times, and each stored value is kept in several copies. We derive upper bounds on the amount of redundancy that is needed to achieve the desired confidence. This allows us to bound both the space complexity and the expected time complexity of the resulting ERRAM program, in terms of T, $delta$ and q.

ACM–CR Subject Classification:

Keyword and phrases: Error probability; Confidence bound; Reliable algorithm; RAM program.

Combinatorial algorithms in the presence of random errors - II *

A. Pedrotti

Scuola Normale Superiore, Pisa, Italy

e.mail: pedrotti@sns.it

Abstract. In the present study, we consider a computational model similar to the classical RAM machine, except that each instruction performed by the machine may fail with probability $q < 1/6e$. The failures are transient and independent of each other. The definition of this model (which we call ERRAM) follows a previous work [9], where we did not allow for addressing errors.

We show how an arbitrary RAM program may be transformed into an ERRAM program, working with assigned probability $1 - \delta$, provided that we know the number T of instructions that are executed by the original program, or an upper bound to it. One major tool that we use is redundancy, namely, each operation is repeated a certain number of times, and each stored value is kept in several copies. We derive upper bounds on the amount of redundancy that is needed to achieve the desired confidence. This allows us to bound both the space complexity and the expected time complexity of the resulting ERRAM program, in terms of T, δ and q.

1 Formulation of the problem

Errors may arise in computational processes due to a wide variety of causes. The interaction between errors and computation has been object of studies, since the very beginning of computer science. For example, a Boolean circuit, designed to compute a certain function, may fail to correctly compute it due to transient, or permanent defects of the gates or of the links. Similarly, a computation carried out on an ordinary computer may fail to provide correct results, due either to software defects, or to transient hardware failures, or to permanent hardware failures.

*This work has been supported in part by MURST of Italy.

In particular, the effect of transient hardware failures has been widely investigated in the case of Boolean circuits. Many studies show how we can design circuits that compute a given function, in spite of the random failure of certain parts of the hardware [?]. One approach is to start with a circuit that, in absence of failures, computes the given function. Then, applying standard transformation rules, a new circuit is found, that is able to correctly compute the function with an assigned probability, in spite of random failures. The major tool that allows this achievement is redundancy, that is, we have to replicate certain portions of the circuit, and to interconnect them in a suitable way. Due to this, the resulting circuit is often deeper and wider than the original one.

In this paper, we adopt a similar approach for the well-known RAM model, as described for example in [1]. We start from an algorithm coded as a RAM program. Working appropriately on it, we get a program that, with high probability, works correctly even if the operations performed by the machine are subject to transient random failures.

We stress that our approach does not work in the case of permanent hardware defects. Moreover, note that we are by no means concerned with the design of algorithms that work is spite of a *known* (or, at least, bounded) amount of erroneous information. Such algorithms, known as *fault-tolerant* algorithms, are able to defeat errors, even if the latter are maliciously created by an adversary, provided that the error pattern satisfies some given constraint. Namely, if the posed conditions are satisfied, these algorithms devise correct results. This is because the restriction on the error pattern allows them to perform a certain amount of reliable computation. A similar situation arises in the studies of Blum [2]-[4], since the computation performed by an unreliable agent is interleaved with checking phases performed on an infallible medium. In our case, this is no longer true, since we cannot fully rely on even one operation. As a consequence, the results devised by our programs are correct with a probability that is strictly less than 1. In fact, we shall prove that, given any positive constant $\delta \leq 1/2$, every RAM program can be easily transformed into a program that works with probability $1 - \delta$, in spite of the random errors.

In a previous work (see [9]) we considered a framework similar to that of the present study, except that the effect of errors was restricted in a somewhat artificial manner. Namely, we supposed that a write into memory could store a wrong value, but not reach a wrong destination. The present setting is far more realistic. For brevity, we will call ERRAM the computational model which we are going to describe.

The work is organized as follows. In section 2, we first list the resources and the instructions of the ERRAM machine. Next, we state in detail where the failures can occur, isolating little pieces of hardware that we assume to be fully reliable. Namely, few operations on a handful of registers will be supposed to be free of error. In section 3, we give the rule to transform an arbitrary RAM program into an ERRAM program, that works within the stated confidence bounds. One main tool is the k-fold replication of each memory register, with the exception of some special registers. Moreover, each operation of the original RAM program has to be repeated in the ERRAM program a certain number of times. This number is strictly related to the same parameter k. In section 3 we carry out the analysis that leads to the determination of k; in section 4 we complete the task of bounding the expected operation complexity of the ERRAM program.

We stress that the value of k that we will find is only an upper bound to the amount of redundancy that makes our strategy work. We do not look for a corresponding lower bound. Similarly, we do not rank the performance of the proposed transformation strategy among those of other (perhaps much more complicated) strategies.

