Reduce Data Coherence Cost with An Area Efficient Double Layer Counting Bloom Filter

Kuan-Ting Chen, Ping-Ru Wu, Bo-Cheng Charles Lai
Department of Electronics Engineering
National Chiao Tung University
Hsinchu, Taiwan
rickie.ee99g@g2.nctu.edu.tw, killuaken@gmail.com, bclai@mail.nctu.edu.tw

***Abstract*—The snoopy protocol is a widely used scheme to maintain cache coherence. However, the protocol requires a broadcast scheme and forces substantial unnecessary data searches at the local cache. This paper proposes a novel Double Layer Counting Bloom Filter (DLCBF) to significantly reduce the redundant data searches and transmission. The DLCBF implements an extra layer of hash function and the counting feature at each filter entry. By using the hierarchical structure of the hash function, DLCBF can effectively increase the successful filter rates while requiring a smaller memory usage than the conventional Bloom filters. Experimental results show that the DLCBF can screen out 4.05X of unnecessary cache searches and use 18.75% less memory compared to conventional Bloom filters. The DLCBF is also used to filter out the redundant data transmission on a hierarchical shared bus. Simulation results show that the DLCBF outperforms conventional filters by 58% for local transmissions and 1.86X for remote transmissions.**

Keywords-multi-core; memory coherence; bloom filter; cache optmization;memory efficient design

I. INTRODUCTION

In modern computing systems, multi-core architectures have dominated from high-end servers to personal computers. The shared-memory Symmetric Multi-processor (SMP) is one of the main architectures that have been widely used. However, data sharing among multiple cores in a SMP introduces cache coherence issues [1]. In order to maintain a coherent memory system, a multi-core system needs to update or invalidate the shared data whenever one of its owners writes a new value to this data location. As depicted in Fig. 1, Core 1 and Core 3 share the same data (A = 5). When Core 1 performs a write operation to its copy of the shared data (A = 7), a coherent system updates the shared copy in Core 3 with the latest result or otherwise invalidates it.

A snoopy coherence protocol [1] is one of the popular methods to enable the cache coherence. When a data operation (read/write) is performed, the snoopy protocol forces all the caches in a multi-core system to check the data operation emerged on the shared interconnection. By invalidating each conflicting data, the protocol preserves data coherence. However, the snoopy coherence protocol adopts a broadcast scheme and every processor in the system needs to perform a search in its associated local cache whenever it sees a broadcast message. From the point of view of a parallel application, the snoopy coherence protocol is invoked whenever two or more processors need to exchange data. In general, the data sharing behavior is usually happened within a certain number of parallel tasks. The number of these affined tasks is typically much smaller than the size of the overall multi-core system. Therefore, if the snooping broadcast happens too frequently, a huge number of needless searches would be performed by the local cache which does not have the data specified by the broadcast message. For example, assume that there is a datum owned by only one processor. When this particular datum is written, the snoopy protocol would broadcast an invalidation message and invoke searches at all the caches while none of these searches are actually needed. This broadcast behavior of the snoopy protocol would unnecessarily put a cache in a busy mode that could increase the energy consumption as well as degrade the system performance.

This paper proposes a novel architecture of Double-Layer Counting Bloom Filter (DLCBF), and applies the DLCBF to filter out the unnecessary data communication and cache searches in a SMP system. A Bloom filter [2] is a classic unit used in database management. It uses hash functions to maintain the data mapping structure and provides an effective method to perform membership querying. However, due to the limited size of the filter, it suffers from rapid array saturation problem [3]. If the dataset of an application is too large, the data mapping structure would saturate and make the filtering mechanism ineffective. The DLCBF proposed by this paper implements an extra layer of hash function and the counting feature at each filter entry. By using the

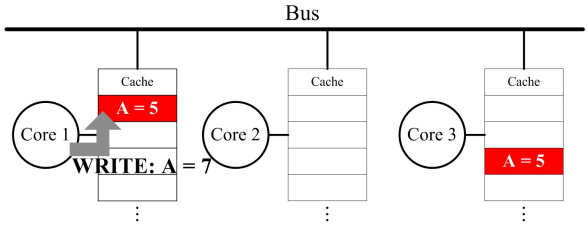

Figure 1. A coherence mechanism when multiple cores share the same data. The colored cache blocks are shared among different cores. They should be either updated or invalidated while this particular block is being written.

