Leveraging Bloom Filters for Smart Search Within NUCA Caches

Robert Ricci, Steve Barrus, Dan Gebhardt, and Rajeev Balasubramonian
School of Computing, University of Utah
{ricci,sbarrus,gebhardt,rajeev}@cs.utah.edu

Abstract

On-chip wire delays are becoming increasingly problematic in modern microprocessors. To alleviate the negative effect of wire delays, architects have considered splitting up large L2/L3 caches into several banks, with each bank having a different access latency depending on its physical proximity to the core. In particular, several recent papers have considered dynamic non-uniform cache architectures (D-NUCA) for chip multi-processors. These caches are dynamic in the sense that cache lines may migrate towards the cores that access them most frequently. In order to realize the benefits of data migration, however, a "smart search" mechanism for finding the location of a given cache line is necessary. These papers assume an oracle and leave the smart search for future work. Existing search mechanisms either entail high performance overheads or inordinate storage overheads. In this paper, we propose a smart search mechanism, based on Bloom filters. Our approach is complexity-effective: it has the potential to reduce the performance and storage overheads of D-NUCA implementations. Also, Bloom filters are simple structures that incur little design complexity. We present the results of our initial explorations, showing the promise of our novel search mechanism.

1 Introduction

It is well-known that on-chip wire delays are emerging as a major bottleneck in the design of high-performance microprocessor chips. As feature sizes are reduced, wire delays do not scale down at the same rate as logic delays [1, 5]. It has been projected that at 35nm technologies, less than 1% of the total chip area will be reachable in a single cycle [1]. Communication between distant modules on a chip will therefore cost tens of cycles and will negatively impact performance.

On-chip cache hierarchies bear the brunt of growing wire delays as they occupy a large fraction of chip area in modern microprocessors. For example, more than two-thirds of the chip area in Intel's Montecito [11, 12] can be attributed to L3 caches that have a capacity of 24MB. Such large cache structures are typically organized as numerous banks to help reduce latency and power consumption [14]. Given an input address, the request is routed to a subset of banks that then service the request. The latency for any cache access is a function of the distance between the bank that contains the requested data and the cache controller. This observation motivated the proposal by Kim *et al.* [7] of a non-uniform cache architecture (NUCA). Within a NUCA organization, the latency for a cache access may be as little as a handful of cycles if the data is located close to the cache controller, or up to 60 cycles if the data is located in a distant bank. This architecture is unlike a conventional cache organization where the cache latency is uniform and determined by the worst-case delay to access any block. Recent proposals have extended NUCA designs to also handle chip multiprocessors [2, 4, 6].

NUCA organizations have been classified as static (S-NUCA) and dynamic (D-NUCA) in the literature [7]. In static-NUCA, an address is mapped to a unique cache bank. Given an address, the cache controller sends the request to a single bank (typically determined by examining the address index bits). While such a mechanism is simple, it does not take advantage of locality. The L2 or L3 cache latency for a data structure is set as soon as it is allocated in the physical memory address space. Dynamic-NUCA attempts to improve performance by leveraging locality and moving recently accessed blocks to banks that are close to

