Design an Efficient Error Detection and Correction by using Advanced Bloom Filter

Gandhe Rajendra Prasad [1], G.Niharika [2]

[1] PG Scholar, Department of ECE, Vaagdevi College of Engineering, Bollikunta, Warangal.

[2] Assistant Professor, Department of ECE, Vaagdevi College of Engineering, Bollikunta, Warangal.

Email - rajendraprasad0912@gmail.com, niss.shiloh@gmail.com

Abstract: Here in this paper we are analyzing the Bloom Filter Error Detection and Correction Mechanism with the counting method. A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set or not. 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. The majority based error detecting and correcting codes are used to detect and correct errors in BF memories. 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.

Key Words: Bloom Filter (BFs), Error Correction Soft Errors, Comparator.

1. INTRODUCTION:

Bloom filters (BFs) provide a simple and effective way to check whether an element belongs to a set. They are used in many networking applications as well in computer architectures. The BFs are also used in large databases (e.g., Google Big table use sit to reduce the disk lookups). 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. To optimize the transmission over the network, another extension known as compressed Bloom filters has been presented. Recently Bloom filter (Biff) codes that are based on BFs have been used to perform error correction in large data sets. In most case, BFs are implemented using electronic circuits. 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.

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 Reliable. For BF implementation, memories are a critical element. For memories, permanent errors and defects are commonly corrected using spare rows and columns. 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 large data memory device that continues to operate correctly but has the wrong value in the affected cell. 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. Web cache sharing Collaborating Web caches use Bloom filters (dubbed “cache summaries”) as compact representations for the local set of cached files. Each cache periodically broadcasts its summary to all other members of the distributed cache. Using all summaries received, a cache node has a (partially outdated, partially wrong) global image about the set

of files stored in the aggregated cache. The Squid Web Proxy Cache uses "Cache Digests" based on a similar idea. Query filtering and routing The Secure wide-area Discovery Service, subsystem of Ninja project, organizes service providers in a hierarchy. Bloom filters are used as summaries for the set of services offered by a node. Summaries are sent upwards in the hierarchy and aggregated. A query is a description for a specific service, also represented as a Bloom filter. Thus, when a member node of the hierarchy generates/receives a query, it has enough information at hand to decide where to forward the query: downward, to one of its descendants (if a solution to the query is present in the filter for the corresponding node), or upward, toward its parent (otherwise). Compact representation of a differential file a differential file contains a batch of database records to be updated. For performance reasons the database is updated only periodically (i.e., midnight) or when the differential file grows above a certain threshold. However, in order to preserve integrity, each reference/query to the database has to access the differential file to see if a particular record is scheduled to be updated. To speed-up this process, with little memory and computational overhead, the differential file is represented as a Bloom filter. Free text searching basically, the set of words that appear in a text is succinctly represented using a Bloom filter.

Constructing BF's: Consider a set A= {a1, a2, .an} of *n* elements. Bloom filters describe membership information of *A* using a bit vector *V* of length *m*. For this, *k* hash functions, *h* ,*h* ,..., *hk* 1 2 with *h* : *X* {1..*m*} *i* □ , are used as described below: The following procedure builds an *m* bits Bloom filter, corresponding to a set A and using *k h* ,*h* ,..., *h* 1 2 hash functions:

Procedure Bloom Filter(set A, hash_functions, integer m) returns filter filter = allocate *m* bits initialized to 0 foreach *ai* in *A*: for each hash function *hj*: filter[*hj(ai)*] = 1 end for each end for each return filter Therefore, if *ai* is member of a set *A,* in the resulting Bloom filter *V* all bits obtained corresponding to the hashed values of *ai* are set to 1. Testing for membership of an element *elm* is equivalent to testing that all corresponding bits of *V* are set:

Procedure Membership Test (elm, filter, hash_functions) returns yes/no for each hash function *hj*: if filter [*hj(elm)*] != 1 return No end for each return Yes *features*: filters can be built incrementally: as new elements are added to a set the corresponding positions are computed through the hash functions and bits are set in the filter. Moreover, the filter expressing the reunion of two sets is simply computed as the bit-wise OR applied over the two corresponding Bloom filters.

