Adapting the Bloom Filter to Multithreaded Environments

Ivan Voras and Mario Žagar [*]

***Abstract*—Many classical algorithms like the Bloom filter were envisioned and created at a time when computers were the size of rooms and multithreading was not yet even explored theoretically. The landscape of modern mainstream computer systems today is heavily dominated with multi-core CPU-s but the effort to make full use of such systems is still ongoing. The Bloom filter has proven itself useful both as a core algorithm and as a supportive or optimizing addition to other data classification algorithms. This work explores adaptations to the basic Bloom filter algorithm for use in multithreaded applications on contemporary SMP systems and the consequences of such adaptations to its overall efficiency.**

***Index Terms*—multithreading, multiprocessing, Bloom filter, data structure, algorithms**

I. Introduction

THE Bloom filter was first described by Burton Howard Bloom in 1970 [1] and has since then become a standard and widely used algorithm for quick set membership testing with large sets and/or low memory conditions. Though it was originally conceived for use in databases as a helper algorithm to reduce disk access frequency, since its introduction it has been successfully used in various network-related tasks. Some applications of the Bloom filter include network packet inspection and classification, forwarding, peer-to-peer indexing and distributed web caching [2][3][4][5][7][8]. Aside from these acknowledged technical applications it is also used in certain surprisingly diverse environments such as computer games [6]. The central useful characteristic of the Bloom filter is the ability to test memberships in large sets in what is practically constant time and memory usage, but with a (preferably low but always present) probability of false positives.

The Bloom filter is an ingenious construct built around a central bitmap or an integer array which is updated and tested depending on the result of hashing functions applied to the stored or tested strings of bytes. This structure is a point of contention in multithreading applications and in this paper we explore several types of modifications to the basic algorithm that enable various degrees of simultaneous multithreaded access to the Bloom filter and analyse their performance and efficiency.

The paper is organized as follows. Section II describes the original Bloom filter and the specific implementation analysed in this work, with references to related works on the topic. Section III presents the analysis of the central problem and the theoretical expectations with regards to efficiency of adapting the algorithm to a multithreaded environment. Section IV presents our test implementations, their performance and efficiency characteristics. Finally, we present conclusions of this work in section V.

II. The Bloom Filter

The traditional Bloom filter is described (for example in [8]) as an *m*-bit array used with *k* hash functions which compress the input string into *k* numbers addressing single bits in the array. To add a string to the set, the string is processed with the hash functions and *k* bits in the array are set to one. To check if the string is in the set, *k* bits whose addresses are obtained by processing the string are checked to be all set to one. False positive tests of set membership can occur because of the hash functions themselves or because of hash array "saturation" by bits set to one from adding multiple elements to the set. False negatives, on the other hand, can never appear. Figure 1 shows a simple example of the bit array.

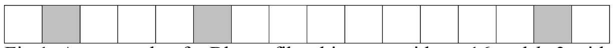
Fig 1. An example of a Bloom filter bit array with *m*=16 and *k*=3 with one element added to the set. Grey squares represent bits set to one.

It is visible from the description of the central algorithm that the quality of the filter depends on many factors: on the number of hash functions *k*, the size *m* of the bit array, the number of elements already added to the set *n* and the quality of the hash functions.

The probability *p* of a complete *k*-bit collision in an *m*-bit filter with good (unbiased) hash functions is approximated [7] by (1), from which the major trends can easily be seen: optimizing for low collision rate requires a large bit array or a small number of entries stored in the set, but the optimal number of hash functions requires a more complex solution whose outcome is given in (2).

$$p=(1-e^{\frac{-kn}{m}})^k \quad (1)$$

$$k=\frac{m}{n}\cdot\ln 2 \quad (2)$$

In practice, slightly suboptimal values for *k* and *m* might be selected to take advantage of external environment factors, for example choosing appropriate powers of 2 for *k* and *m* to optimize calculations and/or memory usage.

[*] Ivan Voras and Mario Žagar are with University of Zagreb Faculty of electrical engineering and computing, Unska 3, Zagreb, Croatia (e-mail: {ivan.voras, mario.zagar}@fer.hr)

This work is supported in part by the Croatian Ministry of Science, Education and Sports, under the research project "Software Engineering in Ubiquitous Computing".

