Interval Filter: a locality-aware alternative to Bloom filters for hardware membership queries by interval classification

R. Quislant, E. Gutierrez, O. Plata and E.L. Zapata

Department of Computer Architecture, University of Málaga,
ETSI Informática, Campus Teatinos, Málaga, E 29071, Spain
{quislant,eladio,oplata,zapata}@uma.es

Abstract. Bloom filters are data structures that can efficiently represent a set of elements providing operations of insertion and membership testing. Nevertheless, these filters may yield false positive results when testing for elements that have not been previously inserted. In general, higher false positive rates are expected for sets with larger cardinality with constant filter size. This paper shows that for sets where a distance metric can be defined, reducing the false positive rate is possible if elements to be inserted exhibit locality according to this metric. In this way, a hardware alternative to Bloom filters able to extract spatial locality features is proposed and analyzed.

1 Introduction

Bloom filters [1] were devised to test whether an element is a member of a set in a time and space-efficient way. They allow insertions of an unbounded number of elements at the cost of false positives, but not false negatives (elements can be added to the set, but not removed). A Bloom filter comprises a bit array and k different hash functions that map elements into k randomly distributed bits of the array. At first, all the array bits are set to 0. Inserting an element into the Bloom filter consists in setting to 1 the k bits given by the hash functions. Test for membership consists in checking that those k bits are asserted.

Bloom filters has been used in a wide range of applications in the domain of networks [2], file searching [6], and more recently in the domain of an emerging and promising field, Transactional Memory (TM) [5]. In this case, elements inserted into the Bloom filter correspond to memory address locations issued by a running program.

TM arises as an alternative to the conventional multithreaded programming to ease the writing of concurrent programs in multicore processors. TM introduces the concept of transaction that allows semantics to be separated from implementation. A transaction is a block of computations that appears to be executed with atomicity and isolation. Thus, transactions replace a pessimistic lock-based model by an optimistic one and solve the abstraction and composition problems.

Hardware Transactional Memory (HTM) implements most of the required mechanisms of TM at the core level. HTM systems execute transactions in parallel, committing non-conflicting ones. A conflict occurs when a memory location is concurrently accessed by several transactions and at least one access is a write. Thus, HTM systems must record all memory reads and writes during the execution of transactions in order to detect conflicts. Bloom filter signatures have been recently proposed to store the addresses of such memory reads and writes [9]. However, these signatures may exhibit high rates of false conflicts when transactions are long-running and large. False conflicts may slow down the execution significantly.

The main contribution of this paper is the design and analysis of a hardware alternative to Bloom filters that has been called the Interval Filter. Compared to a classical Bloom filter, the Interval Filter may show a lower false positive rate for those inserted elements that exhibit spatial locality according to some metric.

Hereinafter HTM is adopted in order to evaluate the proposed filter. However, results could extrapolate to other similar domains. Locality of reference will be exploited to store the locations read and written in an alternative way to Bloom filters, aiming to reduce false conflicts and enhance the execution of large transactions.

The rest of the paper is organized as follows: Section 2 defines the filter and explains how it operates. Section 3 places the Interval Filter within the Transactional Memory domain. Section 4 describes the simulation environment and evaluates the filter. Finally, conclusions are drawn in Section 5.

2 Interval Filter

The Interval Filter (IF) is proposed to reduce false positives in the presence of locality according to some metric. Without loss of generality, in the rest of the paper memory addresses will be considered as the elements to be inserted in the filter. Thus, intervals are defined as chunks of consecutive addresses that can be extracted out of a memory reference trace. Fig. 1 shows the design of the filter. IF comprises n intervals that are recorded as a pair of two full addresses, one representing the lower bound of the interval and the other one representing the upper bound. A valid bit per interval is also needed, $V_0, ..., V_{n-1}$. Each interval bound has two bit lines. Lower bounds are compared with the incoming address incremented by one and upper bounds are compared with the address decremented by one. Hence, $=^l_0, ..., =^l_{n-1}$ return true if the incremented address is equal to the corresponding lower bound of the interval. On the other hand, $>_0, ..., >_{n-1}$ return true if the address is greater than the lower bound. Likewise, $=^u_0, ..., =^u_{n-1}$ and $<_0, ..., <_{n-1}$ are the bit lines for the upper bounds of the intervals. The filter can be thought of as an extended full-associative cache.

