Exploiting On-Chip Routers to Store Dirty Cache Blocks in Tiled Chip Multi-Processors

Abhijit Das*, Abhishek Kumar*, John Jose* and Maurizio Palesi†
*Dept. of Computer Science and Engineering, Indian Institute of Technology Guwahati, India
†Dept. of Electrical, Electronic and Computer Engineering, University of Catania, Italy
{abhijit.das, abhishek18a, johnjose}@iitg.ac.in, maurizio.palesi@dieei.unict.it

***Abstract*—To meet the worst-case performance requirements, on-chip routers in Tiled Chip Multi-Processors (TCMPs) are provisioned with input port buffers. However, in real applications, the average buffer utilisation in routers is significantly low except during peak network congestion. In this work, we exploit the idle buffers in Network-on-Chip (NoC) routers to store dirty cache blocks evicted from the L1 cache. Future requests for such recently evicted cache blocks are serviced from the router buffer, thereby significantly reducing the cache miss latency. We propose two variations on how long we can store these evicted cache blocks on NoC router buffers; one up to a time threshold and another up to the arrival of a demand from an L2 cache bank. We make architectural modifications on the buffer management circuitry of NoC routers to make sure that the evicted, dirty cache blocks are stored in the local routers for the longest duration possible to facilitate local reply. Experimental results show that our proposed modifications achieve a maximum system speedup of up to 15% and an average system speedup of 12%.**

***Index Terms*—Miss latency, Cache coherence, Virtual channel**

I. Introduction

Tiled Chip Multi-Processors (TCMPs) provide the architectural backbone for smart IoT devices and high-end embedded systems. TCMPs are used in the simplest of handheld devices like mobile phones to the most complex data centre servers. One of the most fundamental challenges in designing these systems is to employ policies for efficient utilisation of shared resources. As TCMPs scale to hundreds of cores, one of the shared resources, Network-on-Chip (NoC) plays a vital role in the access latency of memory requests. Past investigations show that NoC resource utilisation is very low for standard multi-processor workloads, with an average injection rate of around 5% [1][2][3]. This under-utilisation of NoC resources and its subsequent utility for other on-chip functions is a promising area of research to explore.

Current NoC based TCMPs like Intel Xeon Phi Processors [4] have small private, write-back L1 caches and shared distributed L2 caches. When an L1 cache miss evicts a dirty (modified) cache block to give space for an incoming requested block, the dirty cache block is sent to the corresponding L2 cache bank for write-back over the NoC. The L1 to L2 cache communication is done through the underlying NoC using wormhole switching. This packet based communication in TCMPs usually traverses through multiple NoC routers. These routers have buffers to store transit packets during their routing and arbitration decisions. Finally, these packets upon reaching the destination tile update the corresponding L2 cache block. However, for applications that show temporal locality of reference, the same cache block which is just evicted and written back to the L2 cache bank may be requested again. In this case, the updated cache block needs to be brought back again, all the way from L2 to L1 cache. Increasing the L1 cache size might solve the problem up to an extent, but it is not cost-effective as large L1 cache may impact the L1 cache hit time and hence affect the instruction pipeline frequency.

Modern TCMPs employ input buffered NoC routers for scalable on-chip bandwidth [5][6]. Our work is also motivated from this fact that we have input buffered routers in commercial NoC based TCMPs [4]. Our experimental results show that, upon running real applications, the average buffer utilisation of routers is generally very low except during peak network congestion [7]. In this work, we store evicted, dirty cache blocks from a tile on empty buffers of the local NoC router. While a write-back cache block is stored in the local router, future requests for the same block can be immediately serviced from the router buffer itself rather than fetching from the distant L2 cache bank. This will significantly decrease the cache miss latency and hence improve overall system performance. We make the following major contributions:

1) We model an NoC architecture that identifies evicted, dirty cache blocks passing over the NoC and stores them in empty buffers of local routers. Subsequent re-referencing to such evicted cache blocks are serviced from the local router buffers to reduce the miss latency.
2) We propose architectural modifications in router logic to forward the stored, evicted dirty cache blocks from an NoC router to L2 cache bank at regular time intervals. This technique is known as Time-Triggered Block Forward (TT-BF).
3) We also propose architectural modifications in router logic to forward the stored, evicted dirty cache blocks from an NoC router to L2 cache bank when there is a request message for the stored data. This technique is known as Message-Triggered Block Forward (MT-BF).