A useful variation of the original construct is the "Counting Bloom filter" [9], in which the bit array is replaced with an array of integers. In this variation, additions to the set are performed by incrementing the *k* integers, which leads to the ability to remove elements from the set by decrementing them.

As others have implicitly noticed (e.g. [6], [8]), the array (in any form) of the Bloom filter becomes itself a sort of a digest (or hash) of the entire set, which can at the same time directly be used to support common set operations such as set union (by "adding" array elements), set intersection and set complement (by comparing and "decrementing" array elements in various ways), of course while still being subject to the probabilities of false positives.

A. Modern uses

The Bloom filter was a somewhat obscure data structure since its conception but has found very successful application in modern data-intensive network-related problems. A search on the IEEE Xplore digital library reveals only a single paper containing the phrase "bloom filter" in its title before year 2000, and 101 papers with titles containing the phrase published between years 2002 and 2009 (inclusive), with nearly 50% of those in years 2008 and 2009. Of course, the structure has found its use in the problem originally described by Howard Bloom and has presumably been quietly used ever since in spell check dictionaries and some databases but it has never been so extensively studied and experimented upon until recent times.

What makes the Bloom filter so successful now is both the promise of O(1) performance with respect to time and memory accesses in its common operations with large sets and the ability to use the resulting Bloom array as a digest for large sets. The keyword here is "large" - both of these applications are centred around the remarkable compression with which the Bloom filter can represent volumes of data while still having useful internal structure (as opposed to simple hash functions).

We observe that these two distinct advantages have been explored mostly in unrelated areas of application and study. The fast and cheap set lookup and update operations have mostly been explored in the context of data classification such as in active network equipment (e.g. [4][5]), while the digest aspect of the filter has proven itself a powerful tool in maintaining data set coverage and accessibility in distributed and peer-to-peer systems [8][10]. A typical example of use in data classification would be in routers or firewalls, where it is convenient to test every packet for membership to a particular flow or a network connection before proceeding with more complex analysis, and the relatively small memory requirements of the Bloom filter make it easy to implement in both high-performance and highly constrained hardware [11].

In its role as a set digest, an implementation of the Bloom filter has become a part of an Internet standard protocol, the *Internet Cache Protocol* (ICP) used by web proxies to discover which sets of documents are cached by their neighbours [8].

B. Working around false positives

While the Bloom filter can be used by itself, the always-present probability of false positives makes it inconvenient to use in most purposes. Specifically, the random nature of hash functions make such collisions unpredictable in the same way as in hash tables.

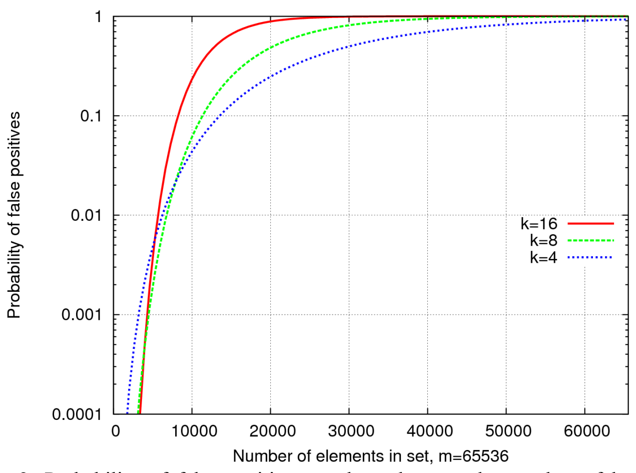

Fig 2. Probability of false positives as dependant on the number of hash functions *k* and the number of elements in the set *n*. Because of the essentially exponential characteristic, the filter can very quickly become unusable if more than the optimum number of elements are added to the set.

As depicted in Fig. 2, the danger of suddenly saturating the filter with set elements to the point where the probability of false positives cannot be ignored is very real in practice. When the number of elements in the array is kept the same, using more hash functions (which translates to a more sophisticated set digest) simply increases the filter saturation. Optimal Bloom filter parameters can be found for specific applications by solving (1) or (2) for *k*, *m*, or *n*, depending on practical considerations.