Same primitive operations than Bloom filters can be performed with the interval filter. Fig. 1 shows, within dash-line boxes, how test for membership and insertions can be implemented. Test for membership consists in checking

the *Match* line to be true. This output line is computed by checking that the incoming address is within an interval. To do so, $>_0, ..., >_n$ and $<_0, ..., <_n$ bit lines can be used in the way shown in Fig. 1. Thus, lookups are relatively fast but insertions are slower and more complicate, as in Cuckoo-Bloom filters [9]. Actually, three cases come up on inserting an address into the interval filter, given that the address is not a member yet. Fig. 2 depicts the insertion algorithm flow chart:

Case 1 If every $=^l_0, ..., =^l_{n-1}$ and $=^u_0, ..., =^u_{n-1}$ bit lines are zero it means that neither valid intervals can be expanded so the incoming address must form a new interval in the filter. Thus, if the filter is not *Full* then the address is inserted into a non-valid interval by storing the original address (neither incremented nor decremented) in both bounds, lower and upper. Conversely, if the filter is *Full* then a valid interval is widen introducing false positives. In order to minimize the number of false positives due to widening, the closest interval bound is chosen. To do so, the address is XORed with the bounds whose $>$ or $<$ bit lines are set to 0. Then, the lower one is chosen as the candidate to store the address in an iterative manner.

Case 2 If either only one lower bound or only one upper bound is equal to the incremented/decremented address then an existing interval is to be widen. This can be done in a straightforward way by only storing the original address into the matched bound.

Case 3 If one lower bound and one upper bound are matched at the same time it means that the incoming address is the one who left to merge two existing intervals. Therefore, one of the two matched intervals is invalidated by clearing its V bit and the remaining interval is widen by setting its lower/upper bound to the lower/upper bound of the invalidated interval. In Fig. 2 the invalidated interval is i and it has been matched in the upper bound so the lower bound of the interval k is set to the lower bound of i. If k is chosen to be invalidated then the upper bound of i is set to the upper bound of k.

3 Application to Transactional Memory

A TM system must record the trace of memory references read and written within each transaction to detect conflicts among transactions. These addresses are classified into two separate sets, the Read Set (RS) and the Write Set (WS), which are also known as the signature of the transaction.

In the beginning, such signature was implemented by adding transactional Read/Write bits to each block in the cache memory. However, modifying caches to track transactional information have been proved to pose major constraints into TM virtualization since transactions are limited to cache sizes, scheduling time-slice (quantum), migration problems,... Also, cache memories are critical fine-tuned structures that should not be modified by including additional hardware. Therefore, Ceze et al. [4] proposed a signature implementation with

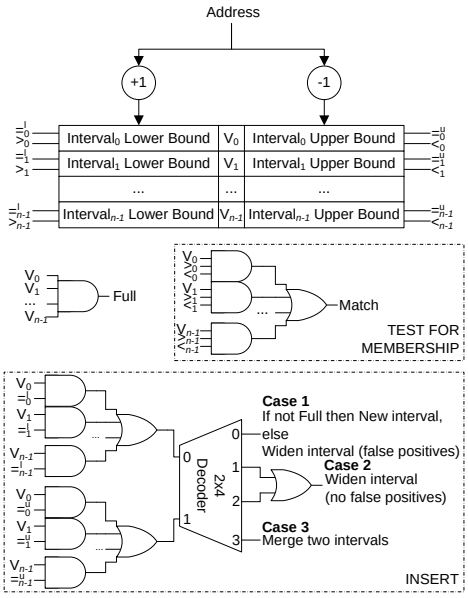

Fig. 1. Interval Filter Design.