Soft Error: Soft error is an error occurrence in a computer's memory system that changes an instruction in a program or a data value. Soft errors typically can be remedied by cold booting the computer. A soft error will not damage a system's hardware; the only damage is to the data that is being processed. There are two types of soft errors: chip-level soft error: These errors occur when the radioactive atoms in the chip's material decay and release alpha particles into the chip. Because an alpha particle contains a positive charge and kinetic energy, the particle can hit a memory cell and cause the cell to change state to a different value. The atomic reaction is so tiny that it does not damage the actual structure of the chip. Chip-level errors are rare because modern memory is so stable that it would take a typical computer with a large memory capacity at least 10 years before the radioactive elements of the chip's materials begin to decay. system-level soft error: These errors occur when the data being processed is hit with a noise phenomenon, typically when the data is on a data bus .The computer tries to interpret the noise as a data bit, which can cause errors in addressing or processing program code. The bad data bit can even be saved in memory and cause problems at a later time. Several techniques have been employed in order to reduce those soft errors and Multiple Bit Upsets.

2. OVERVIEW OF BLOOM FILTER:

2.1. Function of Bloom Filter in Low-Power Deep Packet Inspection

A Bloom filters architecture that exploits the well-known pipelining technique. Bloom filters are frequently used to identify malicious content like viruses in high speed networks. However, the architectures are used to implement Bloom filters are not power efficient. A new Bloom filter

architecture that exploits the well-known pipelining technique. Through power analysis we show that pipelining can reduce the power consumption of Bloom filters which leads to the energy efficient implementation of intrusion detection systems.

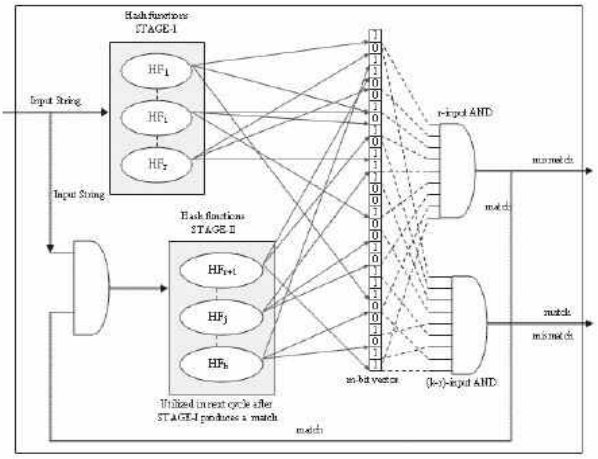

Fig.1 Pipelined Bloom Filter

2.2. Operation of Error Detection and Correction in Content Addressable Memories

A Content Addressable Memory (CAM) is an SRAM based memory which can be accessed in parallel in order to search for a given search word, providing as result the address of the matching data. The use of CAM is widespread in many applications ranging from the controller of a CPU memory cache to the implementation of lookup tables of high speed routers. Like conventional memories, CAM can be affected by the occurrence of Single Event Upsets (SEU) which can alter its operation causing different effects such as pseudo HIT or pseudo-MISS events. In order to avoid the effects of SEUs different approaches have been proposed in previous literature, but all of these solutions require changes to the internal structure of the CAM itself. Differently from previous approaches, in this paper we propose a method that does not require any modification to a CAM's internal structure and therefore can be easily applied at system level, using a suitable redundant CAM component in order to obtain a CAM module with error detection and correction capabilities.