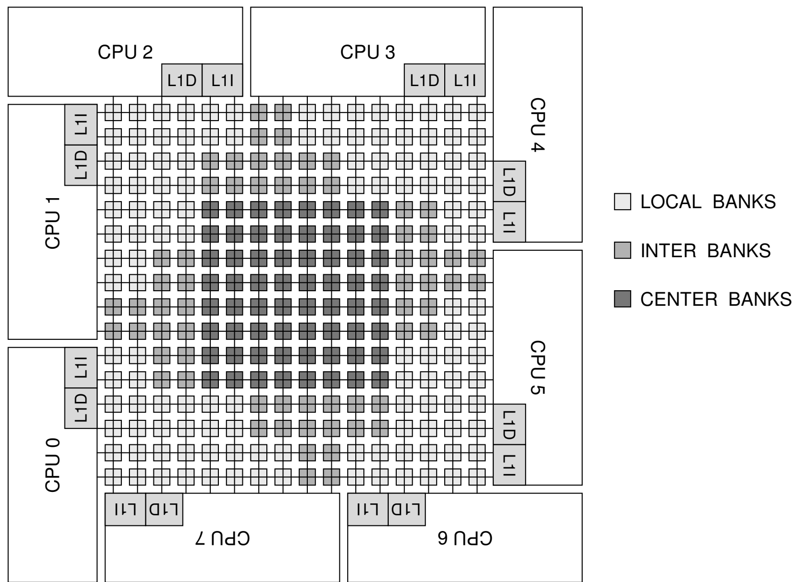

Figure 1. Baseline CMP with 8 cores that share a NUCA L2 cache. The L2 is partitioned into 256 banks and each block is allowed to reside in one of 16 possible banks.

the cache controller. A block is now allowed to reside in different cache banks at different times. Given an address, the cache controller identifies a number of candidate banks to which the request can be sent. In one approach, the banks can be sequentially probed until the data is located – this can significantly increase cache access latency. In a second approach, the banks can be probed in parallel – this can increase contention cycles because of the increased bandwidth pressure on the inter-bank network. Hybrids of the two search approaches have also been proposed [2, 7].

A recent paper explores block migration policies for a D-NUCA organization in a CMP [2]. The authors show that D-NUCA can improve performance, relative to an S-NUCA organization, provided there exists an oracle to identify the bank that stores the data. If a realistic hybrid data search mechanism is incorporated, performance is actually worse than that of S-NUCA. Hence, for D-NUCA to be effective, the cache controller must identify a small subset of banks to probe, with high accuracy. Kim *et al.* [7] propose one mechanism for such a smart search for a single-core chip. The cache controller maintains a partial tag array that stores six bits of the tag for each cache line. The cache controller then forwards the request to only those banks that have tags that match the address. While this mechanism has high accuracy, Beckmann and Wood [2] point out that such partial tag arrays can lead to extremely high storage overheads. In an 8-core CMP with 16MB of cache and block size of 64 bytes, the total storage for 6-bit partial tags is as high as 1.5MB (roughly 72M transistors). The efficient design of a "D-NUCA smart search mechanism" is considered an open problem and an important factor in the success of block migration policies [2].

This paper presents a complexity-effective solution to the smart search problem. It takes advantage of Bloom filters to identify candidate cache banks with high accuracy. Compared to partial tag arrays, it reduces storage requirements by an order of magnitude. Power consumption can also be reduced by avoiding access to large tag arrays and by avoiding transmission of tags on the inter-bank network. This reduction in network traffic decreases routing congestion and can improve latency for all L2 transactions. Bloom filter updates and look-ups are achieved with simple indexing functions.

Section 2 provides background on Bloom filters and the baseline NUCA organization. Section 3 details our proposed smart search mechanism. We present a preliminary analysis of our approach in Section 4 and draw conclusions in Section 5.

2 Background and Related Work

2.1 Non-Uniform Cache Architectures

The baseline processor organization that we use is similar to that of Beckmann and Wood [2] and is illustrated in Figure 1. Each processor core (including L1 data and instruction caches) is placed on the chip boundary and eight such cores surround a shared L2 cache. The L2 is partitioned into 256 banks and connected with a mesh interconnection network. Each core has a cache controller that routes the core's requests to appropriate cache banks.

In a static-NUCA organization, eight bits of the block's physical address can be used to identify the unique bank that the block maps to. Each bank will have to be set-associative to reduce conflict misses.

In an alternative static-NUCA organization, each bank can accommodate a single way. If the L2 cache is 16-way set-associative, a given block can map to 16 possible banks and four bits of the block's physical address are used to identify this subset of banks. When a block is brought into the cache, LRU (or even the block's address) can determine where the block is placed and the block remains there until it is evicted. This S-NUCA organization is no better than the organization described in the previous paragraph (in terms of performance) and requires a "smart search mechanism" to identify the bank that contains the block. We describe it here because it forms the basis for dynamic-NUCA organizations.

