Router Buffer Caching for Managing Shared Cache Blocks in Tiled Multi-Core Processors

Joe Augustine [‡], Raghavendra K [*], John Jose [⋆], Madhu Mutyam [‡]
[‡]Indian Institute of Technology Madras, Tamil Nadu, India
[*]Indian Institute of Technology Tirupati, Andhra Pradesh, India
[⋆]Indian Institute of Technology Guwahati, Assam, India
{joe,madhu}@cse.iitm.ac.in, raghavendra@iittp.ac.in, johnjose@iitg.ac.in

***Abstract*—Multiple cores in a tiled multi-core processor are connected using a network-on-chip mechanism. All these cores share the last-level cache (LLC). For large-sized LLCs, generally, non-uniform cache architecture design is considered, where the LLC is split into multiple slices. Accessing highly shared cache blocks from an LLC slice by several cores simultaneously results in congestion at the LLC, which in turn increases the access latency. To deal with this issue, we propose a congestion management technique in the LLC that equips the NoC router with small storage to keep a copy of heavily shared cache blocks. To identify highly shared cache blocks, we also propose a prediction classifier in the LLC controller. We implement our technique in Sniper, an architectural simulator for multi-core systems, and evaluate its effectiveness by running a set of parallel benchmarks. Our experimental results show that the proposed technique is effective in reducing the LLC access time.**

I. Introduction

Processor designers can embed hundreds of cores in modern tiled multi-core processor [1]–[3] as shown in Figure 1. As the number of cores increases, the underlying bus-based communication system faces scalability issues and architectural bottlenecks. For massively parallel applications with huge communication requirements, a bus-based communication system may not provide the required bandwidth. To resolve this problem, a scalable interconnection system called Network-on-Chip (NoC) [4]–[6] is used to communicate between different cores and the cache hierarchy. Modern tiled multi-core processors use non-uniform cache architecture (NUCA) [7] for cache block mapping to the LLC slices. Cache blocks are mapped to the LLC slices using a static non-uniform cache architecture (S-NUCA) or a dynamic non-uniform cache architecture (D-NUCA) [7], [8]. Conventional S-NUCA schemes use a few bits in the set index portion of the physical address to identify the LLC slice location for a given address. Hence a particular block is always statically mapped to an LLC slice called the *home node* for that block. Such mapping restrictions are not there in the D-NUCA that allows block relocation nearer to the processing cores at run-time. Accesses to cache blocks in the LLC slices that are physically closer to the requesting core take less time than accesses to cache blocks mapped to far away LLC slices. The access times are also dependent on the congestion level and the arbitration policy in the underlying NoC framework. Liu et al. [9] observed that even though

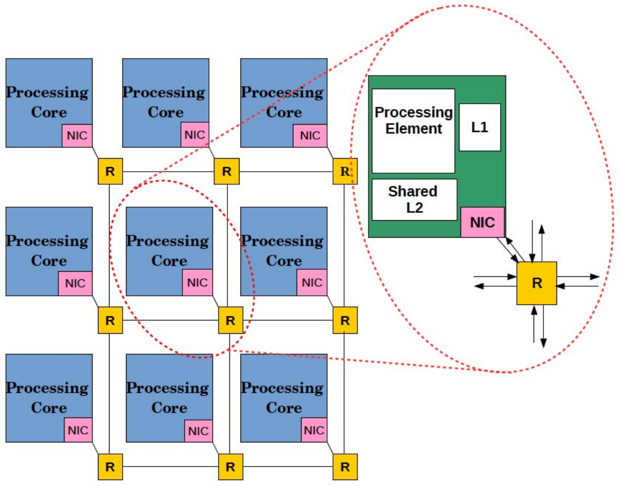

Fig. 1: A tiled multi-core processor with network-on-chip.

there exist very few shared data blocks in a multi-threaded application, these blocks are accessed frequently.

In a tiled multi-core processor connected via distributed communication mechanism like NoC, the cores cannot see other's read-write activity. Copies of cache blocks in different private L1 caches are kept coherent using a directory based cache coherency protocol [10], [11]. Accessing shared data involves a crucial number of coherence communication messages exchanged across the underlying NoC. The coherence protocol and the underlying NoC play a significant role in determining the latency of servicing an LLC request. When many cores request the same LLC data simultaneously, the number of read requests reaching the respective LLC slice will be significantly higher, leading to hotspot formation and subsequent congestion around the tile containing that LLC slice. Reducing the service time of such demand requests from the cores is of utmost importance in ensuring application-level performance. Lightweight directory-based coherence protocol implementations are considered in the past [10], [11]. Storing data and coherence information in underlying NoC can facilitate faster access to data [12]–[14]. To the best of our knowledge, there are no works to reduce congestion caused by

2576-6996/20/$31.00 ©2020 IEEE
DOI 10.1109/ICCD50377.2020.00050

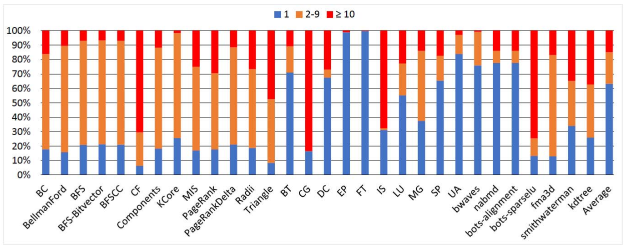

Fig. 2: Distribution of shared read data accesses to the LLC as a function of sharer-length. The classification is done at the cache block granularity.

multiple simultaneous requests for shared blocks at the LLC slices, which we will explore in this paper. Our technique intelligently identifies and makes copies of heavily shared cache blocks in the NoC router of the respective home node, which in turn reduces queueing delay in the LLC access thereby reducing the average access time.