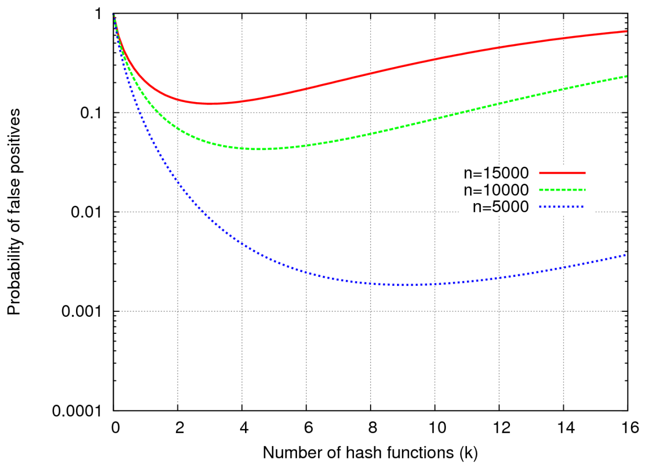

Fig 3. The optimum number of hash functions as dependant on the number of elements added to the set (n) when the filter array size (m) is fixed to 65536.

If the number of filter entries is fixed, the optimum number of filter array entries *k* depends on the predicted number of elements in the set, as shown in Fig. 2.

To work around the unavoidable false positives, the main structure of the Bloom filter is commonly backed by a regular hash table or a similar, deterministic structure. In some cases [5] the Bloom filter structure is integrated with other structures to take better advantage of its strong sides.

C. Our test implementation

To perform experiments with the Bloom filter algorithm we have created a test implementation which is flexible in the parameters important for this study.

The test implementation used in this work is in essence a canonical Counting Bloom filter having the following properties:

- the number of filter array entries *m*=65536
- the number of hash functions *k*=4
- each array element is a 16-bit integer.

The rationale for these choices is as follows. Though we could have chosen any data set for our experiments, including one synthetically created out of random byte strings, we have tried to mimic a realistic application by using lines from Odyssey by Homer[1], of which our variant has 15439 lines, a value which can be stored in a 16-bit counter in the (extremely unlikely) case all lines produce collisions in one or more of the same filter array elements. From results presented in Fig. 1 and 2 we have selected to use *k*=3 hash functions. We have chosen MD4 [12] for the basis of our hash functions[2] because it is conveniently available in our environment and undoubtedly a good hash function for this purpose. The MD4 function produces a 128-bit result, which is too long by itself but which can be divided into 16-bit values each of which can address one of the *m* elements in the filter array. We therefore define our three hash functions $f_i(s)$, where i=1..3 as the *i*-th consecutive 16-bit slice of the result of the function MD4(s).

Our implementation is created in the C programming language and a POSIX-compliant environment, using the POSIX Threads library for multithreading and locking operations, and OpenSSL's implementation of the MD4 hash. Our primary test hardware platform is Intel Xeon 5405 with two 4-core processors (i.e. 8 logical CPUs in total) running the 64-bit edition of FreeBSD 8.0. This is the platform on which all our published results are obtained. Our secondary platform, used for verification of our methods is AMD Opteron 2216HE with two 2-core processors running 32-bit Linux 2.6.28.

D. Previous work

A very high performance application of the Bloom filter is in network packet classification and from this field comes a strong interest in its implementations in multi-core ASIC hardware [15], [16], [17] concentrated around designing hardware matchers that would search a stream of packets.

Our approach differs from the referenced ones in exploring the software side of the implementation – adapting software to run on widely deployed general-purpose architectures as-is, within constraints of existing hardware. Our previous work in this area includes designing concurrent data structures for high-performance memory databases and software cache middleware [20].