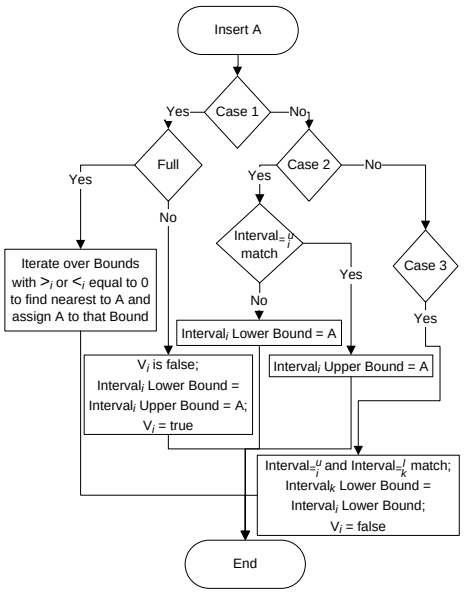

Fig. 2. Insertions Flow Chart.

per-thread Bloom filters fulfilling next requirements: signatures must not tolerate false negatives (undetected true conflicts) but may assume false positives (false conflicts); RS and WS sizes are unknown in advance, therefore, signatures should not limit the number of addresses to be stored; and test and insertion of an address should be fast operations.

Interval filters also fulfill the cited requirements. Moreover, they classified the trace of memory references into intervals to extract spatial locality features and to try to keep read and write sets as concise as possible. Therefore, the IF can be used instead of a Bloom filter to record the RS and WS of large transactions exhibiting locality.

4 Experimental Evaluation

This section is devoted to the experimental evaluation of the IF comparing its performance with that of a Bloom filter of similar hardware complexity. The simulation environment used to evaluate the IF comprises the Simics [7] full system execution-driven simulator and the GEMS's Ruby module [8] that implements the Transactional Memory system. Simics simulates the SPARC architecture and it is able to run an unmodified copy of a Solaris operating system. Solaris 10 has been installed on the simulated machine and all workloads run on top of it. Ruby has been modified to include the IF. The base CMP system consists of 16 in-order, single-issue cores. Each core has a 32KB split, 4-way associative, 64B block private L1 cache. L2 cache is unified, 8MB capacity, 16-bank, 8-way

associative, and 64B block size. Cache-coherence implements the MESI protocol and maintains an on-chip directory which holds a bit vector of sharers.

All workloads used are part of the Stanford's STAMP suite [3]. This suite is designed for Transactional Memory research and includes a wide range of applications laying emphasis on those with long-running transactions and large read and write sets. Such benchmarks are of special interest for signature evaluation since they put the most pressure on signatures.

Synopsys and CACTI 5.3 [10] were used to estimate the area of the Interval Filter and the Bloom filter involved in the evaluation. A SRAM memory with 8-byte words and four separate read/write ports was modeled with CACTI to estimate the Bloom filter area. CACTI was also used to model a full-associative SRAM memory with 32-bit words and 2 banks for the IF. Additional control logic and extra comparators and incrementers used by the IF were modeled with Synopsis and have been proven to have a small impact on the total area. Given an IF with $n = 10$ (i.e. ten intervals), the hardware-equivalent Bloom filter has 4 hash functions of the class H3 (H3 has proven better than others like Bit Selection [9]) and 2048 bits length. Both filters take about $0.09mm^2$ of die area each, using 65nm technology node. Hereinafter, results will be shown for an $n = 10$ interval filter compared to a 2048 bits, 4 hash function Bloom filter.

Experiments were carried out with 4 benchmarks from the STAMP suite: Bayes, Kmeans, Labyrinth and Yada. Such benchmarks exhibit the largest transaction data sets that cause Bloom filters to slowdown the execution because of false conflicts. Table 1 summarizes input parameters and the maximum and average RS/WS size in cache lines for those benchmarks.

Bench	**Input**	**# xact**	max $\|RS\|$	max $\|WS\|$	$\overline{\|RS\|}$	$\overline{\|WS\|}$
Bayes	-v32 -r4096 -n2 -p20 -s0 -i2 -e2	536	2171	1631	81.8	45.1
Kmeans	-m40 -n40 -t0.05 -i random-n1024-d1024-c16	1380	134	65	99.7	48.5
Labyrinth	-i random-x48-y48-z3-n64	158	529	510	128.7	120.7
Yada	-a20 -i dots.2	1338	578	405	60.5	37.5

Table 1. Parameters and data set maximum and average sizes.