II. Related Work and Motivation

Prior research efforts explored the possibilities of efficient on-chip resource utilisation in different capacities. Initial works focused on implementing in-network cache and coherence [8][9][10][11]. Then came the era of power-aware

2159-3477/20/$31.00 ©2020 IEEE
DOI 10.1109/ISVLSI49217.2020.00036

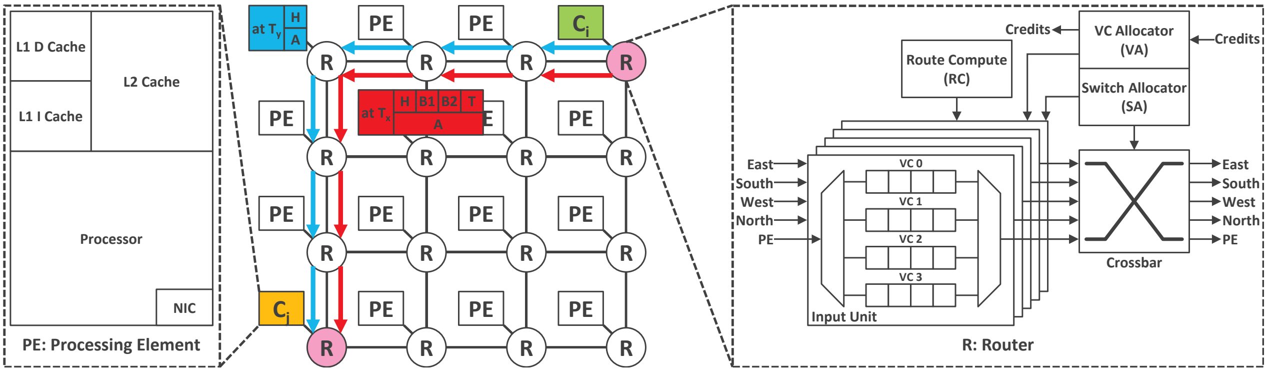

Figure 1: Conceptual view of a 16-core NoC based TCMP. A packet in NoC is divided into multiple smaller units called flits. A request packet consists of a single head flit (H), whereas a reply packet consists of a head flit followed by multiple body flits and ended by a tail flit (H, B1, B2, T). Flits of a particular packet always travel in order.

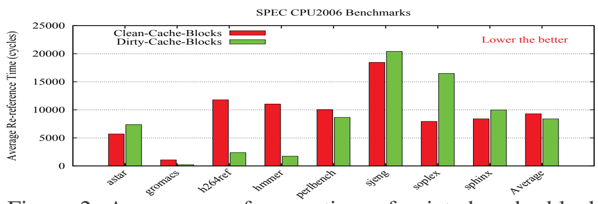

Figure 2: Average re-reference time of evicted cache blocks

NoC designs including bufferless and minimally buffered routers [12][13][14][15]. Now, the trend is more towards data and computation aware NoC designs [16][17][18]. Nevertheless, router buffers in modern TCMPs are underutilised [7], so can we do something with them (**Q1**)?

As shown in Figure 1, at time T_x, core C_i sends an NoC packet (evicted cache block) with address A to core C_j (shown in red). However, at time T_y, core C_i sends another request packet for the same address A to core C_j (shown in blue) where $T_x < T_y$ but are fairly closer. This is a case of re-referencing a recently evicted, dirty cache block. Figure 2 shows the average re-reference time of evicted cache blocks (both clean and dirty) for different multiprogrammed benchmarks (SPEC CPU2006) in a 64-core, Out-of-Order (OoO), 1GHz NoC based TCMP. The *red* bars show the average re-reference time of only clean cache blocks, and the *green* bars show the average re-reference time of only dirty cache blocks. For example, upon running the benchmark *astar*, an evicted clean cache block and evicted dirty cache block are re-referenced within an average time of 5700 cycles (red) and 7300 cycles (green), respectively. Across all benchmarks, on average, within a small interval of around 9000 cycles, an evicted cache block is re-referenced again.