III. The Bloom Filter in Multithreaded Applications

In the ideal case, the Bloom filter is a very scalable structure in multiprocessing if the number of hash functions *k* is low enough with regards to the number of possible elements *m* (which it always is for any sensible application) and if the number of threads working in parallel on the Bloom filter is also low enough (which it currently always is, because technical considerations limit the number of CPU-s in today's computers). An interesting situation arises when the filter is to be updated as a regular part of the application's operation. In this case the problem is completely the same whether the filter is implemented as an array of bits (the original Bloom filter) or an array of integers (the Counting Bloom filter) since general-purpose CPU-s operate on word-size memory units (i.e. there is no way to atomically update a single bit of memory, but there are ways to atomically update words, possibly of various sizes). This leads to the classical race condition problem of read-modify-update by multiple threads. In the case of the Counting Bloom filter the error may manifest in erroneous counters, subsequently leading to errors if elements are removed, while in the original Bloom filters some bits may not be set, leading more directly to the extremely unwanted property of the filter having *false negatives*.

To ensure consistency and data access ordering, all *k* values need to be updated atomically. This is not possible with atomic CPU operations (the case of which we include in a later section), but requires stronger primitives.

We have selected to use POSIX read/write locks (*rwlocks*, also known as shared/exclusive locks) with the property of allowing lock acquisition in either "read" (or shared) mode by multiple threads at the same time or "write" (or exclusive) mode by only a single thread.

A. On scalability

In the worst case of hash function collisions, all active threads try to exclusively lock the same write locks for *k* elements out of *m*. Since the ratio *k/m* is very small, Amdahl's law (3) predicts that the problem as stated is almost embarrassingly parallel (for *k*=3 and *m*=65536, the *k/m* ratio is ≈ .000046).

$$P=\frac{k}{m},\ s=\frac{1}{(1-P)+\frac{P}{N}} \tag{3}$$

Of course, there is still the presence of "hidden variables" in external factors such as the hardware implementation, but these are outside our influence for the experiments in this work.

[1]Obtained from Project Gutenberg, translation by William Cowper, available at http://www.gutenberg.org/ebooks/24269

[2]The MD4 hash function has been considered unsafe for use in cryptography-related purposes for some time now[13] but we are using it not as a security or cryptography primitive but as a good general purpose hash function better than more simple alternatives [14]. We selected it over MD5 because we observed that on our test hardware MD4 can be up to 60% faster than MD5.

IV. Efficiency of Adaptations of the Bloom Filter to Multithreaded Applications

Because of its simple structure, the Bloom filter is very suited for read-only access – tests for set memberships require only simple memory accesses to the filter array. In our implementation the filter array size is 128 KiB, which very easily fits in modern CPUs' L2 caches, and a quarter [18] or half [19] of it can even fit in the L1 caches. This makes it suitably efficient without special effort.

We have found that our test platform's implementation of POSIX rwlocks associates a 40-byte structure with each rwlock present in the application, which can in the literal interpretation lead to the uncomfortable situation of having 40-byte locks locking 2-byte values. To explore the impact of this size ratio and the possibility of working around it, we have added the option to lock regular regions of array elements instead of individual elements to our test implementation, reducing the required number of rwlocks from *m* by an almost arbitrary factor.

To perform a set addition, all needed locks are first acquired in write mode, then the counters are updated and the locks released. To avoid deadlocks, if any of the locks cannot be acquired, all previously acquired locks are rolled back (released), the CPU is relinquished (*yielded*) for the locking thread and the process repeats until all locks can be taken. Locking for set membership tests (i.e. read-only) proceeds in exactly the same way, but using read locks[3]. In addition to the described locking discipline, one of our concerns that prompted this work was that some of the benefits that come from the simple and compact structure can be detrimental to performance in complex multiprocessor environments. We were concerned if the low-level hardware structures such as L1/L2 caches had a visible effects on the application level.

A. Locking the Bloom filter with read/write locks

Our test implementation is parametrized in a way that allows us to vary the data set, the number of threads working on the filter and the number of filter array elements protected by a single rwlock. Our experiments resulted in two indicators: the number of performed operations per second and the number of contentions during the lock acquisition phase which caused locks to be rolled back. Figure 4 shows these results as influenced by varying the number of threads and the number of entries protected by single rwlocks.