2168-3034/12 $26.00 © 2012 IEEE
DOI 10.1109/PAAP.2012.9

IEEE computer society

hierarchical structure of the hash function, DLCBF can effectively increase the successful filter rates while requiring a smaller memory usage than the conventional Bloom filters. The counting feature of DLCBF further enhances the ability to handle the array saturation issue.

This paper implements the DLCBF on two system modules of a SMP system to reduce unnecessary data processing of snoopy coherence protocol. The first module is the local cache of each processor. By connecting a Bloom filter between a cache and system interconnection, the filter mechanism can be used to screen out the unnecessary snooping messages that would be otherwise handled by each processor. The second module is the shared system bus. A bloom filter is embedded in the system interconnection to reduce the costly system-wide data broadcast. When compared with conventional bloom filters, the DLCBF can manage larger data set with fast data accesses while requiring smaller memory area. By deploying the DLCBF in a SMP system, a substantial amount of redundant memory operations and data transmission can be eliminated. In our experiment, the DLCBF can reduce up to 65.8% of unnecessary snoops to local caches with 18.75% less memory usage. Simulation results also show that the DLCBF outperforms conventional filters by 58% for local transmissions and 1.86X for remote transmissions.

The rest of the paper is organized as follows. In section II, we introduce the basic architecture of a Bloom filter (BF). Two modified versions, Counting Bloom Filter (CBF) and Banked Bloom Filter (BBF), are also discussed. Section III shows the proposed filter structure, Double Layer Counting Bloom Filter (DLCBF). The detail functionality and implementation concerns are also discussed. Section IV covers our evaluation methodology and demonstrates the cycle accurate simulation results. Finally, we conclude this paper in section V.

II. Previous Work on Bloom Filter Designs

To give a general background on membership querying techniques, this section will first introduce a simple hash-based method. Then three types of Bloom filters will be discussed, including classic Bloom filter (BF), counting Bloom filter (CBF), and banked Bloom filter (BBF). These three bloom filters are widely used designs and provide different advantages. The proposed DLCBF is a novel architecture which combines the features of these three filter types and benefits from all of them.

A. Simple Hash-based Technique

A membership querying function returns a value of true or false to identify the existence of a given input query. A straightforward way to implement a membership query function is to give an entry to each individual member. However, the total number of the members is usually much larger than the limitation of the memory size in a design. A hash function is a basic yet efficient solution for membership querying. An input query will be sent to a hash function, and a hashed value is returned to index the corresponding entry of the query. However, the single-index hash table is prone to returning many false positives for different queries. For

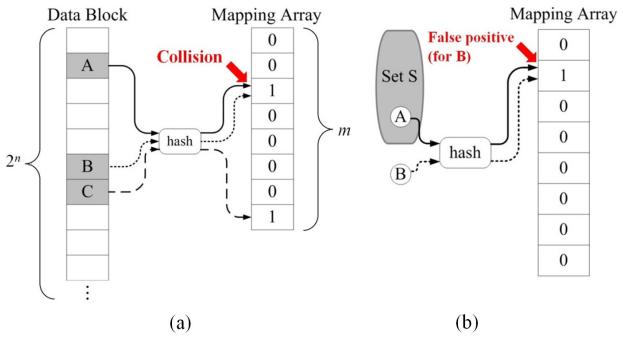

Figure 2. (a) Hash collision. (b) Hash reports a false positive for element *B*.