Clean cache blocks upon eviction are overwritten, whereas evicted dirty cache blocks are written back to the corresponding L2 cache bank. The whole process of eviction, write-back and bringing back of the same cache block on re-reference has a significant impact on cache miss latency. Can this re-reference latency be reduced (**Q2**)? We have two questions, Q1 and Q2 to answer and if we carefully merge them, they solve each others problem. When the same cache block is re-referenced by the same core in near the future, we arrange for a direct reply of the stored block from the local router.

III. Proposed Architecture

In this section, we present our proposed architecture which is aimed towards homogeneous NoC based TCMPs like Intel Xeon Phi Processors [4]. We analyse how we can exploit the empty buffers of NoC routers to store evicted, dirty cache blocks. We show that the future data requests on the same stored cache blocks can be locally replied by maintaining cache coherence. Finally, we explain about forwarding the stored cache blocks using two different optimisations.

A. Storing Evicted, Dirty Cache Blocks in Routers

When a memory request from the processor results in an L1 cache miss, the requested block is brought from corresponding L2 cache bank as packets through the underlying NoC. The victim block is evicted from the cache to make room for the incoming requested block. This eviction of the valid cache block is intimated to the corresponding L2 cache bank to maintain coherency. The coherence update is also communicated through the NoC. Depending on the location of corresponding L2 cache bank, these packets traverse through multiple routers to finally reach their respective destinations.

In MOESI coherency protocol, a valid cache block can be either in *shared*, *exclusive*, *owned* or *modified* state. The actions performed on a block eviction from an L1 cache of conventional TCMP is briefed in Table 1; except the last row shown in *red*. Control and data packets are sent over NoC from L1 to L2 to communicate the eviction. These packets first reach the local router that is directly connected to the tile, get stored in the available buffers of virtual channels (VCs) and take part in routing and arbitration decisions.

As shown in Figure 3, we make use of the empty buffers of VCs in the local router to store evicted, dirty cache blocks for as long as possible. L1 cache sends dirty cache blocks for write-back to L2 cache after getting an acknowledgement as given in Table 1. We identify dirty data packets that are

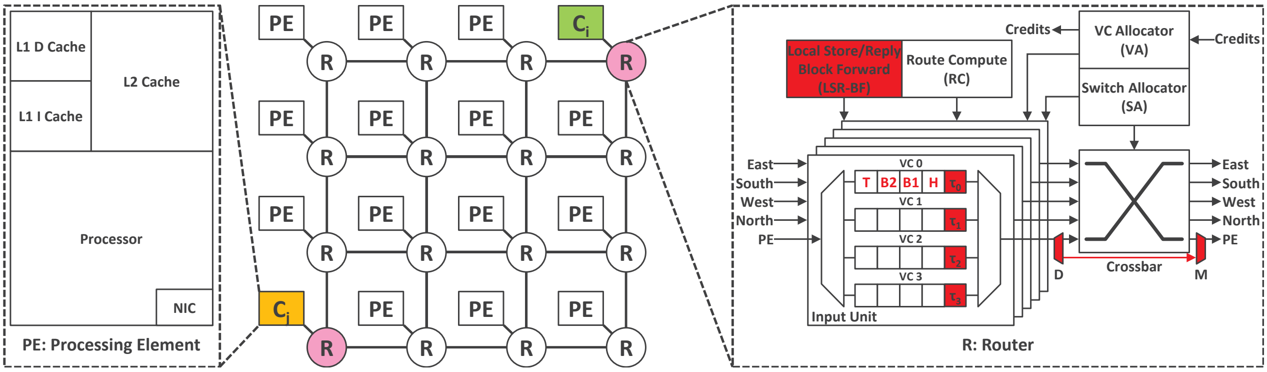

Figure 3: Conceptual view of our proposed optimisation with modified router microarchitecture. All the additional units in the router are shown in red. The local input VC buffer shown with red flits (H, B1, B2 and T) signifies the storage of an evicted, dirty cache block. If the same block is re-referenced in near future while it is stored, we can generate a local reply.