The results of this experiment are quite interesting in their uniformity. Our estimation that low-level hardware structure such as the size of CPU cache lines will have some global effect on the performance curves was proven wrong, at least for the circumstances in question. Though the *k/m* ratio is miniscule, the sheer number of operations performed results in a surprisingly large number of contentions between threads, resulting in retries in locking filter array elements. Up to 10^6 contentions per second could be observed in the worst case (which was when the number of active threads equalled the number of CPUs available).

[3]The problem of "writer starvation" caused by many read acquisitions on a lock preventing a write acquisition is in our case solved by the operating system's implementation of the POSIX rwlocks.

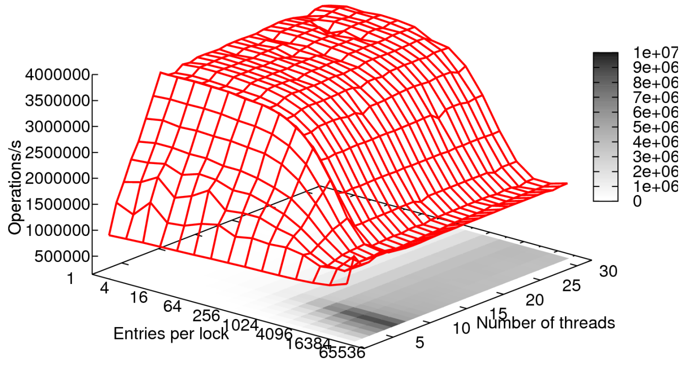

Fig 4. Operations per second performing a random mixture of 95% test and 5% add operations on a Bloom filter, with read/write locking per region of multiple entries and with the rising number of threads. The bottom plane contains a "hot spot map" of retries made to acquire all *k* locks in 10 seconds.

A very positive result of the test was that moving from a single lock per filter array entry to nearly three order of magnitude less locks still results in adequate performance. The performance difference when using eight threads between the case of having each element protected by its own lock and one lock protecting 128 elements is less than 8%, while protecting 1024 elements by a single lock results in nearly 50% less performed operations per second.

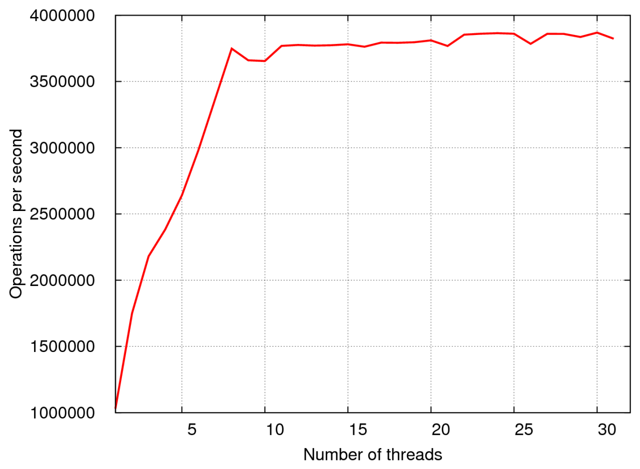

Fig 5. A slice of Fig. 4 for 8 items per rwlock, showing increase in performance depending on the number of active threads.

A closer look at a slice of the results in Fig. 5 shows that, though roughly following the expected linear increase of performance as the number of threads increases to be equal to the number of available CPUs, the speedup is not directly proportional. Specifically, speedup achieved when the number of threads increases from one to two is approximately 70%, it is ≈210% for three threads and ≈360% for eight threads. This is far away from embarrassingly parallel performance expected. Fitting the speedups achieved between one and eight threads to (3) results in an approximate P factor (the "unserializable" part of the operations) for this application of around 0.8.

B.Accessing the Bloom filter with atomic operations

Complex locks such as read/write locks are on the low level created by using primitives available directly from the CPU's instruction set.

These instructions are of course unportable across system architectures and the modifications we made to the test implementations to use them thus make this aspect of the implementation unportable.

Specifically, we have taken advantage of the i386 (and by extension the AMD64) architecture on which LOCKed operations on integer are synchronized in hardware across other logical CPU-s in the system. Though the architecture supports several instructions in this way, we have used only LOCK ADD on a memory operand. Because of its legacy, the architecture natively supports such instructions even on data sizes that are not its native (or maximum) word size; in our case we applied it on 16-bit integers.