We make the following contributions in this paper:

- We propose a congestion management technique in the LLC by keeping highly shared blocks in the network. We use small storage inside the network router to keep copies of these blocks.
- We propose a prediction classifier in the LLC controller to track and identify cache block sharing patterns so that the LLC controller can insert these cache blocks into the network storage.
- We use the reuse rate of the cache blocks in the network storage to track and control the pollution.
- Experimental results for a 64-core system demonstrate that our technique reduces the overall LLC access time on average by 16% and the LLC energy by 21% compared to a baseline system.

II. Motivation

We define *sharer-length* of a cache block as the number of cores accessing the cache block (at the LLC) before it is evicted from the LLC or before it is updated (through a write request). The higher the sharer-length, higher is the demand for the cache block in the LLC. Figure 2 plots the number of sharers to cache blocks in the LLC as a function of sharer-length for various applications taken from Ligra, SpecOMP and NPB benchmark suites [15]–[17]. For example, in application *BC*, out of the total read accesses, 18% (blue) is to the blocks having one sharer-length. Similarly, 66% (orange) of the read accesses is to the blocks having a sharer-length of 2 to 9, and the rest is to the blocks where the sharer-length is greater than 9. We can observe that almost half of the applications (first half of Figure 2) have 40% of the accesses to the shared blocks with sharer-length between 2 and 9. Some of them also have more than 30% of the accesses to the blocks with sharer-length more than 9. Retrieving data from the LLC includes queuing delay and access delay. So, the queuing delay increases for the higher sharer-length. Since every LLC access is due to an L1 cache miss, such an increase in queuing delay causes miss penalty increase as well, which may affect application performance. One of the potential solution to tackle this is to place heavily shared blocks in a place outside the LLC where the queuing delay is absent or minimal.

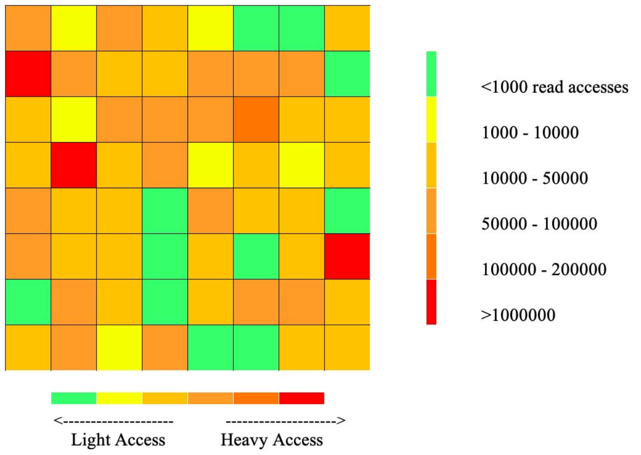

Fig. 3: Heatmap of read accesses by cores to 64 LLC slices.

Figure 3 shows the heatmap of *KCore* application taken from the Ligra benchmark suite. Various heatmap colors indicate the relative difference in the number of read accesses (during Region of Interest) across different tiles as mentioned in the right side legend. Similar heatmap graphs are obtained for around 50% applications that we studied. In many benchmark applications, we observe a good number of read accesses to a small set of LLC slices shown by red color in Figure 3. As shown in the legend, the red tiles get roughly ten times more read requests than orange tiles. Another interesting observation is that the accesses to these red tiles are for a very few shared cache blocks. So if these highly demanded cache blocks are kept in a place outside the LLC, the impact of such hotspots inside the LLC can be reduced.

Considering the above two scenarios, we explore the possibility of keeping high sharer cache blocks in convenient storage inside the NoC router.

III. RBC: Router Buffer Cache

We propose *router buffer cache* (RBC), wherein we keep a set of buffers inside the NoC router to store a copy of highly shared cache blocks. We also propose a prediction classifier in the LLC controller to track and identify cache block sharing patterns and frequency. This runtime classifier allows inserting those cache blocks that demonstrate high sharing at the LLC into the RBC. When a cache block gets evicted from the RBC, the LLC classifier gets the reuse rate of that block in the RBC for improving its prediction accuracy.

When multiple cores request the same cache block, all the requests get queued up at the home node. All these requests have to wait for the previous one to finish, which incurs queuing delay, directory lookup, and data access latency before they get dequeued. If the RBC keeps the blocks that are requested by multiple cores: 1) access to the same block will not incur data access latency from the LLC, and 2) each request takes less time to finish, which in turn reduces the queuing delay.

Figure 4 shows the protocol operation of the RBC design. Apart from the normal operation, the LLC controller and the NoC router with an RBC perform the tasks as explained in the subsections below.