In dynamic-NUCA, LRU/block address determines where the block is initially placed. Access counters associated with each cache line keep track of the processor cores that request the block. The block is gradually moved to a bank that best reflects the "center of gravity of processor requests". Of the 16 candidate cache banks (ways) for a block, eight are in close proximity to each of the eight cores (referred to as the *local* banks), four are in the *center* region, and the remaining four are in the *intermediate* region (shown in Figure 1 by different colors). These 16 banks are classified as a single *bankset* and there exist 16 such banksets. The L2 cache is partitioned into 16 *bankclusters*. Each bankcluster contains one bank from every bankset. Block migration causes a block to move between bankclusters, or stated alternatively, between banks within a bankset.

Given a block address, the cache controller must potentially forward the request to 16 different banks in order to locate the block. The smart search mechanism proposed in this paper will be employed in this context. Beckmann and Wood adopt the following mechanism: the core's local, intermediate, and four center banks are searched in parallel; if the block is not located, the other ten candidate banks are searched in parallel. Such a mechanism entails high performance overheads and usually negates any performance improvements from block migration. The search mechanism of Kim *et al.* [7] maintains partial tag arrays at the cache controller to identify a small subset of banks that can be probed in parallel. Such a mechanism, extended to CMPs, would entail marginal performance overheads, but incur non-trivial storage overheads. Our proposed mechanism has the potential to incur marginal performance and storage overheads.

Block migration incurs other complexities as well (regardless of the search mechanism). Firstly, access counters must be maintained for every cache line (or blocks can be aggressively migrated on every access). Block migration cannot happen simultaneously with block look-up, else there is the potential to signal a false miss (the request fails to see the block in transit and triggers an L2 miss). A four-phase protocol is required to implement migration correctly: (i) cores are informed of the migration so that accesses to that block are temporarily disabled, (ii) acknowledgments are received from the cores, (iii) migration is effected, (iv) cores are informed after migration so that accesses can be enabled again. Block migration requires that the search hints at each core be updated. The search mechanism may dictate the amount of network traffic required to update search hints.

2.2 Bloom Filters

We base our smart search algorithm on the concept of Bloom filters [3]. Here, we describe general Bloom filters. The particular variant that we use is detailed in Section 3.3.

A Bloom filter is a structure for maintaining probabilistic set membership. It trades off a small chance of false positives for a very compact representation. A general Bloom filter cannot return false negatives.

A Bloom filter consists of an array A of m one-bit entries and k hash functions, $\{h_1, h_2, ...h_k\}$. In an empty filter, all bits in A are zero. To add item i to

the filter, $h_1(i)$ is computed. This value is then used to index into A, and $A[h_1(i)]$ is set to one. This process is repeated with each hash function, up to $h_k(i)$.

Testing for set membership is straightforward. To test for the presence of i', we examine $A[h_1(i')]$. If it is zero, then we can tell that i' is not present in the set. If it is one, we continue to check $A[h_2(i')]$ and so on, up to $A[h_k(i')]$. If all such bits are one, the set membership test returns $true$, but if any are zero, the membership test returns $false$.

In set membership tests, false negatives are impossible—if item i has been added to the filter, then we are guaranteed that:

$$\forall j\{1 <= j <= k : A[h_j(i)] = 1\}$$

False positives, however, are possible; any or all of the bits in A could have been set to one by the insertion of a different item. Assuming "good" hash functions, the probability of getting a false positive from a Bloom filter is approximately $1 - e^{-kn/m}$ [3], where n is the number of items inserted into the set, k is the number of hash functions, and m is the size of array A. For example, for $k = 5, m = 2048$, and $n = 256$, the probability of a false positive is 2.1%. Such a filter requires only 2048 bits of space, while an equivalent table with 256 entries for 32-bit numbers would require 8192 bits of space.

It is not possible to remove elements from a Bloom filter. If we were to try to do so, setting any of the bits in A to zero could interfere with another element in the set, causing a false negative. Removal requires a "counting" Bloom filter, in which we keep counts of how many elements in the set require a particular bit in A to be set. Such filters, however, have higher storage requirements, because each entry in A must now hold a counter rather than a single bit. Thus, we do not use counting Bloom filters in this paper.

3 Smart Search Design

Our smart search algorithm is based on the idea of keeping *approximate* information about the contents of L2 bankclusters. Kim *et al.*'s [7] partial tag array uses a similar strategy, though in the context of single-core chips. Such arrays would be prohibitively large for processors with many cores and many bankclusters. Thus, we turn to Bloom filters as a compact approximation.