Table 1: Actions on L1 cache block eviction where, S: shared, E: exclusive, O: owned, and M: modified.

State	Action at L1	Packet	Action at NoC	Action at L2
S	Send invalidation of cache block to L2	Control	Store in local VC for routing and arbitration	Remove entry from sharer list
E	Send invalidation of cache block to L2	Control	Store in local VC for routing and arbitration	Remove entry from owner list
O & M	Ask permission to send cache block to L2	Control	Store in local VC for routing and arbitration	Send acknowledgement to receive dirty cache block
O & M	Send dirty cache block to L2	Data	Store in local VC for routing and arbitration	Write cache block and remove entry from owner list
O & M	*Send dirty cache block to L2*	*Data*	*Store in local VC; routing but no arbitration*	*Wait for the dirty cache block; unaware of local store*

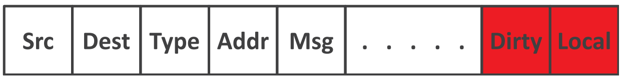

Figure 4: Modified packet header

moving from L1 to L2 cache. Such packets are marked with an extra bit (*Dirty*) in their header as shown in Figure 4. When packets enter the local router and get stored in the buffers of VC for routing and arbitration, *Dirty* bit is checked by the additional *Local Store/Reply and Block Forward (LSR-BF)* unit as shown in Figure 3. LSR-BF unit works in parallel with *Route Compute (RC)* unit. So, when a check is performed to identify a dirty packet, route computation for the same packet is also being performed. Now, if a dirty packet is found by LSR-BF unit, VC and switch arbitration is disabled for the packet. These dirty packets are now stored in the local router until a special action is initiated (discussed in Section III-D). We do not make any change in the L1 and L2 cache controllers; hence they are unaware of this optimisation. From L1 cache controller viewpoint, an evicted dirty cache block is on its way to or already reached the corresponding L2 cache bank. The L2 cache controller is with the impression that the dirty cache block is on its way. The modified action upon a block eviction with our proposal is given in *red* in Table 1. The working of LSR-BF unit is presented in Algorithm 1.

B. Replying to Cache Block Requests from Local Routers

In a conventional TCMP, when a processor encounters an L1 miss on an evicted, dirty cache block (or any cache block in general), a request message (control packet) is sent over NoC to the corresponding L2 cache bank. Upon receiving the request, the L2 cache bank replies with a response message (data packet, containing the requested block) over NoC, back to the requesting L1. Our proposal modifies this communication between L1 and L2 caches in an attempt to respond to new cache miss requests from the local router, if possible.

According to our proposal, when control packet of the requested message reaches the local router, it checks the non-empty VCs with the help of LSR-BF unit using the technique presented in Algorithm 1. For all the stored, dirty packets in the non-empty VCs, LSR-BF unit compares the address of the requested block with the addresses of the block of these stored, dirty packets. If a match is found, we can satisfy the request message from the local router. LSR-BF unit converts the matched dirty packet into a response message (data packet, containing the requested block) and send it to L1 cache. This is done by changing the source and destination of the dirty packet. The request message is dropped and the local reply is directly sent bypassing the crossbar using *D* and *M* as shown in Figure 3 of our modified router microarchitecture. Again, both L1 and L2 cache controllers are unaware of this optimisation. L1 considers that it has received a response message from L2, while L2 thinks the dirty cache block is still on its way to the L2 cache bank. However, the state of the requested block may be different than the stored dirty block (which is either in owned or modified state). For example, suppose the address matched for a requested cache block, but the requested state is shared. Irrespective of the state requested the local reply is always sent in the same state as that in the matched block (owned/modified). This gives an added advantage as future requests for exclusive access of the block is not required.