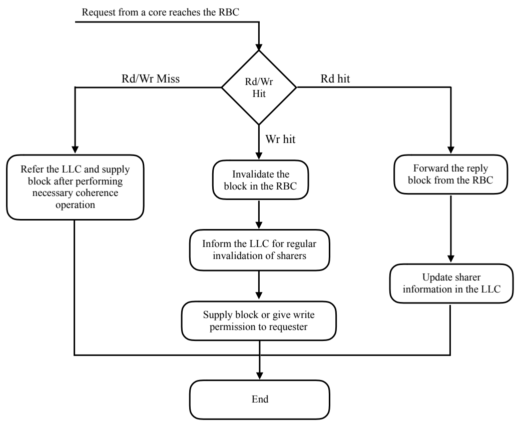
Request from a core reaches the RBC
Rd/Wr Miss
Rd/Wr Hit
Rd hit
Wr hit
Refer the LLC and supply block after performing necessary coherence operation
Invalidate the block in the RBC
Forward the reply block from the RBC
Inform the LLC for regular invalidation of sharers
Update sharer information in the LLC
Supply block or give write permission to requester
End

Fig. 4: RBC protocol operation.

A. *Read Requests*

When a read request reaches the home node router, it checks for the requested block in the RBC. If the block is found, it is sent to the requester as a reply packet consisting of multiple flits from the home node router. The router marks the request as serviced and inserts it into the LLC queue so that the LLC updates the sharer-information in the directory. Since this process is not on the read request critical path, the queuing and directory access delays do not affect the miss penalty associated with the cache miss request. The RBC keeps only those blocks that are in 'S' state. So, whenever there is a hit in the RBC, the replied block will also be kept in 'S' state in the requester L1 cache. The router also keeps track of the number of hits of each block in the RBC using saturating counters. This information is used to control pollution, which is described later in Section III-E.

On the other hand, if the RBC does not contain the block, the request is inserted into the home node queue. The LLC controller processes these requests like in a traditional cache coherent tiled multi-core processor system. However, here we explore the suitability of the cache block to be promoted to the RBC. The details of this process are discussed later in Section III-C. On an LLC miss, the cache block is brought in from the off-chip memory as usual.

B. *Write Requests*

When a write request (occurred due to a write miss in the L1 cache) reaches the home node router, it first looks up its RBC for the requested block. If it is a hit, the router invalidates that block in the RBC, and we insert the request into the home node queue. The home node controller sends invalidation

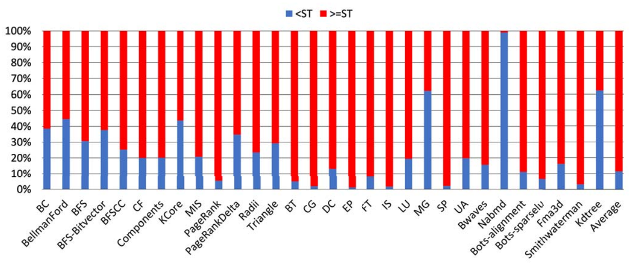

Fig. 5: Sharing pattern of blocks belonging to same page. ST is set to 5.

requests to all the sharers of that block. After receiving the acknowledgment from all the sharers, it sends the block with write permission on it to the requested core.

When it is an upgrade request that occurred due to a write operation on a block in 'S' state in the requester's L1 cache, the LLC will give the write permission without supplying the block. All other processes regarding invalidation of sharers (except the requester) are the same as those of a write request mentioned above. For a write or an update request that results in an RBC hit, before inserting it into the home node queue, we invalidate the corresponding blocks in the RBC. This ordering (RBC invalidate followed by enqueueing in the home node) must be strictly enforced to avoid future immediate read requests operating on stale data due to an RBC hit.

C. *Block Insertion Strategy*

In our experiment, we consider a block as a *high-sharer block* if the number of sharers to the block exceeds a sharer-threshold (ST). We identify the page numbers of such blocks and add them to the history table. The history table can store up to 4-page numbers, and it uses the LRU replacement policy to select a victim. This history table identifies the blocks to be placed in the RBC. Ideally, the RBC should contain blocks that have maximum sharers. These blocks should remain in the RBC to get the maximum number of hits before they are evicted out. Based on our observation, we identify that if a page contains a high-sharer block, there is a very high probability that other blocks in the page also are high-sharer blocks. Figure 5 classifies the sharing pattern of shared blocks belonging to the same page with at least one block having sharers greater than ST. Red shows the fraction of such blocks that have sharers greater than ST, and blue shows the fraction of blocks that have sharers less than ST. To summarise the findings given in Figure 5 if there exists at least one high-sharer block in a page, on an average 90% of the cache blocks in the same page are also high-sharer blocks.

When the LLC controller gets a read request to a block that is in Exclusive (E) state, it indicates that this block is getting a new sharer (the state moves from E to S). A reference to the history table will reveal whether any other block in the same page as that of the requested block has many sharers. If we get a page number hit in the history table, we conclude that the requested block may also have a high number of sharers in the future. Observations made from Figure 5 supplement this conclusion. Hence, during history table hits, the requested block is copied to the RBC such that future read requests can be serviced from there. If we could not find a match in the history table, the block is sent to the requester without inserting it into the RBC. Our experimental results (to be discussed later) also reconfirm the fact that such promotion of blocks from the LLC to the RBC is effective in handling future read requests.