2.3. Bloom Filter Based Associative Deletion

This is not entirely suitable for many new applications, such as deleting one attribute value according to another attribute value for a set of data objects/items with two correlated attributes. The concept for such an operation, called the associative deletion. To realize this operation, used a new Bloom filter data structure, named IABF (Improved Associative deletion Bloom Filter), the association information on the two correlated attributes of items in the given data set. Based on IABF, used an algorithm to perform associative deletions, which can be applied to both normal data and streaming data. The two-attribute scheme in a pairwise manner or by an extended version of IABF, these two methods may not provide the best performance. To further accelerate the operation, also illustrate a hardware coprocessor implementation for a crucial component of the algorithm. Detailed theoretical analysis and experimental results demonstrate that the presented IABF technique can accurately process associative deletions with controlled false positive and negatives. The two-attribute scheme in a pair-wise manner or by an extended version of IABF, these two methods may not provide the best performance.

2.4. Bloom filter in networking field

In Collaborating in overlay and peer-to-peer networks, Bloom filters can be uses for summarizing content to aid collaborations in overlay and peer-to-peer networks. In Resource routing, the Bloom filters allow probabilistic algorithms for locating resources .In Packet routing, Bloom filters provide a means to speed up or simplify packet routing protocols. The Measurement of Bloom filter that is provide a useful tool for measurement infrastructures used to create data summaries in routers or other network device.

3. PROPOSED METHOD:

3.1. 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, *xe* is read from the memory. The correct value *x* has to be *xe* if the error affected the parity bit. If the error affected the *i*th data bit, the correct value will be *x*em*(i)* where xem*(i)* is the value read *(*xe*)* with the *i* th bit inverted. To determine which of those is in fact the correct value *x*, the candidates [*xe* and all the *x*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 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

*Pco*rrection ~ = (1 − *p f p*) ^l

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

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 [*xe* and all the *xem(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, . . . , *h k* can be approximated as 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.

3.3. Bloom Filter Used with Comparator by Parallel Prefix

In order to improve the comparison time of bloom filter, we use parallel prefix comparator to compare incoming data. The construction and the working of our proposed comparator explain given below: In this section, the comparator's design is elaborated which is based on using a novel parallel prefix tree .Each set or group of cells produces outputs that serve as inputs to the next set in the hierarchy, with the exception of set 1, whose outputs serve as inputs to several sets.