Algorithm 1: Working of LSR-BF Unit

Input : VC information, modified packet header
Output: Local store or reply, and block forward
Notations:
n: Number of virtual channels (VC)
τ_i: Time threshold of VC_i, where $0 \leq i < n$
P_i^j: Packet in VC_i where $j \in \{new, stored\}$

```
if P_i^new[Type] == REPLY then
    /* This section is for local store [Section III-A] */
    if P_i^new[Dirty] == 1 then
        τ_i = 128 ∨ 256 ∨ 512
        Disable VA and SA for P_i^new
else
    /* This section is for local reply [Section III-B] */
    for ∀VC_k where τ_k > 0 do
        if P_k^stored[Addr] == P_i^new[Addr] then
            τ_k = 0
            P_k^stored[Local] = 1
            P_k^stored[Dirty] = 0
            P_k^stored[Src] = P_i^new[Dest]
            P_k^stored[Dest] = P_i^new[Src]
            Deallocate VC_i to drop P_i^new
            Bypass crossbar for P_k^stored
/* This section is for TT-BF [Section III-D] */
for ∀τ_i where τ_i > 0 do
    τ_i = τ_i − 1
    if τ_i == 0 then
        P_i^stored[Dirty] = 0
        Enable VA and SA for P_i^stored
/* This section is for MT-BF [Section III-D] */
if P_i^new[Msg] == WB_ACK_DATA then
    for ∀VC_k where τ_k > 0 do
        if P_k^stored[Addr] == P_i^new[Addr] then
            τ_k = 0
            P_k^stored[Dirty] = 0
            Deallocate VC_i to drop P_i^new
            Enable VA and SA for P_k^stored
```

C. Maintaining Cache Coherence

When we consider shared cache, managing coherency across the system is one of the major challenges. When we store evicted, dirty cache blocks in the local router and generate local replies with them, we need to make sure that they are coherent with the entire system. Since both L1 and L2 controllers are unaware of this storage, it is even more complicated and challenging. We store only dirty cache blocks; which are either in *owned* or *modified* state. Hence our discussion about maintaining coherency is limited to them.

For every cache block, the shared L2 cache controller maintains an entry of the corresponding sharers/owner. Only after acquiring a cache block in exclusive state (as owner), L1 cache can modify the block. Hence, the owner of a dirty cache block is the same L1 cache that evicted it out. As long as the dirty cache block is stored in the local router (owned/modified), there is no need to exchange further messages to maintain coherency. During this time, if any other L1 cache requests for the same cache block, L2 cache controller puts it on wait. The L2 cache controller is expecting the modified cache block because it is aware that the L1 cache controller has already evicted the block as per the acknowledgement given by L2.

When we send a local reply from router, the cache block is replied in owned/modified state irrespective of the state it is requested for. The distinction between local reply and others are quantised as a 1-bit value *Local* stored in the packet header as shown in Figure 4. If L1 cache controller receives a reply packet from the local router, it sends a new coherence message *UNBLOCK_PUT_CANCEL* to L2. This coherence message informs L2 that the corresponding dirty cache block for write-back will not be sent. L2 clears the waiting status for that block and all future requests to the same block are directly forwarded to the L1 cache. All of these are the default steps in MOESI coherency protocol, except the sending of *UNBLOCK_PUT_CANCEL*. This way, we ensure that coherency is maintained with our proposed optimisations.