In most of the applications, the number of shared blocks is low, and mostly they belong to the same page. For all the benchmarks that we tested, a history table with four entries is enough to store the page numbers of highly shared blocks.

D. Replacement Policy and Eviction Strategy

When the RBC is full, an existing block needs to be evicted to make space for the new block. A replacement policy is used to select this victim block. This replacement policy plays a crucial role in the performance of the RBC. The RBC should evict a block that has the least chance of getting a hit in the future. In our RBC design, we use the LRU replacement policy to find a victim block. A block becomes hotspot in the LLC when multiple cores request that block within a small time window. These blocks have to be kept in the RBC as long as there are frequent read accesses from the cores. When the number of read accesses on a block in the RBC reduces significantly within a time window, the block will become a potential candidate for victim block. In the LRU replacement policy, when a block is not getting hits, it is pushed to the LRU position. These blocks should get evicted out soon. When we evict a block from the LLC, the LLC controller sends an invalidation to all its sharers as well as to the RBC to maintain coherency.

The RBC communicates only with its local LLC controller. The LLC controller promotes blocks to the RBC and invalidates blocks in the RBC. The router also communicates hits of the evicted blocks back to the LLC controller for pollution control. This communication between the RBC and the LLC controller is through injection and ejection channels of the router. Since this does not generate any new packets into the network, there is no additional communication overhead.

E. RBC Pollution Control

Insertion of a new block to the RBC can cause pollution as cores may request evicted blocks. When pollution increases, we incur the wastage of energy without any performance benefits. Our design also takes care of RBC pollution control.

When a block is evicted from the RBC, the router communicates the number of hits for that particular block to the home node controller. We make use of a two-bit saturating counter per RBC entry to track the number of read hits. Upon eviction, if this hit counter value associated with victim block is less than 3, it indicates that this block was not a suitable candidate to be kept in the RBC. Alternatively, it can be due to a sharer footprint that is larger than RBC size. This can help in determining whether the newly created blocks are causing pollution in the RBC.

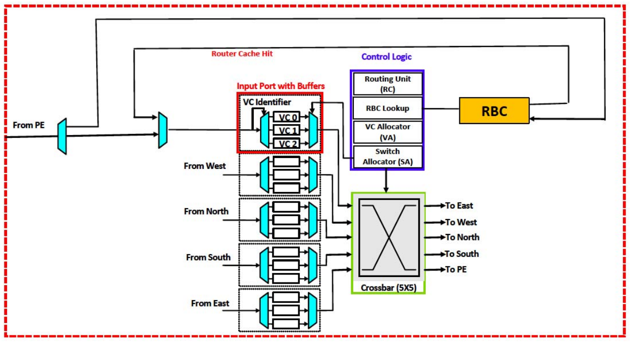

Fig. 6: Proposed router microarchitecture.

To reduce RBC pollution, we have to control the blocks that are getting promoted to the RBC. Since the blocks created in the RBC is dependent on the history table, removing entries from the table will help in reducing the blocks that get hit in the table. When the LLC controller detects pollution, it can either remove a couple of the *most recently used* (MRU) entries from the table or it can clean the entire table so that no new blocks will get created in the RBC for some time.

F. Page Partitioning

The proposed classifier stores the page numbers of the high-sharer blocks in the history table that helps in identifying which blocks to move from the LLC to the RBC. A single block from a page decides whether the rest of the blocks in that page are eligible for getting promoted to the RBC. In some applications, this leads to pollution that can be controlled by the pollution control logic discussed in Section III-E. However, we observe that pollution re-occurs after some time as the history table gets populated. We address this issue by restricting the number of blocks affected by a high-sharer block in a page. Rather than considering all the shared blocks in a page for RBC allocation, we divide a page into four zones. If a block from a given zone inside a page becomes a high-sharer block, only the blocks belonging to that zone are eligible to be promoted to the RBC. We observe that pollution reduces significantly by this zonal page partitioning.

G. Router Microarchitecture

We consider a two-stage router pipeline design as the baseline system [18]. Figure 6 shows the proposed router design. The router contains an 8-entry RBC. When we promote a cache block from the LLC to the RBC, we divide it into multiple flits and store in the RBC so that when there is a hit in the RBC, the router can directly send the flits to the local input port without having to convert the cache block into flits.

Our proposed router has two pipeline stages: route computation (RC) and RBC lookup in stage 1 and speculative virtual channel (VC) allocation and switch allocation in the other. The routing unit computes the output channel for the head flit. In the VC allocation stage, the router allocates VC

Architectural Parameter	Value
Cores	64 cores @ 2.66GHz, Out-of-order, superscalar
Processor Word Size	64 bits
Page Size	4KB
Memory Subsystem	
L1-I Cache per core	32 KB, 4-way, 64B block, 2 cycle latency
L1-D Cache per core	32 KB, 4-way, 64B block, 2 cycle latency
LLC per core	512 KB, 16-way, 64B block, 2 cycle tag latency, 8 cycle data latency
Directory Protocol	Invalidation-based MESI [20]
Main Memory	16 controllers
NoC	
Hop Latency	3 cycles (2-router, 1-link)
Routing	X-Y routing
Topology	8x8 Mesh
Flit Size	8 B
RBC	
Size	576 B (8 blocks; 9 flits each) , fully associative

TABLE I: Architectural parameters used for evaluation.

for the head flit in the downstream router. When multiple flits are competing for the same output port, the switch allocation stage decides the winner. The winning flit moves through the crossbar during the switch traversal stage and reaches the output port to perform the link traversal. We added the RBC lookup in the router pipeline. The RBC lookup is possible only if the request flit completes the route computation operation, which also ensures that the RBC is checked for those request flits that has reached the home node.

