A Synergetic Use of Bloom Filters for Error Detection and Correction

Pedro Reviriego, Salvatore Pontarelli, Juan Antonio Maestro, and Marco Ottavi

***Abstract*— Bloom filters (BFs) provide a fast and efficient way to check whether a given element belongs to a set. The BFs are used in numerous applications, for example, in communications and networking. There is also ongoing research to extend and enhance BFs and to use them in new scenarios. Reliability is becoming a challenge for advanced electronic circuits as the number of errors due to manufacturing variations, radiation, and reduced noise margins increase as technology scales. In this brief, it is shown that BFs can be used to detect and correct errors in their associated data set. This allows a synergetic reuse of existing BFs to also detect and correct errors. This is illustrated through an example of a counting BF used for IP traffic classification. The results show that the proposed scheme can effectively correct single errors in the associated set. The proposed scheme can be of interest in practical designs to effectively mitigate errors with a reduced overhead in terms of circuit area and power.**

***Index Terms*— Bloom filters (BFs), error correction, soft errors.**

I. Introduction

Bloom filters (BFs) provide a simple and effective way to check whether an element belongs to a set [1]. They are used in many networking applications [2] as well in computer architectures [3]. The BFs are also used in large databases (e.g., Google Bigtable uses it to reduce the disk lookups [4]).

The basic structure of BFs has also been extended over the years. For example, counting BFs (CBFs) were introduced to allow removal of elements from the BF [5]. To optimize the transmission over the network, another extension known as compressed Bloom filters was proposed [6]. Recently Bloom filter (Biff) codes that are based on BFs have been proposed to perform error correction in large data sets [7].

In most cases, BFs are implemented using electronic circuits [8], [9]. The contents of a BF are commonly stored in a high speed memory and required processing is done in a processor or in dedicated circuitry. The set used to construct the BF is also commonly stored in a lower speed memory [10].

The reliability of electronic circuits is becoming a challenge as technology scales. Errors caused by interferences, radiation, and other effects become more common. Therefore, mitigation techniques are used at different levels to ensure that the circuits continue to operate reliably [11]. For BF implementation, memories are a critical element. For memories, permanent errors and defects are commonly corrected using spare rows and columns [12]. However, soft errors caused for example by radiation can affect any memory cell changing its value during circuit operation. Soft errors do not produce damage to the memory device that continues to operate correctly but has the wrong value in the affected cell [13]. To deal with soft errors, the use of a per word parity bit or more advanced error correction codes (ECCs) has been common in memories for many years [14].

The BFs have also been proposed to mitigate errors in electronic circuits. For example, in [15], a BF is used to identify the faulty words in a nanomemory. In [16], the use of a CBF is proposed to detect and correct errors in content addressable memories (CAMs). In this case, the CBF is used in parallel with a CAM and the objective is to detect errors in the CAM entries. This is done by checking the results of the CAM and the CBF to ensure that they are consistent. Once an error is detected, a correction procedure is initiated to restore the correct value in the affected CAM entry using an external copy of its contents. In both cases, the BFs are added explicitly and only to detect and/or correct errors and are not present in the original design. The same applies to Biff codes in which the extended BFs are only used for error correction. That is in those cases, the BF is not in the original system and it is explicitly added to protect against errors. This is different from the reuse of existing BFs that are already present in the system to also provide error correction which to the best of our knowledge has not been studied.

In this brief, a scheme to exploit existing CBFs to additionally implement error detection and correction in the elements of the set associated with the CBF is presented. The approach is based on the concept of algorithmic-based fault tolerance (ABFT), which proposes to reuse existing properties or elements of the system to implement fault tolerance with a lower cost [17]. In the line of ABFT, the proposed scheme enables a synergetic reuse of existing CBFs for error detection and correction. The scheme assumes that the elements of the set are stored in a memory protected with a per word parity bit and the CBF is used to implement the correction of single bit errors. The effectiveness of the scheme is illustrated using a traffic classification case study.