Other CPU architectures may or may not support such low-level constructs.

This modification of the test application cannot support data ordering involving multiple accesses, meaning that operations on all the *k* elements of the filter array may be interleaved with operations from other threads / CPU-s (there is no consistent "snapshot" state for all *k* elements at a single point in time). This will manifest itself only in slightly larger probabilities of false positives in the long run, though false negatives still cannot happen.

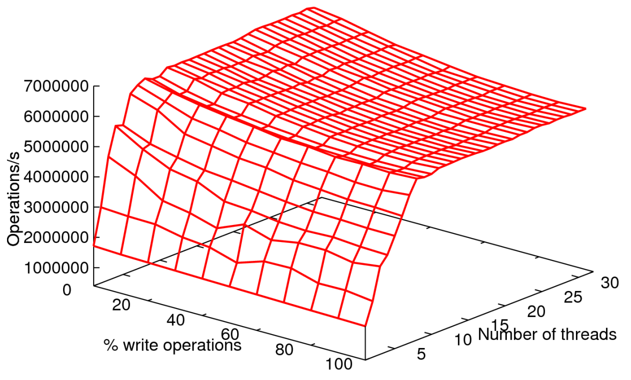

Fig 6. Operations per second with a Bloom filter using atomic memory operations, with increasing number of threads and varying percentage of test operations and add operations.

Results in Fig. 6 reveal significantly higher absolute performance than those presented in Fig. 4, but surprisingly the efficiency in scaling is nearly the same – approximately 380% from one to eight threads.

These experiments were repeated on our secondary hardware platform which, while having different absolute results, exhibited similarly low efficiency, ≈230% for the case using rwlocks between one and four threads and only ≈210% better results in the test using atomic integer operations between one and four threads, a significant difference to the primary platform.

The reasons for this deviation from the expected scalability need to be searched for in factors external to the abstract algorithm – in the hardware on which it runs. While the exact details require further detail study, they are likely to be located in the areas where multiple CPUs access shared resources, like the system and the memory buses.

V.Conclusion

We have studied the Bloom filter data structures, and specifically some modifications to the basic structure and its algorithms to make it usable in multithreaded applications. To make this study possible, we have created a test implementation that allows us to investigate how do many of the recognized parameters influence the performance of the Bloom filter when used in a multithreaded environment.

We have compared two major variations of the original Counting Bloom filter structure, one using POSIX read/write locks to ensure consistent access to the filter array for all hash functions and one using platform-specific atomic CPU instructions without guarantees of consistency during all modifications of the filter array (though individual modifications of the array are of course consistent). Our implementation was tested on two common hardware platforms (AMD64 and i386 server-class hardware from Intel and AMD).

We have shown that when consistent access to the Bloom filter array is required, using read/write locks to protect array elements, it is enough to use up to two orders of magnitude less lock objects than the number of protected elements to achieve good enough performance on our test platforms.

Our analisys of the efficiency of the implementation shows that though the algorithm theoretically has a low ratio of sequential to parallel operations, external factors have a large influence on the scalability of a straightforward implementation.

References

[1] B. H. Bloom. "Space /Time Trade-offs in Hash Coding with Allowable Errors". *Communication of the ACM*, vol. 13, pp. 422–426, July 1970.

[2] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In *Proc. 14'th Annual Allerton Conf. on Communication, Control, and Computing*, pages 636–646, October 2002.

[3] F. Chang, F. Wu-chang, and L. Kang. "Approximate Caches for Packet Classification". In *23'th Annual Conf. of the IEEE, INFOCOM*, pages 2196–2207, March 2004.

[4] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. "Fast Packet Classification Using Bloom Filters". Technical Report 27, Department of Computer Science And Engineering, Washington University in St. Louis, May 2006.

[5] H. Song, J. Turner, S. Dharmapurikar, and J. Lockwood. "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing". In *Proc. of Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 181–192, August 2005.

[6] M. Fischer, "Coding Bloom Filters" [Online] Available: http://www.flipcode.com/archives/Coding_Bloom_Filters.shtml, [Retrieved Nov 1st, 2009].