example, a computer system with *n*-bit memory addresses will introduce 2^n distinct memory locations. In an ideal case, a hash function needs 2^n bits to distinguish each data location. But due to the storage limitation in real systems, the hash function is forced to map the 2^n memory space to a mapping table with only *m* bits, where *m* is much smaller than 2^n. The parameter *m* may vary according to accuracy requirements and available resources. Thus, 2^n - *m* datum could be hashed to a bit that has already been used by another data (Fig. 2(a)). This is referred as a "collision". The collision problem could make the single-index hash function to report a false positive of membership querying. As an example shown in Fig. 2(b), there exists an element *A* belonging to a set *S*. The single-indexed hash unit provides *A* with a particular bit slot in the mapping table and sets this bit to 1. The value 1 indicates that *A* is in set *S*. But another element *B* that does not belong to *S* might also be hashed to the same slot. This conflicting scenario pollutes the meaning of the returned value and creates the situation of a false positive. From this returned value, users cannot tell if the element *B* has really been assigned to the slot or not.

B. Classic Bloom Filter (BF)

A Bloom filter is a space-efficient data structure proposed by Bloom in the 1970s [2]. Bloom filter uses multiple hash units for each element and sets several bits (depends on *k*, the number of hash functions) for each element. Fig. 3 shows how BF maps a single data to the mapping table with $k = 3$. A specific data *C* is considered in

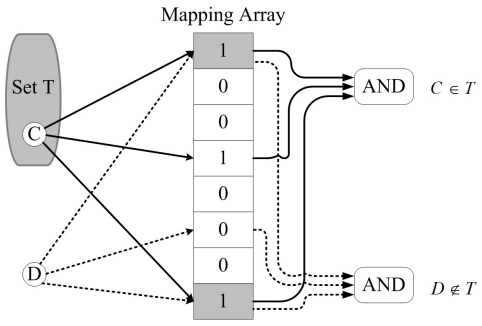

Figure 3. The mapping mechanism of a Bloom filter ($k = 3$)

a particular set *T* only when all the corresponding hashed slots are set. Fig. 3 also shows another element *D*, which does not belong to set *T*, and the corresponding hashed slots. Two of the slots collide with two of *C*'s slots (colored). However, there is another slot of *D* that is not set, so the Bloom filter correctly reports *D* as not in the set *T*. The Bloom filter has less possibility that reports a false positive than a simple hash function because collision must happen in all of the *k* hash functions.

C. Counting Bloom Filter (CBF)

Classic Bloom filter provides a memory-effective way of reducing hash collisions by using multiple hashes. However, a classic Bloom filter suffers from two problems. First, as the number of hash function increases, its mapping slot is "polluted" or "saturated" faster since the Bloom filter requires setting *k* bits for each element. Second, the classic Bloom filter does not support "deleting" or "resetting" the mapping slots. In other words, once a bit is set, the classic Bloom filter has no mechanism to reset it. Eventually, the classic Bloom filter will be filled up with 1's and loses its filtering functionality.

Since the multiple hash function is inevitable for Bloom filters, Fan et al. [4] proposed counting Bloom filter (CBF) to enable resetting a mapping slot. Counting Bloom filter adds an additional counter array along with the mapping slots of the classic Bloom filter. Each *l*-bit counter is associated with a mapping slot in a one-to-one fashion. Whenever an element is inserted to a set, each hashed slot will increment its corresponding counter by 1 and sets the mapping slot to 1. Therefore, the counter indicates the number of elements hashed to it, as depicted in Fig. 4. On the other hand, whenever an element is removed, each slot will decrement its corresponding counter. When a counter is decreased to zero, its corresponding mapping slot will be reset to zero. With this resetting procedure, CBF achieves a lower false positive rate and hence reduces the impact of saturation of the classic Bloom filter.

D. Banked Bloom Filter (BBF)

Both BF and CBF requires *k* lookups from the mapping table because of *k* hash functions. Making these lookups in a serial manner is inefficient and difficult to meet the timing constraint in hardware implementation. However, parallelizing *k* lookups requires large memory bandwidth, so each memory in the filter has to implement *k* read/write ports for querying and updating elements. Banked Bloom filter was proposed to address the issue [5]. Similar to the banked cache access, BBF supports required bandwidth by using banking instead of adding read/write ports. Assume a memory with *p* ports and each port has *B* banks, it can provide a maximum of *p*·*B* simultaneous access as long as no more than *p* operations are accessing the same bank [5].