The basic ideas behind the proposed technique can also be applied when the elements of the set are stored in a memory protected with more advanced ECCs. In addition, a simplified version of the proposed approach can also be used for traditional BFs but in that case, the percentage of errors that can be corrected is much lower. The exploration of the scheme extension to these cases is left for future work.

The rest of this brief is organized as follows. In Section II, an overview of BFs and CBFs is provided. In Section III, the proposed scheme is presented. Then, in Section IV, a case study of traffic classification is used to illustrate the effectiveness and benefits of the scheme. Finally, conclusions are drawn in Section V.

II. Overview of BFs

A BF is constructed using a set of k hash functions to access an array of m bits. The hash functions $h_1, h_2, \ldots, h_k$ map an input element x to one of the m bits. The following two operations are defined in a BF.

1) *Insertion:* To insert an element x in the BF, the bits in the array that correspond to the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are set to one.

Manuscript received September 6, 2013; revised January 23, 2014; accepted March 10, 2014. Date of publication March 27, 2014; date of current version February 20, 2015. This work was supported in part by the Spanish Ministry of Science and Education under Grant AYA2009-13300-C03. This brief is part of a collaboration in the framework of COST ICT Action 1103 “Manufacturable and Dependable Multicore Architectures at Nanoscale”.

P. Reviriego and J. A. Maestro are with the Universidad Antonio de Nebrija, Madrid E-28040, Spain (e-mail: previrie@nebrija.es; jmaestro@nebrija.es).

S. Pontarelli is with the National Interuniversity Consortium for Telecommunications, Rome 00133, Italy (e-mail: pontarelli@ing.uniroma2.it).

M. Ottavi is with the University of Rome Tor Vergata, Rome 00133, Italy (e-mail: ottavi@ing.uniroma2.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2014.2311234

2) *Query:* To query for an element x in the BF, the bits in the array that correspond to the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are read and if and only if all of them are one, the element is considered to be in the BF.

This operation guarantees that if an element has been added to the BF, it will be found when a query for it is done. However, a BF can produce false positives when a query for an element that has not been added to the BF is done. That is an element is incorrectly classified as being stored in the BF when in fact is not in the element set. This can occur if other elements have set to one the positions that correspond to the hash values of that element.

As discussed in [2], assuming that the hash functions are uniformly distributed, after inserting n elements in the BF, the probability $p_0(n)$ that a given bit in the array is zero can be approximated as

$$p_o(n) \cong \left(1 - \frac{1}{m}\right)^{kn} \cong e^{-\frac{k \cdot n}{m}}. \quad (1)$$

Therefore, the probability of a false positive can be approximated as

$$p_{fp}(n) \cong (1 - p_o(n))^k \cong \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k. \quad (2)$$

It can be observed that p_{fp} depends on $(1 - p_0(n))$ and k. The first expression gives the probability that an element in the CBF has a value different than zero and is commonly known as the load factor. The load factor gives an indication of how many elements have been inserted in the CBF and also of the false positive probability. The load factor will be used in the experiments presented in this brief and is defined as

$$lf \cong (1 - p_o). \quad (3)$$

A problem with BFs is that elements cannot be easily removed. This is because a position with a one in the array can be shared by several elements and thus clearing the $h_1(x), h_2(x), \ldots, h_k(x)$ positions for an element x may also affect other elements in the BF. To address this issue, CBFs which are a generalization of BFs were introduced. In a CBF, the array of m bits is replaced with an array of integers of b bits and the operations are defined as follows.

1) *Insertion:* To insert an element x in the CBF, the integers in the array that correspond to the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are incremented by one.
2) *Query:* To query for an element x in the CBF the integers in the array that correspond to the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are read and if and only if all of them are larger than zero the element is considered to be in the CBF.
3) *Removal:* To remove an element x from the CBF, the integers in the array that correspond to the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are decremented by one.