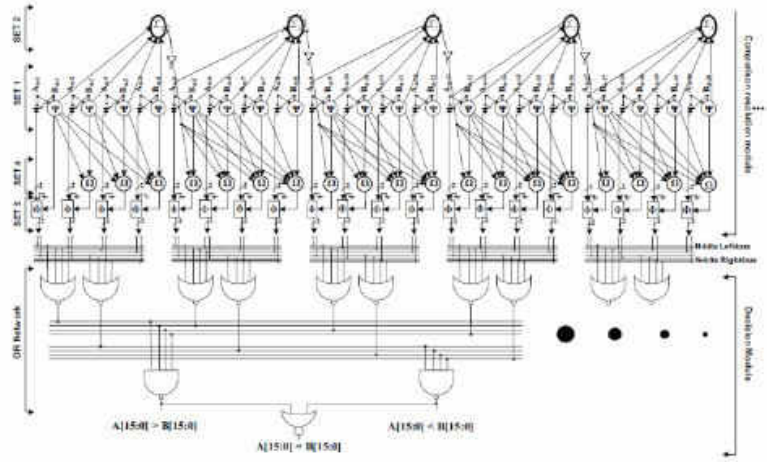

Fig.2 Details of Implementation Architecture

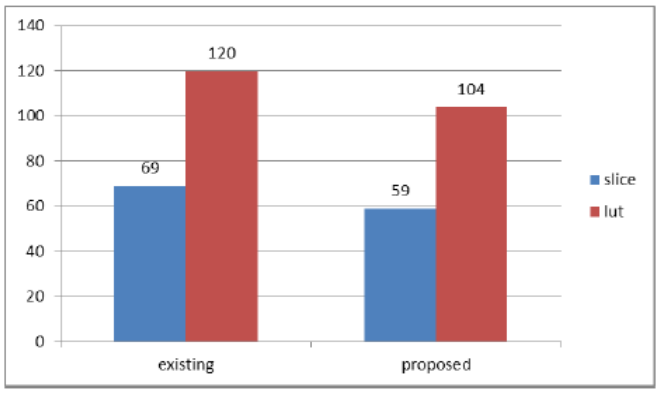

Fig.3 Performance of Existing and Proposed Analysis

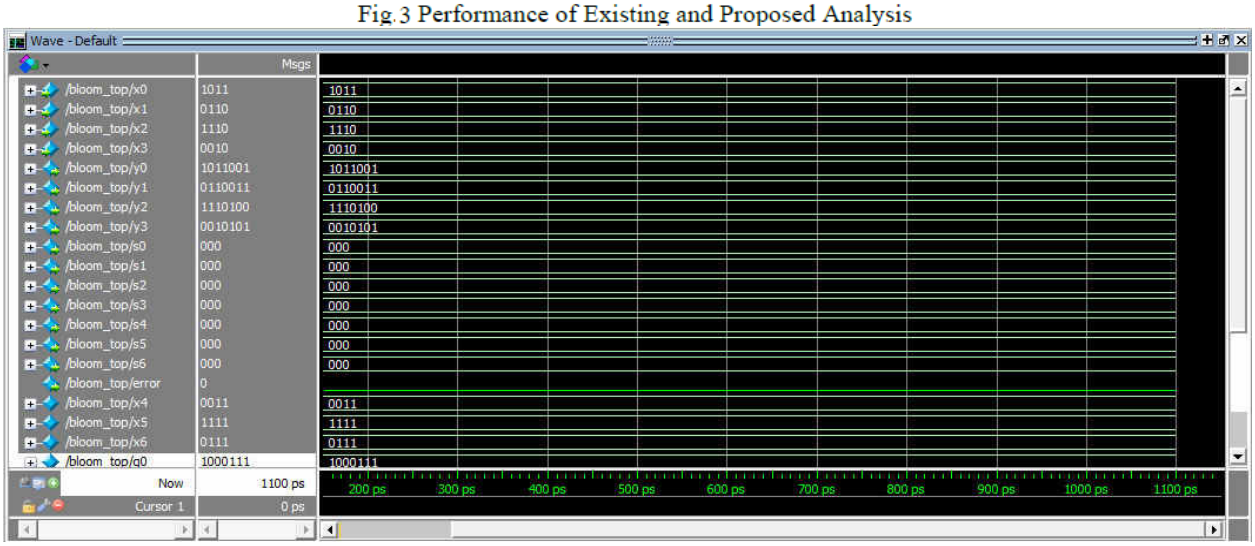

Fig.4 Simulation result for Bloom Filter

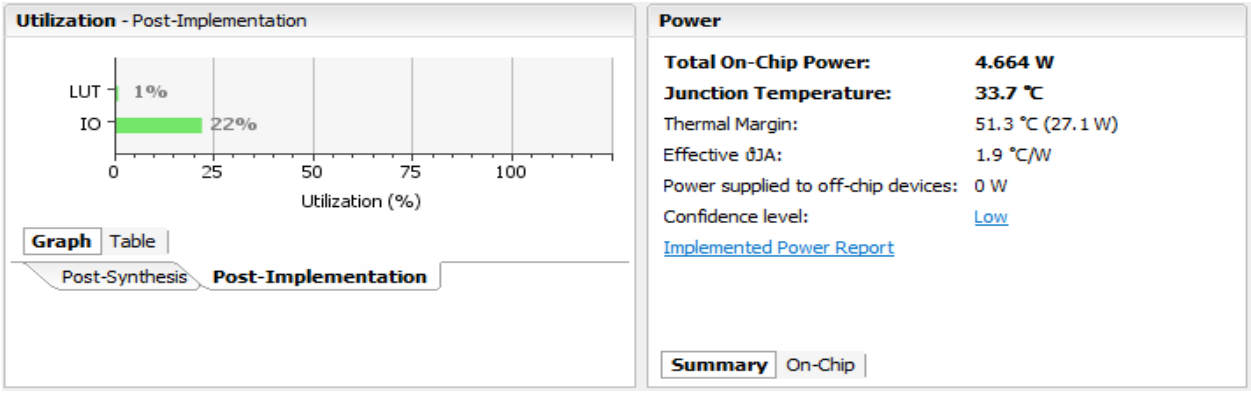

Fig. 5 Synthesis report for Bloom Filter

CONCLUSION:

In this brief, a new application of BFs has been proposed. The idea is to use high-speed low-power comparator in BFs to compare element set. In particular comparator structured as parallel prefix trees with repeated cells in the form of simple stages that are one gate level deep with a maximum fan-in of five and fan out of four, independent of the input bit width. Simulation results show our proposed bloom filter has improved performance in terms of both comparison time and memory protection. 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 show 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. J.Qian, Q. Zhu, and Y. Wang,: "Bloom Filter Based Associative Deletion", IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 8, pp 1986-1998, August 2014.
2. Y. Jin, Y.Wen, and W.Zhang, "Content Routing and Lookup Schemes using Global Bloom Filter for Content-Delivery-as-a-Service", IEEE Systems Journal, Vol. 8, No. 1, pp 268- 278, March 2014.
3. 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.
4. F. Hao, M. Kodialam, T.V. Lakshman, and H. Song,: "Fast Dynamic Multiple-Set Membership Testing Using Combinatorial Bloom Filters", IEEE/ACM Transactions On Networking, Vol. 20, No. 1, pp. 295-308, February 2012.
5. Z. Zhong, and K. Li, "Speed Up Statistical Spam Filter by Approximation", IEEE Transactions on Computers, Vol. 60, No. 1, pp. 120-134, January 2011.
6. D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," IEEE Transactions On Knowledge and Data Engineering, vol. 22, no. 5, pp. 651–664, May 2010.