When applying Bloom filters to the local cache or interconnection of a SMP system, the filters are usually accessed at every cycle. Therefore a delay in the filter is undesirable. However, bank conflicts will stall the accessing procedure and make the filter to be ineffective. Banked Bloom filter uses a hard-wired permutation table to prevent bank conflicts. Fig. 5(a) shows how a banked Bloom filter is organized. With four hash functions, the BBF is configured as four banks to provide a memory bandwidth of $1 \times 4 = 4$ accesses simultaneously. Whenever there is a membership querying to the filter, the permutation table will return a sequence of bit sets to the multiplexers and guide the hash functions to the corresponding banks. Fig. 5(b) depicts a permutation table with four banks ($k = 4$).

III. Double Layer Counting Bloom Filter

Both the classic Bloom filter and counting Bloom filter have difficulty of providing the high accessing bandwidth required by *k* hash functions. The banked Bloom filter mitigates this problem by banked memory accesses. However, all filters still pose significant storage overhead. The classic Bloom filter implements a 1-bit-per-slot, but needs a large number of slots to alleviate the array saturation issue. Although the banked Bloom filter provides large memory bandwidth, it has the same 1-bit-per-slot structure as the classic Bloom filter. The counting Bloom filter, on the other hand, can handle array saturation better than the classic Bloom filter and banked Bloom filter, but it increases the storage overhead by using multi-bit counters instead of the single set bit.

To reduce the storage overhead and further increase membership querying speed while preserving the functionality of the Bloom filter, we propose a novel Double Layer Counting Bloom Filter architecture (DLCBF). The

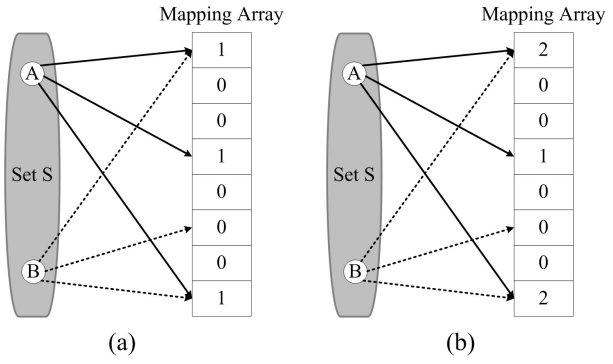

Figure 4. Different data maps to same slots in mapping array. In (a), we cannot tell if a slot is mapped multiple times. In (b), we can decrease the counter to indicate removal of an element.

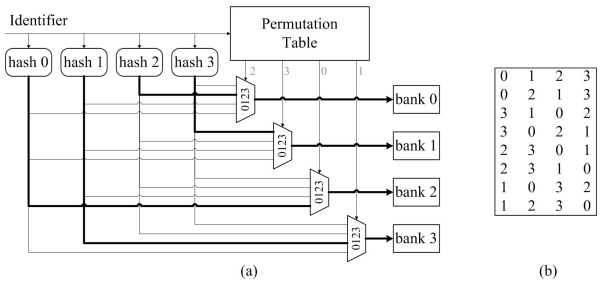

Figure 5. (a) Banked Bloom filter with four hash functions. (b) Hard-wired permutation table of BBF.

DLCBF adopts the idea of two-tier Bloom filter [6]. By adding a second tier cache Bloom filter, the two-tier Bloom filter is able to reduce the membership querying time. Whenever a query is invoked, the two-tier filter will check both the filter cache and main filter simultaneously. If the queried element is cached, the faster filter cache will respond with either a true or a false. Otherwise, the main filter will be responsible for returning the result of membership querying.

The proposed DLCBF differs from the two-tier Bloom filter in the utilization of the extra "tier" or "layer". The DLCBF introduces an extra upper layer of hash units upon a banked Bloom filter (lower layer) to catch the data locality behavior. In other words, the upper layer is responsible for the membership querying for a consecutive memory region, not only for a single element. Besides, the extra hash function can generate different permutation for lower layer hash functions. Fig. 6 depicts the memory structure of the DLCBF.