IV. Evaluation Methodology

We evaluate our proposed design on a 64-core tiled multi-core processor system. Table I shows the default architectural parameters used in our evaluations. All experiments are performed using the Sniper multi-core simulator [19]. We model all the mechanisms and protocol overheads discussed in Section III. We consider one RBC per router that can accommodate eight cache blocks. We store each cache block as a sequence of 9 flits (1 head and eight body flits). In addition to the fixed 2-cycle latency in the NoC routers, network link contention delays are also modeled to get performance statistics.

We evaluate our proposed design using standard multi-threaded benchmark from Ligra [15], NPB [17] and SPEC OMP 2012 [16]. We run these applications to completion and take their average for ten runs during Region of Interest (RoI) for reporting statistics.

V. Experimental Analysis

We analyse the response of each L1 cache miss and conduct an in-depth analysis of the fraction of requests responded from the RBC and the LLC.

A. Impact on Serving the L1 Cache Misses

Figure 7 plots the percentage of RBC hits, LLC hits, and LLC misses (DRAM hits) from the total number of L1 cache misses. Across the majority of benchmarks, we see

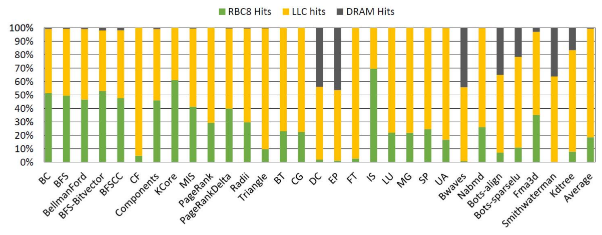

Fig. 7: L1 cache miss type breakdown.

a significant number of RBC hits, which indicates that the proposed scheme of promoting shared cache blocks from the LLC to the RBC is giving good rewards. On average, 20% of read requests are serviced by the RBC.

We also observe that the RBC hit ratio is very low for a few benchmarks. Benchmarks *EP*, *FT*, *Bwaves*, and *Smith-waterman* have very few shared cache blocks, and private cache blocks dominate in these benchmarks. Since the RBC technique works by high-sharer blocks, the above-mentioned benchmarks are expected to show poor RBC hits as shown in Figure 7. RBC8 represents our proposed technique with an RBC capacity of 8 cache blocks. Benchmarks *CF* and *DC* have a significant number of cache blocks that are eligible to be kept in the RBC. But due to the limited size of the RBC, every such block cannot be accommodated in it. So there is a significant reduction in the RBC hit ratio for these two benchmarks. We analyse the high RBC hit rate (around 70%) of *IS* benchmark and find that it has a very few high-sharer blocks. So *IS* benchmark can accommodate most of these high shared-length blocks in the RBC.

B. Impact on the LLC Access Time

We consider the RBC that can accommodate eight cache blocks. However, the RBC size of 8 may not be the best design choice for all the benchmarks. The number of high-sharer blocks will vary across benchmarks, and hence the demand for the RBC size also varies. Figure 8 plots a comparative study of the LLC access time split-up for various RBC sizes over the baseline tiled multi-core processor. RBC8, RBC16, and RBC32 represent our proposed technique with an RBC capacity of 8, 16, and 32 cache blocks, respectively. The base bar represents a conventional tiled multi-core processor without an RBC. The LLC access time consists of queuing delay, tag access delay, data access delay, and bus delay. Queuing delay is the time a request waits in the LLC queue to access the requested block. Tag access delay is the time taken for the LLC controller to access the tag array. Data access delay is the time taken by the LLC controller to access the block from LLC. Bus delay is the time taken by the bus to transfer the requested block from the LLC to the LLC controller.

Across all the benchmarks in Figure 8, we observe that the base design (i.e., without RBC) has a high fraction of the LLC access time spent on queuing delay. This is because every L1 miss request reaches the LLC controller and waits