3.1 Overall Design

At each core on the CMP, we maintain an approximation of the cache lines present in each L2 bankcluster. For our baseline processor (Figure 1), each of the 8 cores maintains 16 filters, one for each L2 bankcluster. While the number of filters is large, as we shall see shortly, each filter is a relatively small structure.

The filters are used to direct L1 misses to the L2 bank or banks that seem likely to contain the requested cache line. The search proceeds in a similar manner to Beckmann and Wood's [2]; when a core's memory access misses in L1, the core's cache controller looks for the cache line in the L2 bankclusters. First, it consults the core's local L2 bankcluster, the Center, and the Inter bankclusters. If migration has had its intended effect, these are the most likely places to find the line. If none of those bankclusters hit, the core tries the other cores' local bankclusters before going to main memory. In our search, we filter out request messages by consulting, in parallel, the Bloom filters for each bankcluster, and only sending messages to those that hit. Due to the possibility of false negatives, discussed in the next section, if no Bloom filters hit, or all of the bankclusters consulted turn out to be false positives, we send messages to all bankclusters skipped in the first pass.

3.2 Managing the Filters

We next deal with content management: when are items inserted into filter, and when are they removed? The design principle we follow here is to keep complexity low by minimizing our changes to the baseline cache design. Thus, rather than adding new messages for filter management, we use the messages already sent by the cache. When one of the L2 banks inserts a new line, we do not broadcast this event to all filters. Instead, we only insert addresses into our filters on-demand; that is, when a core receives a line from some L2 bank. The first time a cache line is accessed by a given core, therefore, there is a compulsory miss in the filters. For this reason, our filters can return a false negative. We also see similar false negatives when a cache line has migrated from one bankcluster to another. Because the probability of false positives returned by a Bloom filter is a function of the number of items inserted into it, inserting no more items than necessary will help keep the false positive rate low.

All filters start empty, with the bits in their arrays cleared. The first time core c accesses line l, it will have no entry for l in any of its filters. Assuming no false positives, all filters will miss, and c will fall back to searching all banks for l. If l is present in bankcluster b, b will send the line to c. Upon receipt of the line, c inserts l into its filter for bankcluster b. Thus, the next time that c misses on l, it will know that b has at some point held line l.

As time goes on, the filters will tend to return more false positives for two reasons. First, due to the fundamental properties of Bloom filters, as more items are inserted into a filter, its false positive rate rises. Second, as time passes, the information in the filters becomes stale; some of the entries they have stored will no longer be accurate because lines may have migrated or been otherwise evicted from their original bankclusters.

We clearly must have a mechanism in place for removing filter entries to avoid a constantly-rising false positive rate. Recall that in a simple Bloom filter, entries cannot be removed. In addition, if we were to track migrations or evictions, this would require new messages between bankclusters and the filters.

In order to keep the complexity of our design down, strategy we adopt is to clear an entire filter when its false positive rate becomes too high. This decision can be made locally by each filter, without requiring global co-ordination. Filters can, for example, maintain a simple n-bit saturating counter that is incremented on each true positive and decremented on each false positive. This provides an approximation of the false positive rate over the last several memory accesses. For our initial implementation, we clear a filter when its ratio of false positives to true positives goes above one—that is, when the false positive rate reaches 50%. There is clearly a tradeoff between clearing the filters frequently to reduce false positives and clearing them infrequently to minimize the number of compulsory misses thus incurred. This is a tradeoff we will explore in future work.

3.3 Filter Design

The filter we chose is similar to the partitioned-address Bloom filter used by Peir *et al.* [13] for predicting cache misses.

The bit array, A is divided into k slices, $A_1, A_2, ...A_k$. Each hash function indexes into its corresponding slice. For example, h_1 indexes into A_1, h_2 indexes into A_2, and so forth. Each slice can be implemented as a separate structure, keeping the time required to access it small. All slices can be accessed in parallel, meaning that the lookup time is independent of the number of slices.