The motivation behind the Interval Filter comes from the Fig. 3. This figure shows the interval histograms for the four benchmarks. It is a classification of the intervals by width, both in read sets and write sets of transactions, and it does not necessarily mean that every transaction in the system has intervals of every width. All the benchmarks show some amount of single addresses, i.e. width-1 intervals, but the most of the addresses can be classified into intervals wider than 1 by extracting spatial locality features. In fact, the number of single addresses in the benchmarks is between 2% and 22% as shown in Table 2. To keep track of address sets as intervals instead of doing so as single addresses could save in space and performance. Fig. 5 shows the execution time of the

Bench	Number of single addresses	
	Read Set	**Write Set**
Bayes	13.1%	2.7%
Kmeans	2.7%	2.5%
Labyrinth	4.2%	3.6%
Yada	22.0%	16.4%

Table 2. Percentage of single addresses.

Bloom filter versus the IF normalized to the perfect filter (i.e. infinite length, no false positives). Two cases can be observed:

- *Bayes and Yada:* Interval filter performs similar or slightly worse than Bloom filters concerning these benchmarks. Two things cause such slowdown: (i) the high percentage of single addresses, see Table 2, and (ii) transactions are made up of small-mid size intervals, as can be inferred from Fig. 3, since the largest interval in Bayes is about 100 addresses long in the figure, while the largest transaction is 2171 addresses long (see Table 1). Also, for Yada, the largest interval is 11 addresses, while the largest transaction is 578 addresses long. Therefore, having an interval filter with $n = 10$ intervals and a great amount of intervals to be stored in it, then "Case 1" (see Fig. 1) will be the most frequent hence introducing lots of false positives.
  Another important fact to consider is the creation of the intervals. Fig. 4 shows the interval creation in the write set of the largest transaction in each benchmark, i.e. it shows the result of having an infinite size interval filter in which addresses are inserted in order of appearance and the number of valid intervals are checked out after each insertion and then plotted. Notice that Bayes and Yada would need between 200 and 350 intervals to keep track of the whole set without false positives, however the interval filter used is 10 intervals. Flat parts in the Bayes curve corresponds to "Case 2" in Fig. 1.

- *Labyrinth and Kmeans*: Interval filter performs equal or better than Bloom filters for these benchmarks. Now the number of single addresses is lower than Bayes and Yada (Table 2) and large transactions are made up of a few large-size intervals, since Fig. 3 shows intervals greater than 400 addresses for Labyrinth while Table 1 shows maximum transactions about 500 and, 70 addresses intervals for Kmeans and maximum transactions of 70 and 130. Therefore, the interval filter does not get full immediately introducing few false positives. Fig. 4 shows a flat creation of intervals for Kmeans while Labyrinth shows a rise and fall that corresponds to interval merging, "Case 3" in Fig. 1.
  The behavior of these benchmarks is due to the data types they manage. Labyrinth makes a copy of a global multidimensional mesh inside a transaction which is represented as a multidimensional array. Kmeans keep a table of objects and attributes within an array. Conversely, Bayes and Yada use more complex and memory-scattered data structures as trees and lists.