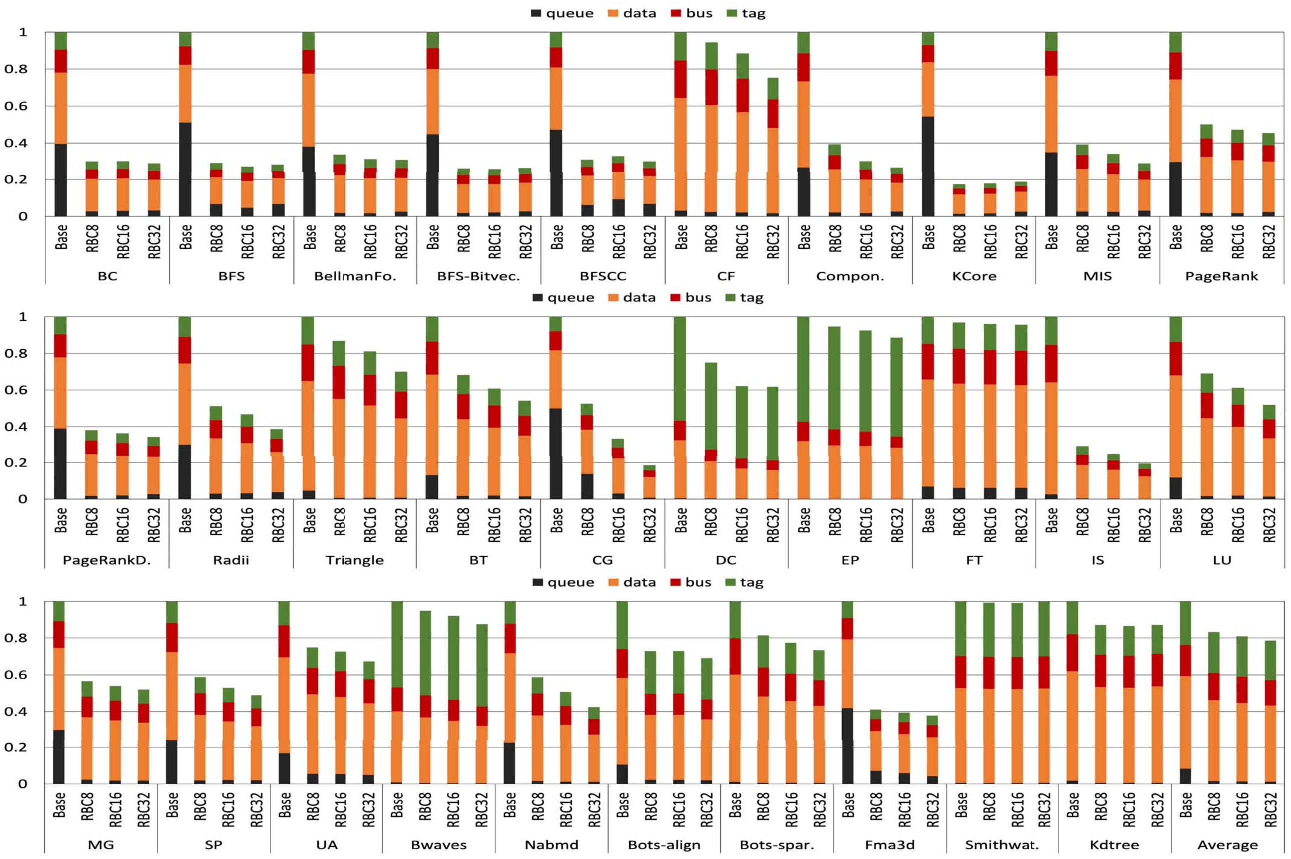

Fig. 8: LLC access time breakdown. Results are normalized to that of the base tiled multi-core processor.

for its turn until all its predecessors are serviced. The RBC design significantly reduces this queuing delay as the router services most of the requests by eliminating the LLC access in its critical path. Even though a block gets serviced from the RBC, we add the request to the LLC queue for updating sharer information in the directory structure. However, the service time of such request is very short as updating sharer information in directory takes significantly less time compared to data access from the LLC. So, the subsequent requests reaching the LLC queue experience less queuing delay.

In general, if an application has a large number of high-sharer blocks, a large RBC is required. A large RBC keeps a block in it long enough to service all accesses to it before it gets evicted by a new block, which is evident from the benchmark *CF* that reduces the access time of shared blocks with an increase in the RBC size. Most of the blocks of *CF* are high-sharer blocks. RBC32 performs better than RBC16 and RBC8 because of this reason. The behavior of benchmarks *Triangle* and *LU* is also similar due to the same reason.

If an application has a few high-sharer blocks, a small RBC is sufficient to get a decent reduction in the access time. We observe that many benchmarks exhibit the behavior of multiple read requests (maybe from different cores), followed by a write request from a core. Under this circumstance, the block gets invalidated (due to the coherence operation) in the RBC before it even gets evicted. The RBC gets populated again upon subsequent read requests to the invalidated block. This invalidation operation on a cache block cannot be eliminated even by using a larger RBC. Benchmarks like *KCore*, *Fma3d* and *PageRank* show more or less the same LLC access time under varying RBC sizes. A significant number of benchmarks under study show access patterns that have blocks with sharer-length less than 9. We empirically conclude that an RBC of size 8 is an optimal choice.

Figure 9 plots the reduction in the average memory access time (AMAT) of various benchmarks using the RBC8 design. For benchmarks such as BC, BFS, CG, etc., that achieve a good reduction in the LLC access time, there is an apparent reduction in the AMAT as compared to the baseline. Overall, there is a 5% reduction in the AMAT. We can observe that reduction in the AMAT is not proportionately translated to a decrease in execution time due to overlap of memory.