7. D. Ficara, A. Di Pietro, S. Giordano, G. Procissi, and F. Vitucci,: "Enhancing Counting Bloom Filters Through Huffman-Coded Multilayer Structures", IEEE/ACM Transactions On Networking, Vol. 18, No. 6, pp. 1977-1987, December 2010.
8. S. Elham, A. Moshovos, and A.Veneris,: "L-CBF: A low-power, fast counting Bloom filter architecture," IEEE Transactions On Very Large Scale Integration (VLSI) Systems, vol. 16, no. 6, pp. 628–638, Jun. 2008
9. B. Bloom,: "Space/time tradeoffs in hash coding with allowable errors," Commun. ACM, vol. 13, no. 7, pp. 422–426, 1970.
10. Broder and M. Mitzenmacher,: "Network applications of bloom filters: A survey," in Proc. 40th Annu. Allerton Conf., Oct. 2002, pp. 636–646.
11. 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.
12. Fay et al.,: "Bigtable: A distributed storage system for structured data," ACM TOCS, vol. 26, no. 2, pp. 1–4, 2008.
13. 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.
14. M. Mitzenmacher,: "Compressed bloom filters," in Proc. 12th Annu. ACM Symp. PODC, 2001, pp. 144–150.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. M. Nicolaidis,: "Design for soft error mitigation," IEEE Trans. Device Mater. Rel., vol. 5, no. 3, pp. 405–418, Sep. 2005.
22. 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.
23. 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.
24. 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.
25. Reddy and P. Banarjee, "Algorithm-based fault detection for signal processing applications," IEEE Trans. Comput., vol. 39, no. 10, pp. 1304–1308, Oct. 1990.
26. 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.
27. 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.
28. 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.
29. 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.
30. CAIDA Anonymized Internet Traces [Online]. Available: http://www.caida.org/data/passive/passive_2012_dataset.xml