There are three main benefits enabled by DLCBF. First DLCBF adopts the banked structure and adds an extra layer to meet the heavy memory bandwidth requirement [5] and reduce membership querying time. With multiple access banks, the DLCBF requires only one lookup for each banked memory while CBF needs k lookups with k different hash functions. Second, DLCBF takes the advantage of data locality. It separates memory space into different regions. Since data in different regions are less likely to be accessed in a consecutive way, an extra layer can filter out unnecessary data operations with higher speed. Third, DLCBF benefits from the extra permutation array, which further lowers the probability of data collision.

IV. EVALUATION

This section first shows the experimental setup used in this paper. Then the performance of DLCBF is analyzed and compared with other Bloom filters in a SMP system. The DLCBF will be implemented on two system modules to reduce unnecessary data processing. The first module is the local cache of each processor. The DLCBF is used to reduce the unnecessary cache searches from snoopy coherence protocol. The second module is on the shared system bus to filter out the redundant data transmission.

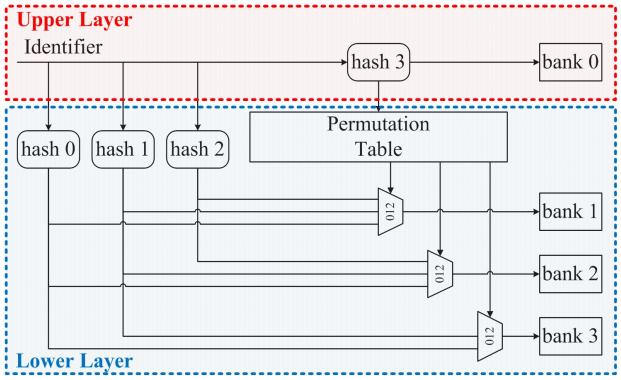

Figure 6. Architecture of Double Layer Counting Bloom Filter.

A. Experimental Setup

We use GEM5 [7], a full-system event-driven simulator, as our experiment platform. We use a simple in-order processor model so that we can evaluate the proposed scheme within a reasonable simulation time. Table I lists the configuration parameters we used in our simulations. A single transaction shared bus takes charge of the communication among processors. The experiments were executed with 11 representative workloads from PARSEC benchmark suite [8].

TABLE I. PROCESSOR AND CACHE/MEMORY PARAMETERS

Component	Parameters
Processor core	2GHz, single-issue in-order
Block size	64bytes
L1 I-caches (Private)	32kB, 2-way, 2-cycle
L1 D-caches (Private)	64kB, 2-way, 2-cycle
Memory	60-cycle access latency
Bus	1GHz, single transaction, 1-cycle overhead/transaction

Four types of Bloom filters were implemented and compared, including the classic Bloom filter (BF), counting Bloom filter (CBF), banked Bloom filter (BBF), and the proposed double layer counting Bloom filter (DLCBF). As depicted in Fig. 7, the Bloom filters were connected in between of caches and the shared bus. All the filters in the experiment used four hash functions. The classic Bloom filter, CBF, and BBF are implemented with 1K bytes of memory per processor core. The DLCBF uses only 0.8175K bytes per core, bringing an 18.75% of memory usage reduction. This difference is from the upper layer of memory array, which is only 25% of a standard memory array.

B. Results of Filtering Unnecessary Cache Searches

Fig. 8 shows the filtered snooping rate using classic Bloom filter, CBF, BBF, and DLCBF for each benchmark. The filtered rate represents how many snoops are screened out by the filter. These are the unnecessary snoops that do not need to be handled by a cache. The experiments were performed on multi-core systems with two, four, eight, and sixteen processors. The right most column of Fig. 8 represents the geometric mean of the filtered rates observed in the benchmarks. We can see that the classic Bloom filter performs poorly in all benchmarks. The reason is that the classic Bloom filter faces array saturation problem and does not support "resetting" a slot whenever an element is removed from the set. In this context, removal of an element is considered as cache line invalidation or eviction. Therefore, its mapping array saturates even when data set

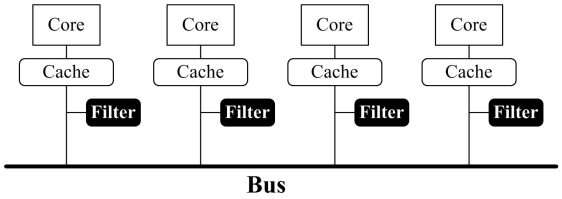

Figure 7. SMP with Bloom filters.