C. Impact on the LLC Hotspot

The primary motivation for proposing the RBC design is to eliminate hotspots in the LLC slices occurring due to simultaneous memory access requests. When we observe the number of LLC read requests across various benchmarks, we find that for *KCore* benchmark, the non-uniformity across tiles is high. Figure 10 shows the heat map of *Kcore* benchmark on a baseline tiled multi-core processor and a tiled multi-core processor with the RBC design. We can see that most of the requests are towards three of the LLC slices (tiles, which are red in Figure 3). Almost all of these requests are towards high-sharer blocks and a high number of writes. Because of this access pattern, the RBC design gives a good performance improvement. Figure 3 shows a significant reduction in the number of accesses to the LLC slices that are hotspots in the baseline. There is around 60% reduction in the number of LLC accesses with the proposed technique. Queuing time at the LLC is reduced by 65%. Overall there is an 80% reduction in LLC access time. We observe similar trends in most of the multi-threaded applications. Heat map analysis of various benchmarks concludes our findings that the RBC design can effectively reduce hotspot formation in the LLC slices.

VI. Timing and Area Overhead Analysis

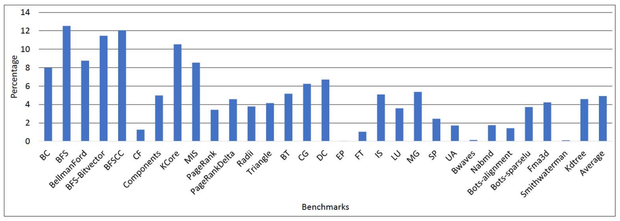

Fig. 9: Reduction in the average memory access time.

To compare hardware timing and area overhead of the new router with respect to existing router designs, we have implemented them in Verilog HDL. First, we synthesized both the designs using Xilinx Vivado 2016.2 IDE followed by post synthesis simulation. The simulation was done using custom designed testbenches. Futhermore, the RBC block was seperately synthesized and programmed into Kintex 7 series FPGA board to verify its intended functionality as buffer cache. Next to design a real concrete model, both the baseline and the proposed designs were synthesized using Synopsys design compiler (2017.09-SP4) with 45nm CMOS library to determine ASIC level critical path and area overhead analysis.

The baseline design consists of a 2-stage pipeline for the NoC router. The stage-1 performs route computation and the stage-2 performs VC and switch allocation. In addition to the above two stages, for our design an RBC lookup is necessary. The critical path delay of stage-1 and stage-2 is 0.69 ns and 0.99 ns, respectively. The maximum path delay of RBC lookup module is 0.23 ns. We observed that previous works [18] report similar critical path for stage-1 and stage-2 of our baseline design. It is interesting to observe that the total time to perform route computation (stage-1 with 0.69 ns), and RBC lookup (0.23 ns) is 0.92 ns which is even smaller than the critical path of stage-2 (0.99 ns). We find that the additional module that is added in stage-1 is not affecting the critical path because the 2-stage pipeline router determines the critical path of the router pipeline. Hence, the proposed lightweight RBC

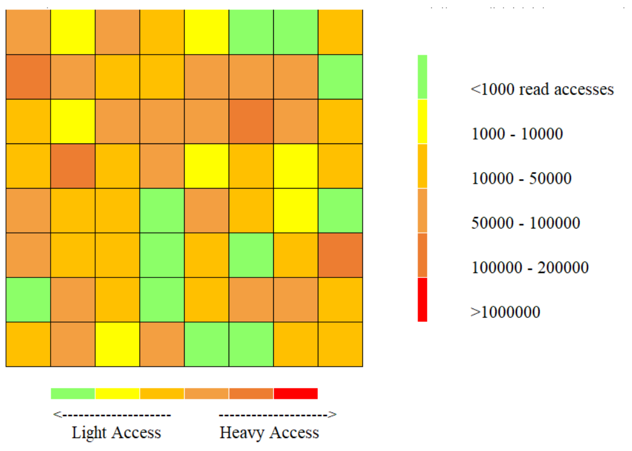

Fig. 10: Heat map of KCore benchmark application.

lookup module that is placed after the routing function module will operate well within the time slack available between the routing function latency and the router cycle time.

In regards to area overhead, the RBC buffers and history table add Non-combinational area overhead of 0.12%. Similarly, the RBC controller logic adds a combinational area overhead of about 0.29% compared to the baseline L2 cache tile. Thus, we expect that the total area overhead is about 0.41% of a typical L2 cache tile.

VII. Related Work

Eisley et al. [13] propose to implement the cache coherence protocol within the network by incorporating directories inside the network routers. In this protocol, a cache fill from the home node stores the coherence information in each router along the routing path. This enables the router to redirect the future L1 misses from other cores to nearest sharer. Aditya et al. [14] build upon the above work [13] by adding a small data store to the network that stores data being transported along with cache coherence protocol inside the router. This enables the router to service requests for data that they hold directly without going to the home node. This technique stores all the shared data along the intermediate routers which can cause pollution in a small sized cache. Similarly, Jinglei et al. [21] propose a network victim cache architecture that removes the directory from the shared LLC and keeps the sharer-information of blocks recently accessed by the L1 cache in the Network Interface Component (NIC). The space saved is used as a victim cache. Jinglei et al. [12] also propose three network caching designs: Network Directory Cache (NDC) design where the directory cache is integrated with the NIC to store the sharer-information of recently accessed shared LLC blocks; Network Shared Cache (NSC) design where a data cache is integrated into the NIC along with the NDC to reduce the L1 access latency; Network Victim Cache (NVC) design where victim cache is integrated into the NIC to reduce the number of accesses to the home node.