The use of integers instead of bits allows the removal of elements as now each position in the array stores the number of elements that share that position. The false positive rate of a properly dimensioned CBF is the same as that of a standard BF.

III. Proposed Scheme

The proposed scheme is based on the observation that a CBF, in addition to a structure that allows fast membership check to an element set, is also in a way a redundant representation of the element set. Therefore, this redundancy could possibly be used for error detection and correction.

To explore this idea, a common implementation of CBFs where the elements of the set are stored in a slow memory and the CBF is stored in a faster memory is considered. In particular, it is assumed that the elements of the set are stored in DRAM while the CBF is stored in a cache [10]. The reasoning behind this is that the CBF is accessed frequently and needs a fast access time to maximize performance, while the elements of the set are only accessed when elements are read, added or removed and therefore the access time is not an issue. It should also be noted that when the entire element set is stored in a slow memory, no incorrect deletions can occur as they would be detected when removing the element from the slow memory. Therefore, the false negatives issue in CBFs discussed in [18] is not a concern in our case.

Typically, memories are protected with a per word parity bit or with a single bit error correction code [14]. This is based on the observation that most errors affect a single bit or even if they affect multiple bits, the errors can be spread among different words by the use of interleaving [19]. In addition, soft errors are rare events so that the time between errors is typically large [13]. The arrival rate for terrestrial applications is in the order of at least days or weeks and therefore, it is commonly assumed that errors are isolated. That is, by the time a soft error arrives any previous soft error has been corrected or detected. This is an assumption that is needed, for example, when single bit error correction codes are used.

In the following, one of these two most common protection options is used. In particular, it is assumed that both the DRAM and the cache are protected with a per word parity bit that can detect single errors. As when using single bit error correction codes, it is also assumed that errors are isolated.

The goal for this implementation is to achieve the correction of single bit errors using the CBF. That is, the CBF would enable single bit error correction without incurring in the cost of adding an ECC to the memories.

The first step to achieve error correction is to detect errors. This is done by checking the parity bit when accessing either the DRAM or the cache. To ensure earlier detection of errors, the use of scrubbing to periodically read the memories could be considered [20]. Once an error is detected, a correction procedure is triggered.

If the error occurs in the CBF, it can be corrected by clearing the CBF and reconstructing it using the element set. If the error occurs in the element set, the procedure is more complex and can be divided in two phases that are described in the following sections. The idea is that the simpler and faster procedure is used first and only when it is unable to correct the error, the second more complex error correction procedure is used subsequently.

A. Simple Procedure for the Correction of Errors in the Element Set

To present the simple correction procedure, let us assume that a single bit error affects element x and that it is detected using the parity bit. Therefore, x_e is read from the memory. The correct value x has to be x_e if the error affected the parity bit. If the error affected the ith data bit, the correct value will be $x_{em}(i)$ where $x_{em}(i)$ is the value read (x_e) with the ith bit inverted. To determine which of those is in fact the correct value x, the candidates [x_e and all the $x_{\mathrm{em}}(i)$] can be tested for membership to the CBF. If only one of the candidates is found in the CBF, then no false positives have occurred and the value found is the correct one. Instead, if more than one candidate is found, the procedure is unable to find the correct value due to the occurrence of false positives. In this case, the advanced procedure described in Section III-B must be used. This simple and fast procedure requires only $l + 1$ queries to the CBF, where l is the number of bits in each element of the set. However, the correction rate that can be achieved depends on the false positive rate of the CBF. In particular, the probability that an error can be corrected using this procedure can be approximated as

$$P_{\text{correction}} \cong \left(1 - p_{fp}\right)^{l} \tag{4}$$

which is the probability that none of the l candidates that are not x return a false positive on a query. The above formula does not take into account that some elements on the set may only differ in one or two bits from another element in the set. In that case, the proposed correction procedure may fail as one of the candidates may also be a valid element and therefore, the advanced procedure must be used. This effect will be heavily dependent on the properties of the elements in the set and will therefore be application dependent. In any case, to account for it, the probability given by (4) should be used as an upper bound rather than an approximation.