D. Forwarding Stored Dirty Packets

Since there are limited buffers in local NoC router, we can not keep the dirty packets stored there forever. A new packet is injected into the NoC through the local router and they also need free VC in the local input port. If a packet cannot be injected into the local router due to unavailability of a free VC, we identify stored dirty packets there and take necessary steps to move them out of local router buffer. This is done by enabling VC and switch arbitration for dirty packets which were disabled when the dirty packets were stored in the buffer. This makes sure that the corresponding VC will be free in subsequent cycles and hence make space for new injection. By storing dirty packets in local routers, we do not want to delay or suppress new packet injection. In case, we find multiple dirty packets stored in the local router, we pick the oldest of them to enable for arbitration. If we do not find any dirty packet stored, we do not do anything, since all other packets will normally take part in routing and arbitration decisions.

Even in the absence of injection pressure, we do not keep the dirty packets stored forever. The L2 cache controller is expecting the dirty cache block for write-back. It may also be making all other requesters wait for the same cache block. If this wait is very long, then it might hamper the performance of other requesters. We propose two techniques to trigger the forwarding of stored dirty packets from the local router.

Time-Triggered Block Forward (TT-BF): An evicted, dirty cache block is stored in the local router as a dirty packet until a certain time threshold. We add a threshold counter correspond to each VC of the local input port of NoC routers as shown in Figure 3. When a counter reaches the threshold, the dirty packet in the corresponding VC is enabled for arbitration. This triggers forwarding of the dirty block towards its corresponding L2 cache bank for write-back. Till

Table 2: Simulation configuration

Processor	64 OoO x86 cores
L1 cache	16KB, 4-way, 64B blocks, private, split
L2 cache	128KB×64 cores, 8-way, 64B blocks, shared
Directory	4; one located at each corner
Cache Coherence	MOESI distributed directory
NoC	8×8 2D mesh, 4 VCs/port, 128-bit flit channel
Routing	2-stage routers, X-Y dimension-order routing
Packets	1-flit for control packets, 5-flit for data packets
Benchmarks	SPEC CPU2006 (Multiprogrammed)

the time a block is locally stored, access requests for the same block by others is delayed by the time threshold in L2 cache.

Message-Triggered Block Forward (MT-BF): As per MT-BF, while waiting for a write-back cache block if L2 receives a request for the same block by others, it sends a coherence message *WB_ACK_DATA* to L1. It is the same acknowledgement message that L2 send L1 to receive a dirty cache block for write-back. We just make L2 resend the same message when a request comes for the locally stored block. When this coherence message reaches the local router, it triggers forwarding of the stored block. The coherence message is dropped and the dirty packet is enabled for arbitration. LSR-BF unit takes care of this implementation as presented in Algorithm 1. As LSR-BF unit works in parallel with the RC unit, it is not in the critical path of execution.

IV. Experimental Analysis

A. Simulation Framework and Workloads

We model the proposed system on event-driven gem5 simulator [19]. Our system configuration is similar to Intel Xeon Phi Processor 7235 [4] with shared and distributed L2 cache. For certain limitations in gem5, we could not exactly model the cache configuration of Intel Xeon Phi Processor 7235. However, our cache configuration gives a realistic hit rate of around 95% for all the benchmarks we used. Our system configuration is presented in Table 2 for reference. We modify GARNET [20] module in gem5 to implement our modified router microarchitecture. We modify MOESI_CMP_directory protocol in Ruby to maintain cache coherency.

To evaluate the performance, we use various random mixes of SPEC CPU2006 multiprogrammed benchmarks. Each mix consists of a combination of 4 different benchmarks with varying MPKI values and re-reference interval running 16 copies each (4×16: 64) as given in Table 3. By separately profiling each benchmark, we choose a smaller representative window of instructions to have a tractable simulation time.

B. Performance Evaluation

We evaluate the baseline (no optimisation) and proposed architectures for performance, area and power. All our results are normalised with respect to the baseline architecture.

Network Stall Time: It is defined as the number of cycles the processor stalls waiting for a network packet. This metric helps us to understand the direct impact of NoC based communication. Figure 5 shows the normalised network stall time with respect to the baseline system. For all the workload mixes, both TT-BF and MT-BF experience reduction in network stall time over baseline. This is mainly for the local replies, as the packets need not travel all the way to the L2 cache bank or even worse to the off-chip memory. M3 and M4 have significant reduction as they contain benchmarks with lower average re-reference time of evicted, dirty cache blocks.