2 The ERRAM model

Let us recall that the RAM machine of [1] includes two special registers, namely, the instruction counter IC and the accumulator A. For convenience, we assume that the RAM machine is equipped with a second accumulator B, and the set of instructions of [1] is modified as follows:

- instructions are added to load and store B;
- the operands to binary instructions are always drawn from the accumulators A and B.

Moreover, to make the analysis simpler, we do not allow for indirect addressing.

Let us now list the resources of the ERRAM machine. We suppose that it has an instruction counter IC, three accumulators A, B and C, and a file of $\overline{M}$ ordinary registers, which we call *memory cells.* The set of instructions is the same as that of the RAM machine discussed above, with the addition of instructions to load and store C.

Let us consider to be given an arbitrary program P, written for the two-accumulator RAM machine described above, and some instance π for this program. Let M be the amount of RAM memory required by $P(\pi)$; similarly, let T

be the number of instructions executed by $P(\pi)$. To run $P(\pi)$ on the ERRAM machine, we logically partition the $\overline{M}$ ERRAM cells into M *locations*; each location includes K cells (we suppose that $\overline{M} = KM$). Each location is intended to keep a single information, which is held in multiple copies to enhance tolerance against addressing errors. On the other hand, note that the special registers are not replicated, since we agree that they cannot be hit by addressing errors. Let $k \leq K$ be the number of copies that are held in each location; k is to be determined later. To make the analysis more clear, we shall suppose that $k = 6r + 1$; however, our analysis works for an arbitrary odd integer as well.

Recall that, when discussing RAM programs, one commonly supposes that the problem fits into the memory of the machine. Similarly, we want to deal only with problems that fit into the ERRAM memory. If the original RAM program needs a memory of size M, its ERRAM version will need a memory of size kM. As we shall see, k depends on the following parameters:

- the number T of RAM operations executed by ALG;
- the confidence parameter δ;
- the probability q that an operation performed by the ERRAM fails.

Thus, the memory requirements of an ERRAM program do not depend only on those of the original RAM program. In the following, we suppose that the condition $\overline{M} \geq kM$ is fulfilled.

We shall divide the ERRAM instructions into two classes: ST instructions (store-type) and LT instructions (load-type), to achieve register-to-memory and memory-to-registers actions, respectively. As in [1], direct and immediate addressing are allowed, with the usual meaning. Each instruction is implemented with a micro-program composed of *operations.* At the operation level, the register-to-memory and memory-to-register actions will be denoted as *write* and *read* operations, respectively. The failures take place at the operation level.

2.1 Register-to-memory instructions

ST instructions include the store and arithmetic instructions; they always act on memory locations. The store instructions STA X, STB X, STC X, aim to transfer the value kept in the accumulator they refer to, into the memory location X. Let us describe, for example, the STA X instruction. We denote with $X[j]$ the j-th cell of location X. Since we want the information to be present in the memory in k copies, we execute the following program:

for $j := 1$ **to** $6r+1$ **do**
    $X[j] := A$;

An error can occur while performing the assignment $X[j] := A$; we then say that the assignment fails. More precisely, the assignment may perform in three ways:

- with probability $1-q$, the assignment performs correctly, that is, the right value is transferred into the right destination;

- with probability $q/2$, there is an error in the data movement, causing a wrong value to be written into $X[j]$;

- with probability $q/2$, there is an addressing error, that is, the value destinated to $X[j]$ is transferred into a randomly chosen memory cell. All the $\overline{M}$ cells may be hit by the erroneous write with the same probability.

Note that this formulation rules out the possibility that a *wrong* value is written into a *wrong* destination. This assumption is somewhat artificial; namely, it would be reasonable to suppose that errors in data movement and addressing errors arise independently. If this were the case, they might simultaneously affect the same assignment. Let us explain why the formulation we have given is more convenient. First, let us note that, from the point of view of the intended destination $X[j]$, it does not matter whether the error affects the datum, or the address, or both. In each case, $X[j]$ is very likely to contain the wrong value after the failure. Namely, even if the error affects the address, but not the datum, the old content fails to be overwritten by the new one, which we expect in general to be different.

Let us take the point of view of a memory cell, that is hit by an addressing error. If such cell belongs to a location other than X, we expect that the value kept in it is lost, no matter if it is overwritten with the true content of A, or with something other. However, if the addressing error hits $X[j]$, $j \neq i$, $1 \leq j \leq k$, and we also allow an error in the data movement, it is easy to see that the location X may be damaged twice by the failure of a single assignment. On the other hand, it is clear that such event would be exceptionally rare; it is convenient to rule it out at all from our analysis. This is achieved by the proposed formulation.