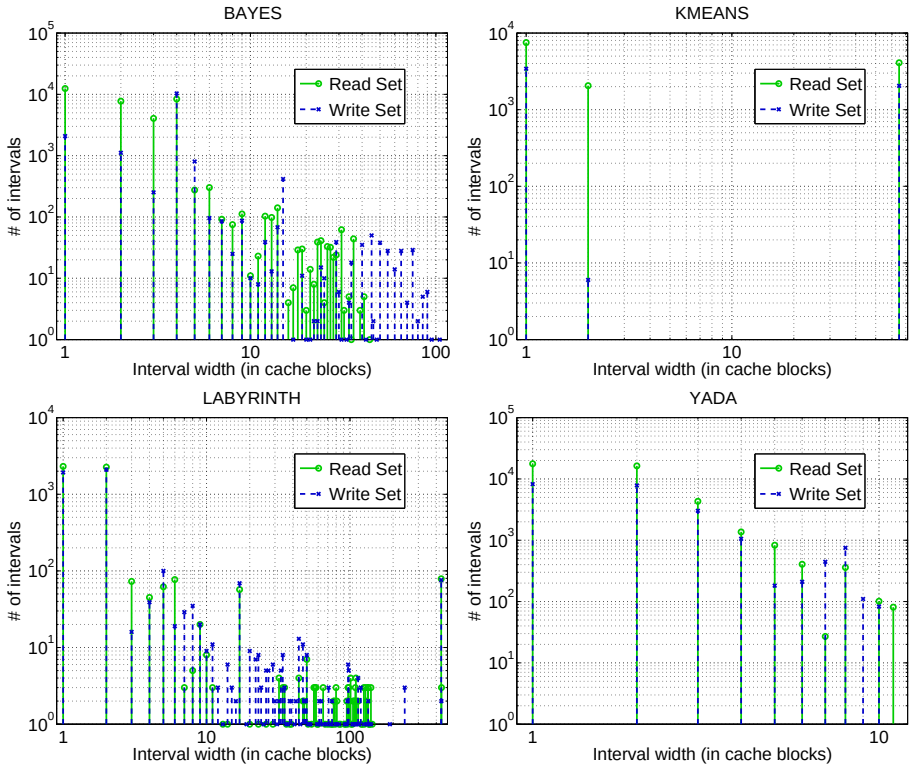

Fig. 3. Number of intervals of different widths for Bayes, Kmeans, Labyrinth and Yada, both RS and WS (log scale).

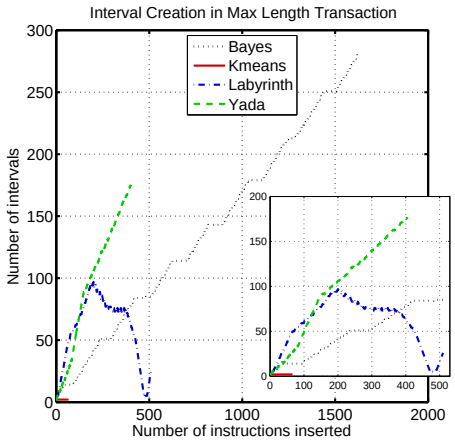

Fig. 4. Write set interval creation for maximum length transactions of Bayes, Kmeans, Labyrinth and Yada.

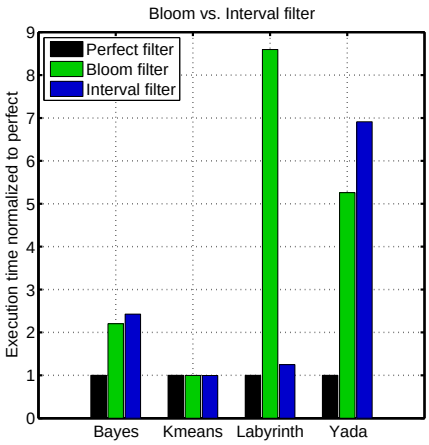

Fig. 5. Execution time of Bayes, Kmeans, Labyrinth and Yada normalized to the Perfect filter.

5 Conclusions

This paper proposes the classification of elements in a metric space into intervals to store them in a concise manner. For that purpose, a new Interval Filter hardware architecture has been developed that reduces false positives in the presence of locality as an alternative to Bloom filters. As case of study, the Interval Filter is evaluated in the context of Transactional Memory. The proposed filter is able to record contiguous addresses without introducing false positives. The Interval Filter presents excellent results when there exist few large intervals although it may perform similar or worse than regular Bloom filters for data streams with poor locality features.

Acknowledgment

This work has been supported by the Ministry of Education of Spain with project CICYT TIN2006-01078.

References

1. B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
2. A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.
3. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE Int'l Symp. on Workload Characterization (IISWC'08)*, 2008.
4. L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *33th Ann. Int'l. Symp. on Computer Architecture (ISCA'06)*, pages 227–238, 2006.
5. M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.
6. M. Jimeno, K.J. Christensen, and A. Roginsky. Two-tier bloom filter to achieve faster membership testing. *Electronics Letters*, 44(7):503–504, 2008.
7. P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
8. M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood. Multifacet's general execution-driven multiprocessor simulator GEMS toolset. *ACM SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.
9. D. Sanchez, L. Yen, M.D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *40th Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO'07)*, pages 123–133, 2007.
10. S. Thoziyoor, N. Muralimanohar, J. Ho Ahn, and N.P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.