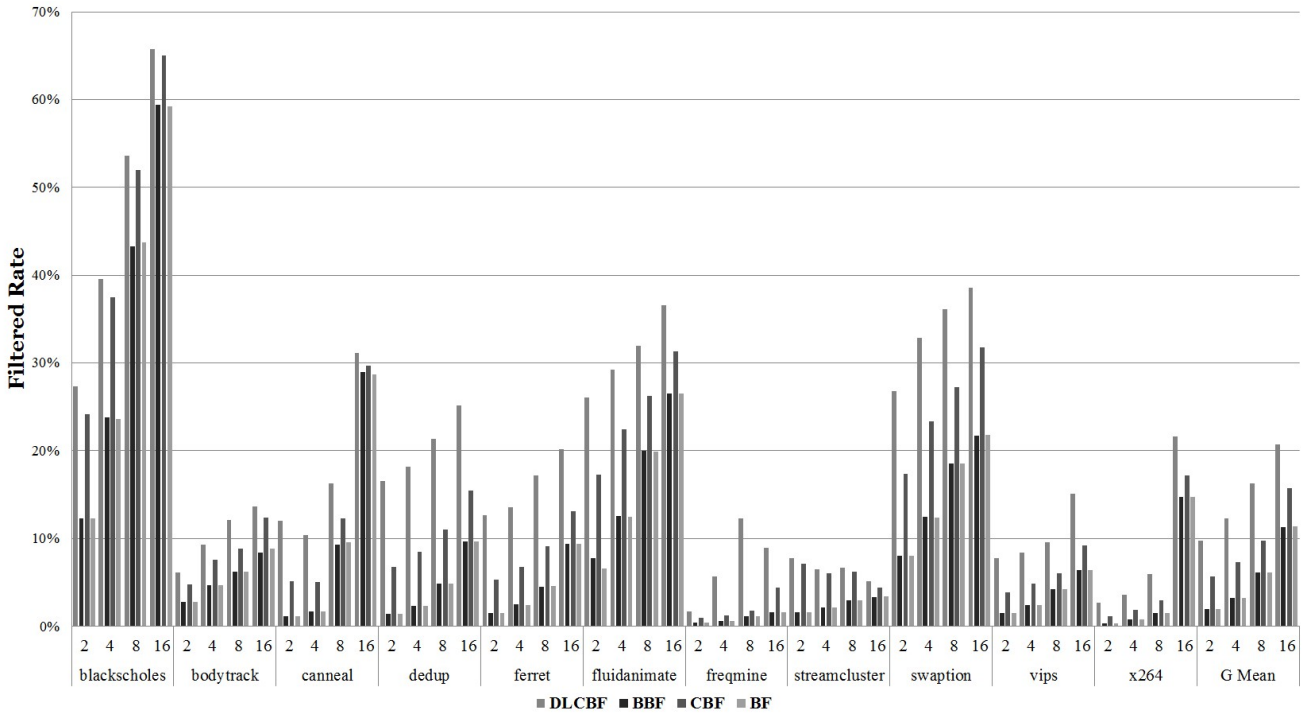

Figure 8. Filtered rate of classic Bloom filter (BF), counting Bloom filter (CBF), banked Bloom filter (BBF), and double layer counting Bloom filter (DLCBF).

size is small and the classic Bloom filter loses its filtering functionality very quickly. Banked Bloom filter suffers from the same reason, although it supports faster querying access. On the other hand, because the counting feature enables the "resetting" ability, CBF reduces the array saturation rate. With lower saturation rates, CBF achieves better filtering behavior than the classic and banked Bloom filters. When compared with a classic Bloom filter, CBF achieves up to 3.57X filtered rate improvement.