The arithmetic instructions ADD X, SUB X, MULT X, DIV X perform the intended binary operations on the contents of the accumulators A and B, storing the results into X. For example, the instruction ADD X is described by the following program:

for $i := 1$ **to** $6r + 1$ **do**
    $X[i] := A + B$;

It is reasonable to think that the arithmetic computation is subject to failure. However, note that such failure cannot be distinguished from an error affecting the transfer of the result into the assigned destination. It is then easy to get convinced that the error model discussed for store instructions is suited to arithmetic instructions as well.

2.2 Memory-to-registers instructions

LT instructions transfer information from a location to either an accumulator, or the instruction counter. This class includes the load instructions LDA X, LDB X, LDC X (X may be either a location or a constant) and the jump instructions JUMP l and JZERO l (l is a label).

Let us give the program that realizes, for example, a LDA X instruction, where X is a memory location.

Program L1:
$i := 0$;
repeat
    $i := i + 1$;
    $t := 0$;
    $A := X[i]$;
    for $j := 1$ **to** $6r + 1$ **do**
        if $A = X[j]$ **then** $t := t + 1$;
until $t \geq 3r + 1$ **or** $i = 6r + 1$

Let us explain briefly the intention behind this strategy, and the hypotheses on which we are relying. We expect some copies among the $6r + 1$ that are stored in X to be wrong, but we expect the majority of them to be correct. We select in turn a value, and load it into the accumulator; next, we check it by comparing it with all the $6r + 1$ values stored. We call *attempt* a single iteration of the **repeat** cycle. If the value that we are checking obtains the majority of the votes, we claim it as the right result. Otherwise, we enter another attempt.

The assignment $A := X[i]$ may itself fail; this happens with probability q. As before, we are free to think that the failure is due either to an error occurred during the data movement, or to an addressing error. The latter may cause the intended accumulator A to be loaded with the contents of a memory cell, other than the right operand. If fact, it is reasonable to think that, when we are writing into an accumulator, the destination of the write is correct, while its origin may not be so.

Let us consider the **for** cycle that checks the correctness of the value just loaded into A. Each comparison $[A : X[j]]$ may fail, again with probability q. As usual, we do not distinguish whether the error is due to the selection of a memory cell other that $X[j]$, or to an incorrect comparison of the right two values. On the other hand, we suppose that the operations involving the indices i and j, and the counter t, are free of error; namely, they may be performed with some special pieces of hardware.

The case of a load operation, having a constant as an operand, as well as the case of a jump instruction, differs slightly from the description given in program $L1$. Let us consider, for example, a JUMP l instruction. The program that realizes it is the following:

Program L2:
$i := 0$;
repeat
    $i := i + 1$;
    $t := 0$;
    $IC := l$;
    for $j := 1$ to $6r + 1$ **do**
        if $IC = L$ **then** $t := t + 1$;
until $t \geq 3r + 1$ **or** $i = 6r + 1$

The program that describes the JZERO l operation is the same, except that it includes an initial test on the accumulator A; it is reasonable to suppose that this test is infallible.

Note that none of the fallible operations performed by program $L1$ or $L2$ may result in an incorrect writing into a place other than A. Thus, no corruption of useful results may take place during the execution of an arbitrary LT instruction.

3 Design of the ERRAM program

Let us see how the execution of an ERRAM program may produce a wrong final result. First, note that the result of an LT instruction is the value left in the destinated accumulator, or in the instruction counter, by the last executed attempt. On the other hand, it is not clear how the result of an ST instruction may be defined. However, we are not directly interested in the behavior of ST instructions, since the output of the ERRAM program is drawn exclusively from the accumulators. Let us call *execution trace* the sequence of the results of the LT instructions performed by the program. Moreover, let us suppose to have a RAM machine with three accumulators and the same set of instructions as our ERRAM. After having recorded the execution trace of the ERRAM program, we run it on the three-accumulator RAM machine, recording the execution trace as well. If the two traces agree, we conclude that the ERRAM program has performed correctly. Otherwise, we consider the first place at which the two sequences disagree. Let us call *divergent* the corresponding LT instruction.

In this section, we first provide conditions that are sufficient to ensure that an LT instruction does not diverge. Next we see how, with an appropriate choice of k, we can ensure that these conditions are satisfied for each executed LT instruction, with an assigned probability $1-\delta$.

Let us recall the bounds on the deviation of a random variable with binomial distribution from the expected value. Let S be the sum of h independent Bernoulli variables, each of parameter t, $0<t<1$. Given a constant λ such that $t<\lambda\leq 1$, it is well known that:

$$prob\,\{S\geq \lambda h\}\leq e^{-h\left(\lambda\ln\frac{\lambda}{t}+(1-\lambda)\ln\frac{1-\lambda}{1-t}\right)}.$$

Since $\lambda>t$, the second term of the exponent is negative. However, we can bound its modulus as follows:

$$(1-\lambda)\ln\left(\frac{1-t}{1-\lambda}\right)<(1-\lambda)\left(\frac{1-t}{1-\lambda}-1\right)=\lambda-t<\lambda,$$

whence we get

$$prob\,\{S\geq \lambda h\}\leq e^{-h\lambda\left(\ln\frac{\lambda}{t}-1\right)}=\left(\frac{et}{\lambda}\right)^{h\lambda}. \tag{1}$$

In the following we will use only this simplified bound.