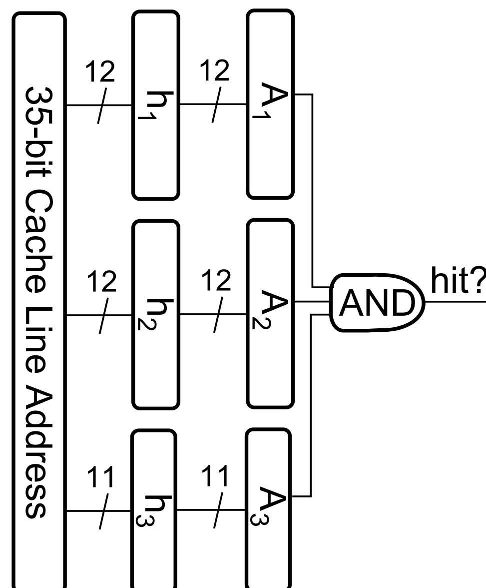

Figure 2. Design of our Bloom filter, showing hash functions h_1 through h_3, which are used as indexes into bit array slices A_1 through A_3. A hit is detected when ones are returned from all array slices.

We use the simplest hash function possible, the identity function. We split up the address into several sets of bits, then use each to index into one of the array slices.

The layout of our filter is depicted in Figure 2. We assume a physical address width of 41 bits—this is one bit wider than the current AMD Athlon 64 and Sun Niagara [8] processors, and can accommodate two terabytes of memory. We also assume 64 byte wide cache lines, giving 6-bit offsets. This leaves us with 35-bit cache line addresses. We divide these bits into two groups of 12 and one group of 11, and use each group to index into an array slice. Our design can easily be adapted to other address widths by changing the number and size of the array slices.

Our filters are reasonably sized structures; there are two arrays of 4096 bits each, and one array of 2048 bits. The total size for one filter is thus 10 Kb. Since

each core needs 16 filters, one for each bankcluster, each core requires 160 Kb, and all eight cores together require 1280 Kb, or 160 KB, of storage. Assuming 6-transistor SRAM cells, the total filter size is approximately 7.68 million transistors.

In comparison, if we were to extrapolate the 6-bit partial tag array used as a filter by Kim *et al.* [7] to an 8-way CMP with 16 banksets, it would require 1.5MB of storage (approximately 72 million transistors). The physical layout for both of these approaches can be organized as a roughly square memory structure. Our structure requires proportionally more space for the decoders and other RAM components, but we estimate they should not be more than 20% of the storage transistors. Thus, our design has much lower overhead, at least eight times smaller, than the existing work in this area.

Filter sizes and hash functions different from the ones we use in this paper may result in better performance or smaller filters—we leave a careful study of filter sizes and hash functions to future work.

4 Results

We now present the results of our initial exploration. We have implemented our Bloom filter smart search using the GEMS 1.2 [10] toolset. The *Ruby* cache model of GEMS includes the D-NUCA design we have described as our baseline. GEMS interfaces with the full-system functional simulator, Simics [9]. For evaluation, we used 12 benchmarks from the SPLASH-2 [15] parallel multithreaded program set. We start with a cold cache in order to take into account compulsory misses in the filters. All programs ran for at least one billion instructions, and at least 500 thousand L2 accesses.

Our initial implementation tracks the management of the Bloom filters and records their accuracy, but does not yet modify the search itself. Thus, the results we present here reflect the accuracy of the filters, and not the IPC or power improvements. Overall, we find that the filter accuracy is high enough to indicate that we have identified a promising technique for smart search. Our results merit further examination, including quantifying the cycles and power saved. We leave these evaluations to future work.

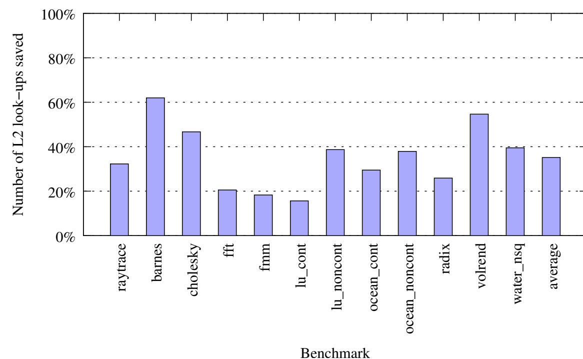

Figure 3. Percent of messages saved by our smart search per L2 access.