B. Advanced Procedure for the Correction of Errors in the Element Set

When the simple procedure described in Section III-A cannot correct an error, a more advanced technique can be used. The correction process starts by making a copy of the CBF in DRAM memory. Then, all the elements in the set except for the erroneous one are removed from the CBF. This will leave a CBF with only the values that correspond to the original value of the element x. Once that is done, the candidates [x_e and all the $x_{em}(i)$] can be queried over the CBF that has only x as an entry. As in the previous procedure, if only one of the candidates matches the CBF, that is the correct value. If more than one candidate matches the CBF then the error cannot be corrected. The probability that a given value x and another value y produce exactly the same values of the hash functions $h_1, h_2, \ldots, h_k$ can be approximated as

$$P_{\text{CBF}(x)=\text{CBF}(y)} \cong \frac{k!}{m^k}. \tag{5}$$

Therefore, the correction probability for this advanced procedure can be approximated as

$$P_{\text{correction}} \cong \left(1 - \frac{k!}{m^k}\right)^{l} \tag{6}$$

which will be very close to 100% in many practical scenarios as m is typically large.

The increased correction rate comes at the cost of a more complex correction procedure that needs the replication of the CBF, the removal of all the entries except the erroneous one $(n-1)$, and finally the query for the $l+1$ candidates. However, as soft errors are rare events, and the procedure is only needed when the simple procedure presented before cannot correct an error, the scheme can be useful in real applications.

Finally, it must be noted that when the CBF experiences overflows in the counters, this second technique cannot be used. This should not be a big issue as the overflow probability is typically very low when four bits per counter are used [21]. In any case, since overflows are detected once that occurs, this second procedure can be disabled.

The same scheme could be applied to a memory protected with a single error correction double error detection (SEC-DED) code to correct double errors. In that case, the simple procedure would be of little use in most cases as the number of candidates to test is $\binom{l+1}{2}$ and therefore, it is unlikely that none of them gives a false positive. The advanced correction procedure on the other hand will be able to correct the errors with a probability close to one. The detailed evaluation of this scenario is left for future work.

IV. Evaluation

The proposed scheme has been evaluated using a real example of a CBF used to speed-up traffic classification. Pairs of IP addresses (source and destination host) are stored in a multiple hash table [10], and a set of CBFs allows to know in which table the element is stored, providing a fast retrieve of the value associated to the element. Since IP version four is considered, the size of the elements is 64 bits. To test the effectiveness of the scheme, the CBF is filled using values from real data traces that are publicly available [22]. Different CBF sizes and load factors were tested. In particular, values of $m = 16, 32$ and 64K were considered and for each of them load factors of 0.12, 0.25 and 0.5 were tested. The number of hash functions k was also varied between 4 and 8. As explained before, the load factor here refers to the probability that a position in the CBF is different from zero that is $1 - p_0$ given in (1). For each configuration, once the CBF was filled to the desired load level, a single bit error was introduced in one of the elements and the error correction procedures described in Section III were applied. This process was repeated 10 000 times so that 10 000 random single bit errors were tested in each case. First, the simple error correction procedure and if it is unable to correct the error, the advanced procedure is used. In all cases, the single bit errors were corrected. This can be explained as the probability that an error is not corrected for the advanced procedure is in the worst case (m = 16K and k = 4) approximately 2.1×10^{-14} [obtained using (6)]. This shows that in practical terms most errors will be corrected. The effectiveness of the simple error correction procedure greatly depends on the load of the CBF and it is shown in Figs. 1–3 (note that errors not corrected by this procedure