3.1 A strategy against addressing errors

Recall that we denoted with T the instruction complexity of the two-accumulator RAM program P, on the given instance π. Moreover, let L (resp. S) be the number of LT instructions (resp. ST instructions) performed by $P(\pi)$; clearly, we have $L+S=T$. It will be clear from the following that we need not know even approximately the values of L and S. On the other hand, the design of the ERRAM program needs that we know T beforehand. Later on, we will see how our approach can be immediately extended to the more realistic case in which we know only an upper bound $\tilde{T}$ to T.

It is easy to get convinced that, if the program is very long compared to the size $\overline{M}$ of the memory, addressing errors become a major problem. Namely, let us suppose that there is a location X which is written at the beginning of the program, and is retrieved only at the end. Let us suppose that the number of ST instructions that are executed meanwhile is, for example, $2\overline{M}/q$. Let us focus on a particular cell $X[i]$; recall that $X[i]$ may be erroneously hit by a single write with probability $q/2$. Thus, the expected number of times that $X[i]$ is overwritten, before X is retrieved, is equal to one. Hence, it is clear that the value kept in X will hardly survive. Let us see how this difficulty may be overcome.

We let $\alpha = K/k$, that is, α is the ratio between the number of cells that are available in each location, and the number of cells that we need to write. Clearly, we have $\alpha \geq 1$; for simplicity, we suppose that αM is an integer. We partition the execution of the program in a certain number of *rounds*. Each round, excluding the last one, is made up of two phases. In the first phase, we run the program until it halts, or until $(2\alpha - 1)M$ ST instructions are executed. Since the instructions executed in this phase are drawn from the original two-accumulator RAM program, we give it the name of *program phase*. After the $(2\alpha - 1)M$-th ST instruction has been completed, an interruption mechanism forces the jump to a special routine, consisting of a single cycle in which each location is first loaded into the accumulator C, and is then immediately restored. Note that this routine is parametrized only by the number M of locations needed by the ERRAM program; hence, it is to a large extent independent of the program being run. Moreover, since the additional routine acts exclusively on the reserved accumulator C, which is not used during the program phase, the execution of the second phase does not interfere with the progress of the program. The pair of consecutive instructions (LDC X, STC X), will be called the *refresh* of X; the whole second phase will be called *refresh phase*. It is clear that the last round needs not have a refresh phase.

It is easily checked that no more than $2\alpha M - 1$ ST instructions may be executed before the first refresh of X, or after the last, and that exactly $2\alpha M - 1$ ST instructions are executed between two successive refreshes of X. Thus, $2\alpha M - 1$ is an upper bound to the number of ST instructions that may erroneously overwrite a given location X, due to addressing errors, while X is waiting to be reloaded into some accumulator. Note that this bound is attained whenever X is not stored nor loaded between two refreshes.

The execution of the ERRAM program is broken into $f = \lceil S/(2\alpha - 1)M \rceil$ rounds. Since, excluding the last round, we have $f - 1 \leq S/(2\alpha - 1)M$ rounds that include a refresh phase, up to $S/(2\alpha - 1)$ ST instructions and $S/(2\alpha - 1)$ LT instructions may be introduced in addition to those performed by the original RAM program. Denoting with L_1 and S_1 the augmented counts of LT instructions and ST instructions, respectively, we get

$$L_1 \leq L + \frac{S}{2\alpha - 1} \leq L + S = T, \tag{2}$$

$$S_1 \leq \frac{2\alpha}{2\alpha - 1} S. \tag{3}$$

3.2 Choosing the amount of redundancy

Let us focus an arbitrary LT instruction, having a memory location as an operand. For example, we take a LDA X instruction (see program $L1$). We assume that some previous ST instruction acted on X, since we are not interested in the behavior of LT instructions acting on locations that have not been initialized.

Let us recall that a single iteration of the **repeat** cycle of program $L1$ was called an *attempt*. Let U be the random variable, counting the number of attempts that are executed by the program $L1$. Clearly, we have $1 \leq U \leq 6r + 1$. In particular, the number of assignments $A := X[i]$ that are performed is precisely equal to U. Let us suppose that, after the completion of the instruction, we make $6r + 1 - U$ independent coin tosses, whose outcome equals "tail" with probability q. We consider the following conditions:

E_1 : let W_1 count the number of writes that failed during the last ST instruction, acting on X. Then, we require that $W_1 \leq r$;

E_2 : let W_2 count the number of memory cells belonging to X that have been overwritten due to addressing errors, since the last ST instruction, acting on X. Then, we require that $W_2 \leq r$;

E_3 : let us consider the number of assignments $A := X[i]$ that fail, throughout the U attempts performed, and the number of tails obtained in the $6r+1-U$ subsequent coin tosses. Let us sum these two terms to obtain a random variable W_3. Then, we require that $W_3 \leq r$;

E_4 : let $W_4^{(i)}$ count the number of comparisons that failed in the i-th attempt performed. Then, we require that $W_4^{(i)} \leq r$, for $1 \leq i \leq U$.