[7] A. Broder, M. Mitzenmacher. "Network applications of bloom filters: A survey", *Internet Mathematics*, vol. 1, 2004, pp. 485-509.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", *IEEE/ACM Transactions on Networking* vol. 8, pp. 281-293, 2000.

[9] M. Mitzenmacher. "Compressed Bloom Filters". *IEEE/ACM Transactions on Networking* vol. 10, pp- 604-612, 2002.

[10] P. Hebden, A.R. Pearce, "Bloom filters for data aggregation and discovery: a hierarchical clustering approach", *Proceedings of the 2005 International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pp. 175-180, 2005.

[11] M. J. Lyons and D. Brooks, "The design of a bloom filter hardware accelerator for ultra low power systems". *Proceedings of the 14th ACM/IEEE international Symposium on Low Power Electronics and Design* , pp. 371-376, 2009.

[12] R. Rivest, "RFC 1320: The MD4 Message-Digest algorithm", IETF Network Working Group Request for Comments Document. [Online] Available: http://tools.ietf.org/html/rfc1320 [Retrieved Nov 11th 2009]

[13] B. d. Boer, A. Bosselaers, "An Attack on the Last Two Rounds of MD4". *Proceedings of the 11th Annual international Cryptology Conference on Advances in Cryptology*, 1991.

[14] J. Stone, C. Partridge, "When the CRC and TCP checksum disagree". *SIGCOMM Comput. Commun.* Rev. 30, 4, pp. 309-319, 2000.

[15] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. Lockwood, "Deep packet inspection using parallel Bloom filters," *Proceedings of the 11th Symposium on High Performance Interconnects*, pp. 44-51, Aug. 2003.

[16] Haoyu Song, Fang Hao, M. Kodialam, T.V. Lakshman, "IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards". *Proceedings of IEEE INFOCOM 2009*, pp. 2518-2526, April 2009

[17] K. Nikas, M. Horsnell, J. Garside, "An adaptive bloom filter cache partitioning scheme for multicore architectures". *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation 2008*, pp. 25-32, July 2008.

[18] Intel Corp. "Intel Xeon ® Processor 50'00 Sequence", Intel Technical Documents [Online] Available: http://www.intel.com/p/en_US/products/server/processor/xeon5000/technical-documents [Retrieved Nov 12th 2009]

[19] Advanced Micro Devices Corp. "AMD Opteron ™ Processor Product Data Sheet", AMD White Papers, [Online] Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf [Retrieved Nov 12th 2009]

[20] I. Voras, D. Basch, M. Zagar, "A high performance memory database for web application caches", *Proceedings of the 14th IEEE MELECON Conference*, pp.163-168., May 2008.

VI. Biographies

I. Voras (M'06), was born in Slavonski Brod, Croatia. He received Dipl.ing. in Computer Engineering (2006) from the Faculty of Electrical Engineering and Computing (FER) at the University of Zagreb, Croatia. Since 2006 he has been employed by the Faculty as an Internet Services Architect and is a graduate student (PhD) at the same Faculty, where he has participated in research projects at the Department of Control and Computer Engineering. His current research interests are in the fields of distributed systems and network communications, with a special interest in performance optimizations. He is an active member of several Open source projects and is a regular contributor to the FreeBSD operating system. Contact e-mail address: ivan.voras@fer.hr.

M. Žagar (M'93-SM'04), professor of computing at the University of Zagreb, Croatia, received Dipl.ing., M.Sc.CS and Ph.D.CS degrees, all from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER) in 1975, 1978, 1985 respectively. In 1977 M. Žagar joined FER and since then has been involved in different scientific projects and educational activities.

He received British Council fellowship (UMIST - Manchester, 1983) and Fulbright fellowship (UCSB - Santa Barbara, 1983/84). His current professional interests include: computer architectures, design automation, real-time microcomputers, distributed measurements/control, ubiquitous/ pervasive computing, open computing (JavaWorld, XML,..).

M. Žagar is author/co-author of 5 books and about 100 scientific/ professional journal and conference papers. He is senior member in Croatian Academy of Engineering. In 2006 he received "Best educator" award from the IEEE/CS Croatia Section.. Contact e-mail address: mario.zagar@fer.hr.