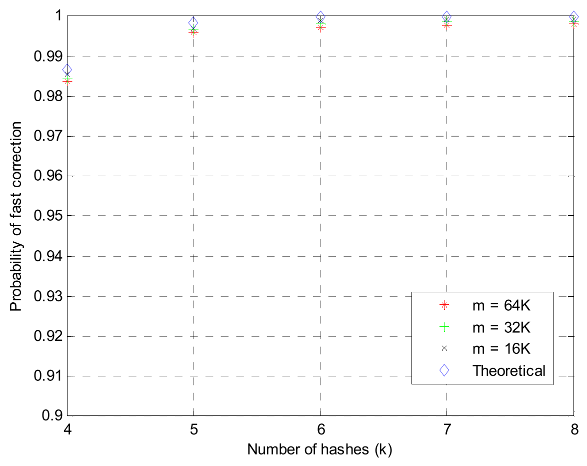

Fig. 1. Probability of error correction using the simple procedure for different values of m and k and comparison with (4) when load is 0.12.

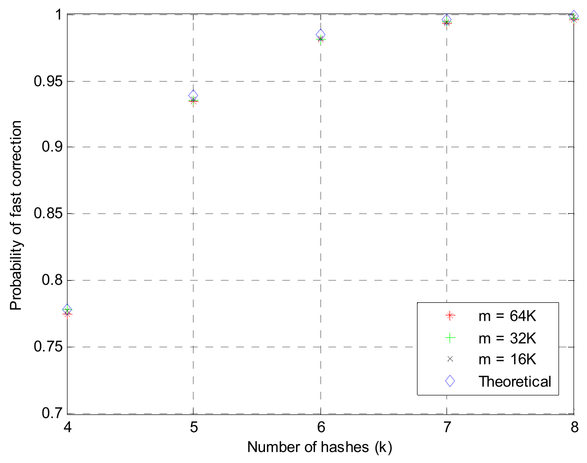

Fig. 2. Probability of error correction using the simple procedure for different values of m and k and comparison with (4) when load is 0.25.

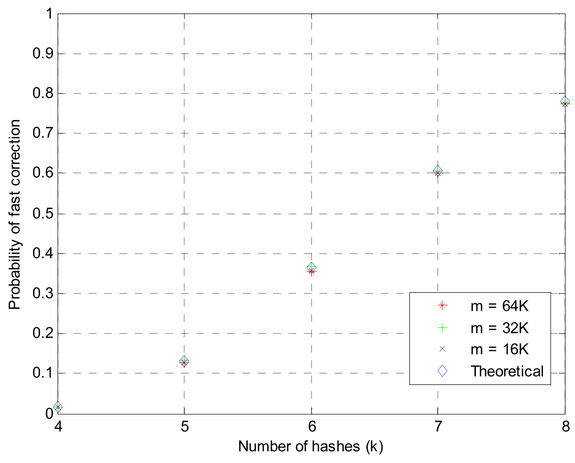

Fig. 3. Probability of error correction using the simple procedure for different values of m and k and comparison with (4) when load is 0.50.

are subsequently corrected by the advanced procedure as described before). It can be observed that for low loads the scheme can correct most errors while at a load of 0.5 it is only effective when the number of hashes is large. In all the cases, the upper bound given by (4) is close to the actual results obtained in simulation. These observations suggest a possible enhancement of the correction procedure. Namely, when the load is high a number of entries can be removed and when a low load is achieved, the simple procedure can be used to correct the errors. In this way, the complete advanced procedure will be used only for a small fraction of the errors. The study of this refinement is left for future work.

It can be observed that the results do not depend on m. This can be explained as for values of m much larger than one, as those commonly used in practical applications, the CBF is close to the asymptotic behavior in all cases [2].

Finally, the cost savings obtained by using the proposed scheme can be estimated as the implementation of an SEC code on a 64 bit element requires 7 bits. Therefore, as with only a parity bit and the CBF SEC can be achieved, the savings would be 6 bits per element set or roughly 10% of the memory storage required for the element set. A different way to look at the benefits is that SEC can be implemented in the element set when the system memory is protected only with a per word parity bit. That is, reliability can be increased without adding new hardware resources.