Let us analyze the implications of the conditions E_1 to E_4. Note first that, if E_1 and E_2 are satisfied, then at least $4r+1$ of the copies contained in X are equal to each other. Let v be their common value. Let us prove that, if in addition we assume E_4, then v is correctly claimed as result if and only if it is ever loaded into A. Namely, suppose that an attempt loads A with the value v. Then, t may fail to be incremented either if A is compared with a cell that does not contain v (due to $E1$ and $E2$, this happens at most $2r$ times), or if the comparison with a cell containing v fails (due to $E4$, this happens at most r times). Thus, at the end of the attempt we have $t \geq 3r+1$. On the other hand, let us suppose that an attempt loads A with a wrong value $v_1 \neq v$. Then, even if all the wrong copies are equal to v_1, t cannot be incremented more than $2r$ times due to correct comparisons. Moreover, by $E4$, t is incremented at most r times due to incorrect comparisons. Thus, it cannot reach the threshold value $3r+1$, such that the **repeat** cycle is exited if and only if $i = 6r+1$. Note that in this case, by the discussion above, v has never been loaded into A. Let the random variable H denote the index of the first attempt that loads A correctly; if there is no such attempt, we set $H = 6r+2$. Assuming E_1, E_2 and E_3, we have $H \leq 3r+1$. Thus, the right value is loaded at least once into A.

Finally, let us note that the above reasoning also shows that, if E_1, E_2 and E_4 are satisfied, then $U = H$. Combining all these observations, we get the following result:

Lemma 1 *If an LT instruction fulfills the conditions E_1 to E_4, then it cannot diverge. Moreover, the right result is claimed precisely when it is first loaded into A.*

Let us give a lower bound to the probability that E_1 is satisfied. Clearly, W_1 is the sum of k independent Bernoulli random variables, each of parameter q. We use (1), setting $\lambda = 1/6$, $t = q$ and $h = 6r+1$, to get a lower bound to the probability that condition E_1 is satisfied. Letting

$$\beta = (6eq)^{r+\frac{1}{6}}, \tag{4}$$

we get

$$prob\{E_1\} \geq 1 - \beta. \tag{5}$$

Let us turn to E_2; since W_2 is independent of W_1, we have that E_2 is independent of E_1. As we have already seen, no more than $2M\alpha - 1$ ST instructions directed to locations other than X may have been performed since X was last written, or since the beginning of the program. Since the probability that E_2 is satisfied decreases, as the number of ST instructions that can hit X increases, we may suppose that these ST instructions are precisely $2M\alpha - 1$ or, for simplicity, $2M\alpha$. Recall that each location X includes an initial segment of $6r+1$ useful cells, while the cells $X[6r+2]$ to $X[K]$ are left unused. Any ST instruction that does not act on X, can hit the segment of useful cells of X with probability $q/2M\alpha$. Since W_2 is the sum of independent Bernoulli variables, we may apply (1) setting $h = 2M\alpha$, $t = q/2M\alpha$, and $\lambda = (6r+1)/12M\alpha$. We obtain

$$prob\{E_2\} \geq 1 - \beta \tag{6}$$

We now turn to E_3. Note that W_3 is the sum of $6r+1$ independent Bernoulli variables, each of parameter q. Recall that the definition of W_3 makes use of the random variable U, which in turn depends on W_1, W_2 and the $W_4^{(i)}$. On the other hand, the distribution of W_3 does not depend on any of these variables. It is easily seen that $prob\{E_3\} = prob\{E_1\}$, hence

$$prob\{E_3\} \geq 1 - \beta. \tag{7}$$

Let us finally consider E_4. Clearly, $W_4^{(i)}$ is independent of $W_4^{(j)}$, for $j \neq i$; it is also independent of W_1, W_2 and W_3. However, the number of variables $W_4^{(i)}$ that we consider is equal to U; this variable depends on W_1, W_2, W_3, and also on all of the $W_4^{(i)}$. Let $\mathcal{G}$ denote the set of indices j such that one of the following things happen:

- $X[j]$ was written with a wrong value;
- $X[j]$ was hit by some addressing error;
- the assignment $A := X[j]$ failed, if $j \leq R$, or the $(j-R)$-th toss of the coin responded "tail", if $j > R$.