With an additional hash function and hard-wired permutation table, DLCBF divides and maps the whole memory space to several storage arrays. This design avoids the data in different memory spaces colliding with each other. Inside each storage array, DLCBF utilizes multiple hash functions to prevent collision. In addition, the counting feature enables a much lower probability of collision and the ability to reset an element more effectively. Therefore, DLCBF further improves the filtered rate and significantly outperforms the classic Bloom filter, CBF, and BBF. The average improvement of filtered rate is 4.05X and 72.31% when compared to classic Bloom filter and CBF respectively.

Another observation from Fig. 8 shows that the filtered rate increases with the number of processors for all four filters. This is because that when the number of processors increases, each processor is responsible for a smaller size of data set. For example, assume the total data set is 1MB; each processor in a 2-core multi-processor system would deal with 512kB of data. And in a 4-processor system, each core will be responsible for only 256kB of data. The effective data set size allocated to each individual core decreases with the increasing number of processors in a system. The smaller effective data set size lowers the memory space that needs to be handled by each filter. Therefore, the characteristics of each filter are improved with more processors in a system.

C. Results of Filtering Unnecessary Data Transmission on A Shared Segmented Bus

The segmented bus is proposed as an energy-efficient, bus-based on-chip interconnection [9]. It separates a long bus into several shorter buses and organizes them in a hierarchical manner. Fig. 9 depicts a 16-core processor with the segmented bus and filters. As we know, not every coherence transaction is expected to be broadcasted to every processor. A segmented bus implements two filters, In-filter and Out-filter, at each sub-bus to maintain some knowledge of cache contents in the local and remote segments and screens out unnecessary transactions. The In-filter keeps track of cache lines that are currently in the segment. When a

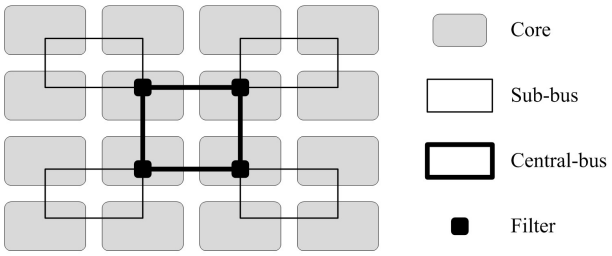

Figure 9. Segmented bus architecture for 16-core processor.