Table 3: Application scheduling for various workload mixes

Mix	Benchmark Instances			
M1	astar(16)	perlbench(16)	hmmer(16)	sjeng(16)
M2	astar(16)	perlbench(16)	soplex(16)	sphinx(16)
M3	h264ref(16)	gromacs(16)	hmmer(16)	sjeng(16)
M4	h264ref(16)	gromacs(16)	soplex(16)	sphinx(16)
M5	hmmer(16)	sjeng(16)	soplex(16)	sphinx(16)

Miss Latency: It is defined as the number of cycles required to replace a data block from L1 cache with a new incoming block. Figure 6 shows the normalised miss latency with respect to the baseline system. As expected, TT-BF reduces miss latency for all the workload mixes with the local replies. Significant reduction can be seen with MT-BF, as it improves the miss latency of waiting requesters in L2 cache bank by triggering an immediate block forward.

Speedup: We use total Instructions Per Cycle (IPC) of the system to compare speedup. Figure 7 shows the normalised speedup with respect to the baseline system. For all the workload mixes, both TT-BF and MT-BF achieve better speedup than the baseline system. This improvement was intuitive from the reduction in miss latency as shown in Figure 6. M3 achieves the highest speedup of 14.78% as it consists of benchmarks whose average re-reference time of evicted, dirty cache blocks is least. On average, a speedup of 11.97% is achieved across all benchmark mixes.

C. Sensitivity Analysis on Design Parameters

For all results discussed so far with TT-BF optimisation, we have considered time threshold (τ) as 256 cycles. To decide the optimal value of τ, we perform a sensitivity analysis by taking different values of τ. The best logical value to try with is 8401 cycles; the average re-reference time of evicted, dirty cache blocks as shown in Figure 2. However, keeping the dirty cache blocks stored for such a long duration hampers the performance of other requesters who are waiting for the same block in the corresponding L2 cache bank. So, we try a defensive approach; the smallest re-reference time of evicted, dirty cache blocks as shown in Figure 8. As we can see, the smallest re-reference time of all the benchmarks we considered is under 80 with an average value of 49 cycles. So, we conduct our experiments with the value of τ as 64, 128, 256 cycles etc. Figure 9 shows why we chose 256 as the optimal value of τ, as increasing the value beyond that starts hampering the performance. However, for MT-BF optimisation, we keep the value of τ as 8192 ($\approx$ 8401) cycles; closest to the average re-access time of evicted, dirty cache blocks (Figure 2). Hence, in MT-BF optimisation, an evicted, dirty cache block is either