From the definition of the set $\mathcal{G}$ it follows that, if we make the assumption $|\mathcal{G}| = g$, $0 \leq g \leq 6r+1$, then $\mathcal{G}$ ranges over all the sets of g indices with uniform probability. This follows from evident symmetry considerations. Recall that H denotes the minimum index that does not belong to $\mathcal{G}$, provided that $|\mathcal{G}| \leq 6r$. Note that this is true whenever E_1 to E_3 are satisfied, since then we have $|\mathcal{G}| \leq 3r$. For $0 \leq h \leq |\mathcal{G}| \leq 6r$, we have:

$$prob\{H = h+1 | \ |\mathcal{G}| = g\} = \frac{\binom{6r-h}{g-h}}{\binom{6r+1}{g}}.$$

Namely, the first h indices must belong to $\mathcal{G}$, $h+1$ must lie outside $\mathcal{G}$, and the other $g-h$ elements of $\mathcal{G}$ are drawn from the remaining $6r-h$ positions with uniform probability.

Recall that, if E_1 to E_4 are satisfied, then by lemma 1 we have $U = H$, whence:

$$prob\{E_4 \mid E_1, E_2, E_3\} = prob\left\{W_4^{(1)} \leq r, \ldots, W_4^{(H)} \leq r \mid E_1, E_2, E_3\right\}.$$

Since the distribution of each of the $W_4^{(i)}$ is the same as the distribution of W_1, we get

$$prob\{E_4 \mid E_1, E_2, E_3\} \geq \sum_{0 \leq h \leq 3r} (1-\beta)^{h+1} prob\{H = h+1 \mid E_1, E_2, E_3\}$$

$$= \sum_{0 \leq h \leq 3r} (1-\beta)^{h+1} \sum_{h \leq g \leq 3r} prob\{H = h+1 \mid\mid \mathcal{G} \mid= g\}\, prob\{\mid \mathcal{G} \mid= g \mid E_1, E_2, E_3\}$$
$$= (1-\beta) \sum_{0 \leq g \leq 3r} prob\{\mid \mathcal{G} \mid= g \mid E_1, E_2, E_3\} \sum_{0 \leq h \leq g} (1-\beta)^h \frac{\binom{6r-h}{g-h}}{\binom{6r+1}{g}}.$$

Since

$$\sum_{0 \leq h \leq g} \frac{\binom{6r-h}{6r-g}}{\binom{6r+1}{g}} (1-\beta)^h \geq \sum_{0 \leq h \leq g} \frac{\binom{6r-h}{6r-g}}{\binom{6r+1}{g}} (1-h\beta) = 1 - \beta \sum_{0 \leq h \leq g} \frac{h\binom{6r-h}{6r-g}}{\binom{6r+1}{g}},$$

and

$$\begin{aligned}
\sum_{0 \leq h \leq g} h \binom{6r-h}{6r-g} &= \sum_{1 \leq h \leq g} \binom{6r-h}{6r-g} + \sum_{2 \leq h \leq g} \binom{6r-h}{6r-g} + \ldots + \sum_{g \leq h \leq g} \binom{6r-h}{6r-g} \\
&= \binom{6r}{6r-g+1} + \binom{6r-1}{6r-g+1} + \ldots + \binom{6r-g+1}{6r-g+1} \\
&= \binom{6r+1}{6r-g+2},
\end{aligned}$$

we get

$$\begin{aligned}
prob\{E_4 \mid E_1, E_2, E_3\} &\geq (1-\beta) \sum_{0 \leq g \leq 3r} prob\{\mid \mathcal{G} \mid= g\} \left[1 - \beta\left(\frac{g}{6r-g+2}\right)\right] \\
&\geq (1-\beta)\left(1 - \beta\frac{3r}{3r+2}\right) \\
&\geq (1-\beta)^2.
\end{aligned}$$

Taking the bounds (5) to (7) into account, and recalling that E_1, E_2 and E_3 are independent, we have $prob\{E_1, E_2, E_3\} \geq (1-\beta)^3$. Hence, we finally get

$$prob\{E_1, E_2, E_3, E_4\} \geq (1-\beta)^5.$$

Since $1-(1-\beta)^5 < 5\beta$, recalling the definition (4) of β, we get

Lemma 2 *Let Q_1 be the probability that an LT instruction, having a location as an operand, fails to load the intended value. Then, we have*

$$Q_1 < 5\,(6eq)^{r+\frac{1}{6}}.$$

Let us see what happens if the LT instruction has a constant or a label as an operand. Let us focus, for example, on the JUMP l instruction described in program $L2$. Let P_S (resp. P_F) the probability of accepting the right result (resp., a wrong result) in a given attempt. It is easily seen that the probability Q_2 that the instruction loads a wrong result satisfies:

$$Q_2 = P_F + (1-P_S-P_F)P_F + (1-P_S-P_F)^2 P_F + \ldots = \frac{P_F}{P_S+P_F} \leq \frac{P_F}{P_S}. \quad (8)$$