4.1 Messages Saved

The key metric for evaluating our smart search is how many messages it saves compared to the baseline search. A saved message is a block request sent to an L2 bankcluster by the baseline search, but filtered out by our search. The effect of fewer messages is decreased traffic on the L2 routing network, resulting in lower power and fewer contention cycles. Power is also saved because fewer bankclusters have to check their tag arrays.

Figure 3 shows the average messages saved per L2 access. The average savings across all benchmarks is 35%, and two are better than 50%.

4.2 Transistor Efficiency

We now evaluate our work in terms of its complexity effectiveness. To do so, we look at the number of messages saved on each L2 access per million transistors. We compare our filter with an idealized 6-bit partial tag array, much like the one used in prior work [7]. Each core has one tag array per bankset, just as we have one bloom filter per core per bankset. We do not model synchronization messages for the tag arrays, assuming that the arrays are perfectly synchronized with the corresponding bankset. Thus, the tag arrays in this test performs better than would a real implementation. As previously calculated, our filters are assumed to require 7.68 million transistors, and the partial tag arrays are assumed to require 72 million transistors.

Figure 4 shows the the results of this test. The number of messages saved for both structures is similar. The accuracy of the idealized partial tag array

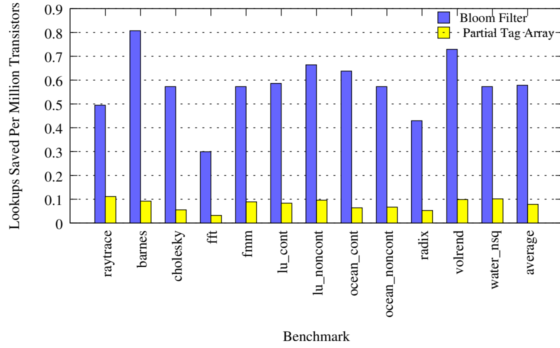

Figure 4. Average messages saved on each L2 access, per million transistors. We compare our filters with an idealized 6-bit partial tag array.

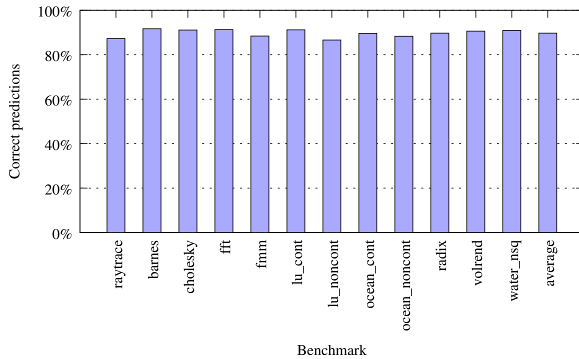

Figure 5. Overall filter accuracy.

is slightly higher—it saves on average 1.2 more messages per lookup. However, the Bloom filters, because of their much smaller size, make more efficient use of transistors. On average, they are able to save more than 6 times as many messages per million transistors.

4.3 Filter Accuracy

We now look at filter accuracy in finer detail, shown in Figure 5. This graph shows the number of true positives and true negatives over all filter lookups. As we can see, our filter accuracy is quite good overall.

The dominant cause of inaccuracy is false positives. This is due, in large part, to migration. When a cache line migrates, our filters learn the new location, but still remember the old location. We believe that the primary way to improve our filter accuracy will be to handle these migrations, removing line addresses from their old bankclusters.

To illustrate this point, we consider a modified version of D-NUCA that does not migrate blocks—it

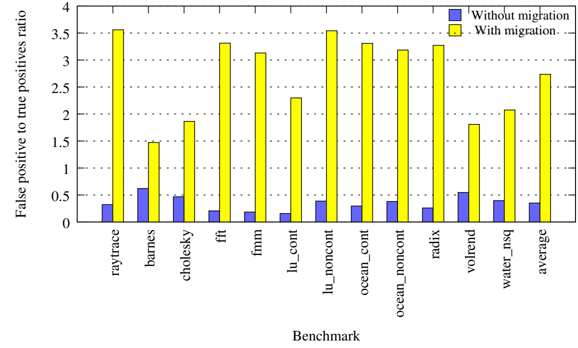

Figure 6. False positive to true positive ratio for D-NUCA caches with and without migration.

brings them into the cache according to D-NUCA polices, but does not move them thereafter. We do not claim that this cache policy is desirable, we use it only to isolate the effects of migration on our false positive rate.