All the above related works that use in-network cache coherence [13] propose storing coherence information in the network. The main idea behind these works is to place the tag and/or shared data in the routers on the response path. However, there are two major caveats in these designs with respect to conventional directory-based protocol. First, replication of the tag and data blocks in the routers reduces the effective caching space available inside the network. Next, the invalidation traffic generated is high due to replacement of the same coherence information stored in multiple routers. Furthermore, the pressure on the neighbouring private L1 caches will also be high because the requesting core now reads the cache block from the closest sharer rather than the home node. Hence, manufacturers prefer directory based protocol over network coherence protocol to achieve cache coherence [22]–[24]. Our work is based on directory-based protocol that is an industry standard. We make use of the network storage to reduce the congestion in LLC slices without shifting the directory to the network (or bringing data close to the requester).

VIII. Conclusion

As the last level cache (LLC) is shared and distributed in tiled multi-core processors, accessing shared cache blocks from different cores can become a bottleneck. To overcome this issue, we proposed an intelligent caching mechanism, namely, RBC, in the NoC routers, where we stored highly shared cache blocks from the LLC. To identify highly shared cache blocks, we proposed a prediction classifier in the LLC controller. We also suggested a pollution control mechanism for RBC. We implemented our proposed technique using a multi-core simulator. We evaluated our technique's effectiveness by running a set of parallel benchmarks and provided a detailed experimental analysis. Experimental results showed that RBC with just eight entries reduces overall LLC access latency by 16% when compared to a system without RBC. We also showed that our technique reduces hotspots at the LLC slices. We conclude that our technique is instrumental in dealing with shared cache blocks in tiled multi-core processors.

References

[1] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, "Multi-Core Cache Hierarchies," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–153, 2011.

[2] M. Zhang and K. Asanovic, "Victim replication: Maximizing Capacity While Hiding Wire Delay in Tiled Chip Multiprocessors," in *32nd International Symposium on Computer Architecture*, June 2005, pp. 336–345.

[3] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya, "Integration Challenges and Tradeoffs for Terascale Architectures," *Intel Technology Journal*, vol. 11, no. 3, pp. 173–184, Aug 2007.

[4] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[5] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*, 1st ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

[6] M. Galles, "Spider: A High-Speed Network Interconnect," *IEEE Micro*, vol. 17, no. 1, pp. 34–39, Jan. 1997.

[7] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 211–222, Oct. 2002.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 184–195, Jun. 2009.

[9] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, "Enhancing L2 Organization for CMPs with a Center Cell," in *20th International Conference on Parallel and Distributed Processing*, Washington, DC, USA, 2006, pp. 32–32.

[10] C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System," in *National Computer Conference and Exposition*, New York, NY, USA, 1976, pp. 749–753.

[11] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," in *15th Annual International Symposium on Computer Architecture. Conference Proceedings*, May 1988, pp. 280–289.

[12] J. Wang, Y. Xue, H. Wang, and D. Wang, "Network caching for Chip Multiprocessors," in *IEEE 28th International Performance Computing and Communications Conference*, Dec 2009, pp. 341–348.

[13] N. Eisley, L. Peh, and L. Shang, "In-Network Cache Coherence," in *39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006, pp. 321–332.

[14] A. Yanamandra, M. J. Irwin, V. Narayanan, M. Kandemir, and S. H. Narayanan, "In-Network Caching for Chip Multiprocessors," in *4th International Conference on High Performance Embedded Architectures and Compilers*, Berlin, Heidelberg, 2009, pp. 373–388.

[15] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Feb. 2013.

[16] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in *International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, Berlin, Heidelberg, 2001, pp. 1–10.

[17] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, Sep. 1991.

[18] C. Nicopoulos, S. Srinivasan, A. Yanamandra, D. Park, V. Narayanan, C. R. Das, and M. J. Irwin, "On the Effects of Process Variation in Network-on-Chip Architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 3, pp. 240–254, July 2010.

[19] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011, pp. 52:1–52:12.

[20] G. Kurian, S. Devadas, and O. Khan, "Locality-aware data replication in the last-level cache," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 1–12.

[21] J. Wang, Y. Xue, H. Wang, and D. Wang, "Network Victim Cache: Leveraging Network-on-Chip for Managing Shared Caches in Chip Multiprocessors," in *4th International Conference on Embedded and Multimedia Computing*, Dec 2009, pp. 1–5.

[22] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, "Cache coherence protocol and memory performance of the intel haswell-ep architecture," in *2015 44th International Conference on Parallel Processing*, Sep. 2015, pp. 739–748.

[23] L. Hung, W. Starke, J. Fields, F. O'Connell, D. Nguyen, B. Ronchetti, W. Sauer, E. Schwarz, and M. Vaden, "Ibm power6 microarchitecture," *IBM Journal of Research and Development*, vol. 51, pp. 639–662, 11 2007.

[24] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, March 2010.