Now, the right result is claimed whenever the assignment $IC := l$ succeeds and, at the same time, no more than $3r$ comparisons $[IC : l]$ fail. Note that the two events are independent; to bound the probability that the latter event takes place, we use (1) setting $\lambda = 1/2$, $h = 6r+1$, $t = q$. We obtain

$$P_S \geq (1-q)(1-(2eq)^{3r+\frac{1}{2}}). \quad (9)$$

On the other hand, a wrong result may be claimed only if $IC := l$ fails and, at the same time, $3r+1$ or more comparisons fail; hence,

$$P_F \leq q(2eq)^{3r+\frac{1}{2}}. \quad (10)$$

We want now to bound the error probability of the program by applying the union bound on the error probabilities of the single instructions. Incidentally, let us note that the events E_1 referring to two different LT instructions may not be independent; namely, they even coincide when the two LT instructions are preceded by the same ST instruction. The events E_2 referring to different instructions are easily seen not to be independent, too. However, note that the union bound can be applied to events that are not independent as well.

Note that from (8), (9) and (10), it easily follows that $Q_2 < Q_1$. We can then suppose, as worst case, that all the L_1 LT instructions performed by the ERRAM program act on memory locations. Incidentally, let us note that all the LT instructions added in the refresh phases are of this type. Since, by (2), we have $L_1 \leq T$, the probability that there is one divergent LT instruction is bounded by $T\,Q_1$. Using lemma 2, and equating $5T\beta$ to δ, we get the following result:

Theorem 1 *The choice*

$$k = 6\,\frac{\ln T + \ln\left(\frac{5}{\delta}\right)}{-\ln(6eq)}$$

ensures that, with probability at least $1-\delta$, the ERRAM program produces the right result.

4 The expected execution time

To complete the complexity analysis of our strategy, we have to find the expected number of attempts that are executed by an arbitrary LT instruction. Denote with $\mathcal{E}$ the event that, for each LT instruction, acting on a memory location, the conditions E_1 to E_4 are satisfied. If k is chosen according to theorem 1, we know that $prob\{\mathcal{E}\} > 1-\delta$. Recall that, by lemma 1, the condition $\mathcal{E}$ is sufficient to ensure that no instruction diverges, and thus, that the program behaves well. However, outside $\mathcal{E}$ we have no longer this guarantee. If one instruction diverges, the program begins to work on data that are incorrect, and we cannot predict its behavior. In particular, its instruction complexity is no longer kept under control. Hence, if $\mathcal{E}$ is not satisfied, there is little interest in bounding the expected complexity of a single instruction.

Let us first focus on LT instructions acting on memory locations. Recall from section 3 the definition of the random variables H and U, and the definition of the set $\mathcal{G}$. With the same considerations as in that section, we get:

$$prob\{U > h \mid E_1, E_2, E_3, E_4\} = \sum_{0 \leq g \leq 3r} prob\{H > h \mid\, \mid \mathcal{G} \mid = g\}\, prob\{\mid \mathcal{G} \mid = g \mid E_1, E_2, E_3, E_4\}\,.$$

Now, we have

$$prob\{H > h \mid\, \mid \mathcal{G} \mid = g\} = \frac{\binom{6r+1-h}{g-h}}{\binom{6r+1}{g}},$$

since H exceeds h if and only if the indices 1 to h belong to $\mathcal{G}$. We then obtain

$$\mathbf{E}\left[U \mid E_1, E_2, E_3, E_4\right] = \sum_{0 \leq g \leq 3r} prob\{\mid \mathcal{G} \mid = g \mid E_1, E_2, E_3, E_4\} \sum_{0 \leq h \leq g} \frac{\binom{6r+1-h}{6r+1-g}}{\binom{6r+1}{g}}.$$

Since

$$\sum_{0 \leq h \leq g} \frac{\binom{6r+1-h}{6r+1-g}}{\binom{6r+1}{g}} = \frac{\binom{6r+2}{6r+2-g}}{\binom{6r+1}{g}} = \frac{6r+2}{6r+2-g},$$

using the fact that $g \leq 3r$, we obtain

$$\mathbf{E}\,[U \mid E_1, E_2, E_3, E_4] \leq 2\,\frac{3r+1}{3r+2} < 2.$$

Let us turn to LT instructions, acting on constants or labels. Recall that, in such instructions, an attempt claims some result with probability $P_S + P_F > P_S$. Hence, the expected number of attempts that are executed is less than $1/P_S$, that is far less than 2 (see (9)).