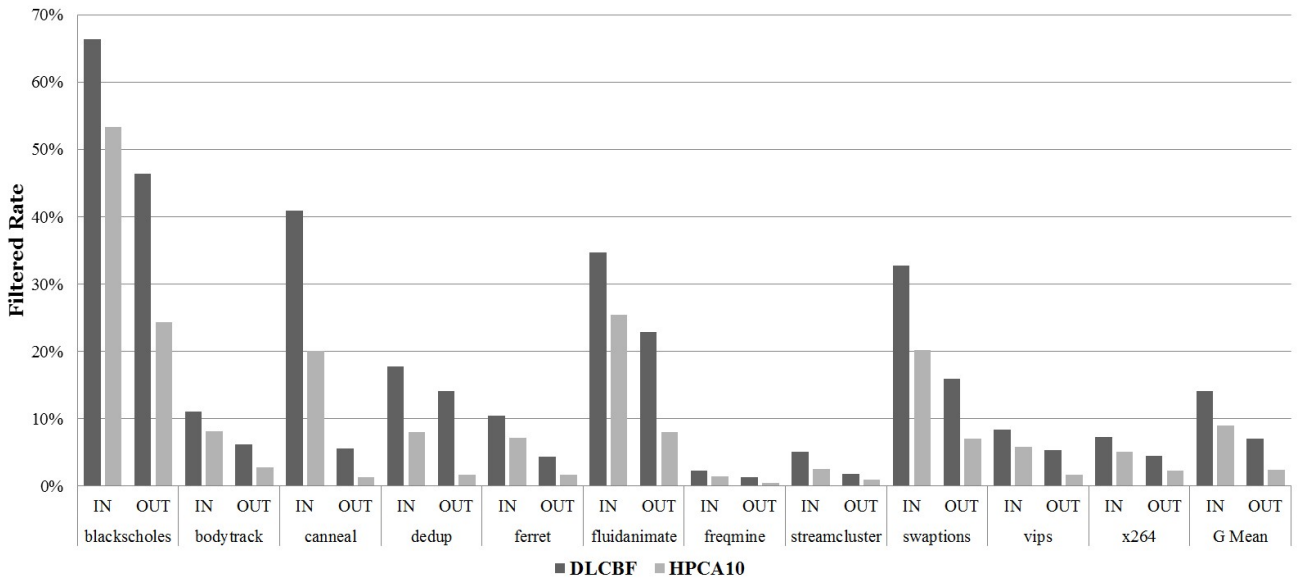

Figure 10. Filtered rate of DLCBF and HPCA10 filter.

broadcast is invoked, every segment will first look up at its local In-filter. And a broadcast to local processors will be performed if the In-filter allows. The Out-filter keeps track of cache lines that is sent out to remote segments. If a specific cache line has never been sent out before, the local segment does not broadcast it to the remote segments. Each segmented bus filter contains two arrays of 8192 entries, one array for In-filter and the other for Out-filter. Every entry is consisted of a 10-bit counter. In all, each filter requires 20K bytes of storage overhead.

In this experiment, we apply the DLCBF to a segmented bus. We simulated a 16-processor system with a segmented bus. We compared two filter schemes. The first scheme is the Bloom filter design (HPCA10) used in [9], and the second scheme adopts the proposed DLCBF. The HPCA10 is implemented with 20kB of memory, while the DLCBF is 13kB in size. Fig. 10 shows the simulation results, DLCBF and HPCA10 [9] are compared. Basically, the HPCA10 filter is a counting Bloom filter with big counters and a large storage. The outperforming filtered rate of DLCBF confirms that the additional hash is helpful even when compared to a large CBF. The average improvement of filtered rate is 58% for In-filter and 1.86X for Out-filter in comparison to HPCA10.

V. Conclusion

In this paper, we propose an area efficient double layer architecture of Bloom filter, DLCBF. By adding extra filtering layer, DLCBF can use 18.75% smaller memory to achieve 4.05X and 72.31% better filtered rate when compared with a classic Bloom filter and CBF respectively. When applying on segmented bus, DLCBF outperforms HPCA10 filter by 58% for In-filter and 1.86X for Out-filter.

Acknowledgment

This work is sponsored by Nation Science Council under grant NSC 101-2220-E-009 -037.

References

[1] J. L. Hennessy, and D. A. Patterson, Computer architecture: a quantitative approach: Morgan Kaufmann Pub, 2011.

[2] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Commun. ACM, vol. 13, no. 7, pp. 422-426, 1970.

[3] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee, “Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches,” in Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design, San Fancisco, CA, USA, 2009, pp. 165-170.

[4] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” IEEE/ACM Trans. Netw., vol. 8, no. 3, pp. 281-293, 2000.

[5] M. Breternitz, G. H. Loh, B. Black, J. Rupley, P. G. Sassone, W. Attrot et al., “A Segmented Bloom Filter Algorithm for Efficient Predictors,” in Computer Architecture and High Performance Computing, 2008. SBAC-PAD '08. 20th International Symposium on, 2008, pp. 123-130.

[6] M. Jimeno, K. J. Christensen, and A. Roginsky, “Two-tier Bloom filter to achieve faster membership testing,” Electronics Letters, vol. 44, no. 7, pp. 503-504, 2008.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu et al., “The gem5 simulator,” SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1-7, 2011.

[8] C. Bienia, “Benchmarking modern multiprocessors,” Princeton University, 2011.

[9] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian, “Towards scalable, energy-efficient, bus-based on-chip networks,” in High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, 2010, pp. 1-12.