V. Conclusion

In this brief, a new application of BFs has been proposed. The idea is to use the BFs in existing applications to also detect and correct errors in their associated element set. In particular, it is shown that CBFs can be used to correct errors in the associated element set. This enables a cost efficient solution to mitigate soft errors in applications which use CBFs.

The configuration considered in this brief is that of a memory protected with a per word parity bit for which it is demonstrated that the CBF can be used to achieve single bit error correction. This shows how existing CBFs can be used to achieve error correction in addition to perform their traditional membership checking function.

The general idea can also be used when the memory is protected with more advanced codes. For example, if an SEC-DED code is used, the CBF could be used to correct double errors. In addition, the simplest part of the error correction scheme can also be applied to traditional BFs to achieve some degree of error detection and correction. The exploration of these alternative configurations is left for future work.

References

[1] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[2] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Proc. 40th Annu. Allerton Conf.*, Oct. 2002, pp. 636–646.
[3] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, "Jetty: Filtering snoops for reduced energy consumption in SMP servers," in *Proc. Annu. Int. Conf. High-Perform. Comput. Archit.*, Feb. 2001, pp. 85–96.
[4] C. Fay *et al.*, "Bigtable: A distributed storage system for structured data," *ACM TOCS*, vol. 26, no. 2, pp. 1–4, 2008.
[5] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. 14th Annu. ESA*, 2006, pp. 1–12.
[6] M. Mitzenmacher, "Compressed bloom filters," in *Proc. 12th Annu. ACM Symp. PODC*, 2001, pp. 144–150.
[7] M. Mitzenmacher and G. Varghese, "Biff (Bloom Filter) codes: Fast error correction for large data sets," in *Proc. IEEE ISIT*, Jun. 2012, pp. 1–32.
[8] S. Elham, A. Moshovos, and A. Veneris, "L-CBF: A low-power, fast counting Bloom filter architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 6, pp. 628–638, Jun. 2008.
[9] T. Kocak and I. Kaya, "Low-power bloom filter architecture for deep packet inspection," *IEEE Commun. Lett.*, vol. 10, no. 3, pp. 210–212, Mar. 2006.
[10] S. Dharmapurikar, H. Song, J. Turner, and J. W. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *Proc. ACM/SIGCOMM*, 2005, pp. 181–192.
[11] N. Kanekawa, E. H. Ibe, T. Suga, and Y. Uematsu, *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*. New York, NY, USA: Springer-Verlag, 2010.
[12] D. Bhavsar, "An algorithm for row-column self-repair of RAMs and its implementation in the alpha 21264," in *Proc. Int. Test Conf.*, 1999, pp. 311–318.
[13] M. Nicolaidis, "Design for soft error mitigation," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 405–418, Sep. 2005.
[14] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124–134, 1984.
[15] G. Wang, W. Gong, and R. Kastner, "On the use of bloom filters for defect maps in nanocomputing," in *Proc. IEEE/ACM ICCAD*, Nov. 2006, pp. 743–746.
[16] S. Pontarelli and M. Ottavi, "Error detection and correction in content addressable memories by using bloom filters," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1111–1126, Jun. 2013.
[17] A. Reddy and P. Banarjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, no. 10, pp. 1304–1308, Oct. 1990.
[18] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 5, pp. 651–664, May 2010.
[19] P. Reviriego, J. A. Maestro, S. Baeg, S. J. Wen, and R. Wong, "Protection of memories suffering MCUs through the selection of the optimal interleaving distance," *IEEE Trans. Nucl. Sci.*, vol. 57, no. 4, pp. 2124–2128, Aug. 2010.
[20] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Trans. Rel.*, vol. 39, no. 1, pp. 114–122, Apr. 1990.
[21] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," in *Proc. ACM SIGCOMM*, Sep. 1998, pp. 254–265.
[22] (2012). *CAIDA Anonymized Internet Traces* [Online]. Available: http://www.caida.org/data/passive/passive_2012_dataset.xml