From these results it follows that, denoting with $\overline{T}$ the operation complexity of the ERRAM program, and using equations (2) and (3), we have

$$\begin{aligned}\mathbf{E}\left[\overline{T} \mid \mathcal{E}\right] &= k\,(\mathbf{E}\,[U \mid E_1, E_2, E_3, E_4]\,L_1 + S_1) \\ &\leq k\,(2L_1 + S_1) \leq 2k\left(L + \frac{1+\alpha}{2\alpha-1}\,S\right). \end{aligned} \tag{11}$$

Since $\alpha \geq 1$, we get

Theorem 2 *If k is chosen according to theorem 1, then there exists a condition $\mathcal{E}$, with* $\text{prob}\,\{\mathcal{E}\} > 1 - \delta$, *such that if $\mathcal{E}$ if fulfilled, then the ERRAM program produces the right result, and its operation complexity satisfies*

$$\mathbf{E}\left[\overline{T} \mid \mathcal{E}\right] \leq 24\,T\,\frac{\ln T + \ln\left(\frac{5}{\delta}\right)}{-\ln(6eq)}.$$

Let us list some improvements that can be obtained with very little effort:

- if we do not know the exact value of T, but only an upper bound $\tilde{T}$ to it, the choice suggested by theorem 1 is modified as follows:

  $$k = 6\frac{\ln \tilde{T} + \ln\left(\frac{5}{\delta}\right)}{-\ln(6eq)},$$

  whence

  $$\mathbf{E}\left[\overline{T} \mid \mathcal{E}\right] \leq 24T\,\frac{\ln \tilde{T} + \ln\left(\frac{5}{\delta}\right)}{-\ln(6eq)};$$

- if we have $\alpha \geq 2$ (that is, at least one half of the memory is left unused), from (11) we easily obtain that the upper bound contained in theorem 2 is halved;

- if $T \leq 2\alpha M$, such that no refresh phase has to be added, equation (11) is substituted by the following:

$$\mathbf{E}\left[\overline{T} \mid \mathcal{E}\right] = k(\mathbf{E}\left[U \mid E_1, E_2, E_3, E_4\right] L + S) \leq k(2L + S) \leq 2kT.$$

In this case the upper bound of theorem 2 is halved as well.

5 Conclusions

In this work we have studied the design of programs that work on a fallible RAM machine, with assigned confidence. We have defined the ERRAM model, where each operation fails with probability q; the failures are considered to be independent. We have taken an arbitrary RAM program, assuming to know in advance its instruction complexity T, or at least an upper bound to it. Given a confidence parameter $1 - \delta$, we have shown how it can be transformed into an ERRAM program, that works with the desired confidence. To achieve this, we have made extensive use of redundancy. Namely, in the ERRAM program each operation is repeated several times, and each stored value is held in multiple copies. The latter precaution has to be taken because of the presence of addressing errors, that may overwrite useful values. The instructions that read from the memory have an embedded majority voting mechanism, whose implementation is accomplished by the use of a minimal quantity of infallible hardware. We have given an upper bound to the number of times that each operation has to be repeated to get the desired confidence on the final result. For any fixed failure probability $q < 1/6e$, the expected number of repetitions turned out to be $O(T/\delta)$. Similarly, we have shown that storing $O(T/\delta)$ copies of each value is sufficient for our purposes.

References

[1] A. Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[2] M. Blum and S. Kannan, Designing programs that check their work, *Proc. 21st ACM Symposium on the Theory of Computing* (1989), 86-95.

[3] M. Blum, M. Luby, and R. Rubinfeld, Self-testing and self-correction programs with applications to numerical problems, *Proc. 22st ACM Symposium on the Theory of Computing* (1990), 73-83.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, Checking the correctness of memories, *Proc. IEEE Symposium on the Foundations of Computer Science* (1991), 90-99.

[5] R.S. Borgstrom and S.R. Kosaraju, Comparison-based search in the presence of errors, *Proc. ACM Symposium on the Theory of Computing* (1993), 130-136.

[6] M.A. Breuer and A.D. Friedman, *Diagnosis and Reliable Design of Digital Systems,* Computer Science Press, Woodland Hills, CA, 1976.

[7] D.E. Knuth, *The Art of Computer Programming,* vol. 3: *Sorting and Searching,* Addison-Wesley, Reading, MA, 1973.

[8] K.B. Lakshmanan, B. Ravikumar, and K. Ganesan, Coping with erroneous information while sorting, *IEEE Transactions on Computers,* **40** (1991), 1081-1084.

[9] F. Luccio and A. Pedrotti, Combinatorial algorithms in the presence of random errors, *Dipartimento di Informatica, University of Pisa,* Technical Report TR-13/95 (1995).

[10] N. Pippenger, Analysis of error correction by majority voting, in *Advances in Computing Research,* Volume 5, JAI Press, 1989, pp. 171-198.

[11] N. Pippenger, Reliable computation by formulas in the presence of noise. *IEEE Transactions on Information Theory,* **34** (1988), 194-197.

[12] N. Pippenger, On networks of noisy gates, *Proc. IEEE Symposium on the Foundations of Computer Science* (1985), 30-38.