The ratio of false positives to true positives with and without migration are shown in Figure 6. This ratio serves to isolate the percentage of false positives that result from migration, and we can see in this figure that the effect is significant. If we can, in future work, find a way to address migration, the accuracy of our filters will improve. Counting Bloom filters, which allow removal of set elements, may be of help, but it remains to be seen whether their higher storage requirements would be worth the benefit.

4.4 Per-L2 access Statistics

Finally, we examine the "per-L2 access" characteristics of the Bloom filters. We consider each L2 access (resulting from an L1 miss) as a whole. Thus, if *any* of a core's sixteen filters correctly predicts a hit in a particular bankcluster, we consider the whole access to have hit, regardless of the predictions from other filters. If there are no true positives, but one or more false positives, then the whole access is considered a false positive. If there are no positives returned, we classify the access as a true or false negative. Results are aggregated across all eight cores.

Figure 7 shows the Bloom filter accuracy for the SPLASH-2 benchmarks. This shows the percentage of L2 accesses for which our filters return the correct prediction (either a true positive or a true negative) on

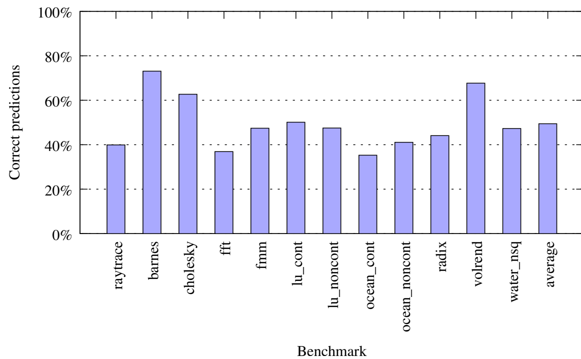

Figure 7. Filter accuracy per L2 access.

the first pass. We believe that the primary way to improve this accuracy will be to deal with migrations, as discussed in Section 4.3.

5 Conclusion

In this paper, we have explored the use of Bloom filters to create a smart search algorithm for CMPs with D-NUCA caches. Our results are very promising, showing that such filters can have very high accuracy, which results in a reduction of block requests. Our filters are complexity effective—they make efficient use of transistors, and do not make changes to the baseline coherency protocols.

Our initial explorations leave room for future work. Of most immediate importance, we will move on to quantify the cycle and power savings that result from our smart search. As shown in Section 4.3, if we can address the false positives that linger after block migration, very large improvements will likely result. Counting filters may help address this problem, but using them in this context is not straightforward.

Acknowledgements

We thank Liqun Cheng for helping us with Simics and benchmark suites. Virtutech AB for provided us with the Simics licenses necessary for our simulations. We also thank Wisconsin's Multifacet group, especially Bradford Beckmann and Mike Marty, for their help with GEMS.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of ISCA-27*, pages 248–259, June 2000.

[2] B. Beckmann and D. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of MICRO-37*, December 2004.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors, July 1970.

[4] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of ISCA-32*, June 2005.

[5] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, Vol.89, No.4, April 2001.

[6] J. Huh, C. Kim, H. Shafi , L. Zhang, D. Burger, and S. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of ICS-19*, June 2005.

[7] C. Kim, D. Burger, and S. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *Proceedings of ASPLOS-X*, October 2002.

[8] P. Kongetira. A 32-Way Multithreaded SPARC Processor. In *Proceedings of Hot Chips 16*, 2004. (`http://www.hotchips.org/archives/`).

[9] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[10] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[11] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2), March/April 2005.

[12] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz. The Implementation of a 2-Core Multi-Threaded Itanium Family Processor. *IEEE Journal of Solid-State Circuits*, 41(1), January 2006.

[13] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom fi ltering cache misses for accurate data speculation and prefetching. In *Proceedings of Int'l Conference on Supercomputing (ICS)*, pages 189–198, June 2002.

[14] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

[15] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA-22*, pages 24–36, June 1995.