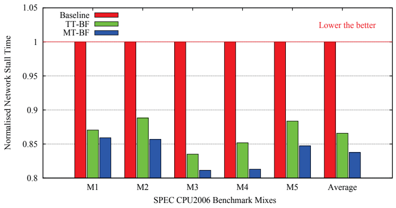

Figure 5: Network Stall Time

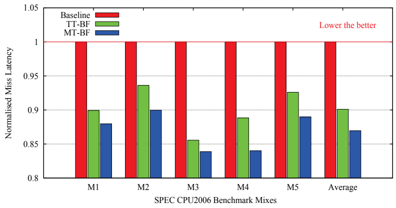

Figure 6: Miss Latency

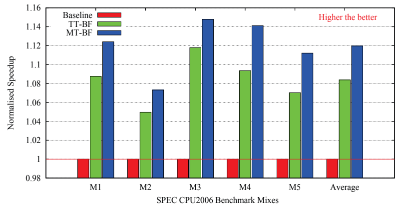

Figure 7: Speedup

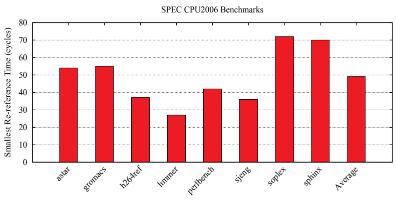

Figure 8: Smallest re-reference time

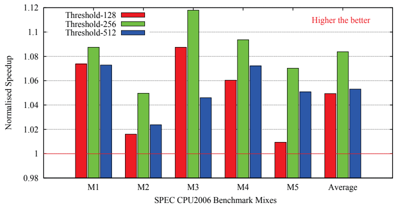

Figure 9: Variation of time threshold

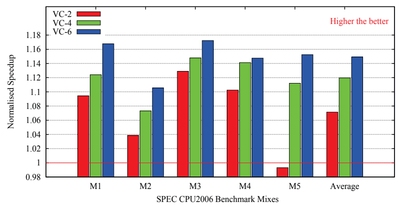

Figure 10: Variation of number of VCs

forwarded after 8192 cycles or forwarded upon receiving a message from L2 cache bank, whichever is earlier. This makes sure that the performance is optimised by taking the best of time threshold and message trigger.

As our optimisation is based on storage of evicted, dirty cache blocks in buffers of VCs, we also explore the impact of number of VCs per port on system performance. For all the results discussed so far, we have considered number of VCs as 4 (as presented in Table 2). However, in Figure 10, we show that varying number of VCs will have different implications on the system performance. Based on the system requirements, appropriate number of VCs can be chosen and hence will get appropriate improvement with our optimisation.

D. Area and Power Evaluation

We use DSENT [21], integrated with gem5 to evaluate the area and power of 8×8 2D mesh NoC in our proposed architectures. In DSENT, we use 22nm processor technology at 1GHz operating frequency. The addition of LSR-BF unit and threshold counters (τ_is) incur a negligible area overhead of 1.02% and a leakage power overhead of 1.03% compared to the baseline routers. However, due to significant improvement in overall system performance, we achieve 4.51% reduction in dynamic power compared to the baseline routers.

V. Conclusion

In this work, we proposed two optimisations to exploit idle buffers in on-chip routers. We used the idle buffers to temporarily store evicted, dirty cache blocks and facilitate local replies for future requests. Our two optimisations are a time-based, and a message-based forwarding of dirty blocks stored in the buffers of local routers. These optimisations are proposed to keep the dirty blocks stored in the local router for as long as possible and yield maximum benefit with local replies. Experimental evaluations demonstrated that our optimisations significantly improves overall system performance. Our future work is about analysing the issues in considering evicted, clean blocks for local storage and reply.

References

[1] N. Barrow-Williams *et al.*, "A Communication Characterisation of SPLASH-2 and PARSEC," in *IISWC*, 2009.
[2] P. Gratz and S. W. Keckler, "Realistic Workload Characterization and Analysis for Networks-on-Chip Design," in *CMP-MSI*, 2010.
[3] R. Hesse *et al.*, "Fine-Grained Bandwidth Adaptivity in Networks-on-Chip using Bidirectional Channels," in *NOCS*, 2012.
[4] (2017) Intel Xeon Phi Processor 7235. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/128694/intel-xeon-phi-processor-7235-16gb-1-3-ghz-64-core.html
[5] B. K. Daya *et al.*, "Quest for High-Performance Bufferless NoCs with Single-Cycle Express Paths and Self-Learning Throttling," in *DAC*, 2016.
[6] G. Michelogiannakis *et al.*, "Evaluating Bufferless Flow Control for On-Chip Networks," in *NOCS*, 2010.
[7] H. Farrokhbakht *et al.*, "SMART: A Scalable Mapping and Routing Technique for Power-Gating in NoC Routers," in *NOCS*, 2017.
[8] N. Eisley *et al.*, "In-Network Cache Coherence," in *MICRO*, 2006.
[9] J. Wang *et al.*, "Network Caching for Chip Multiprocessors," in *IPCCC*, 2009.
[10] A. Yanamandra *et al.*, "In-Network Caching for Chip Multiprocessors," in *HiPEAC*, 2009.
[11] J. Wang *et al.*, "Network Victim Cache: Leveraging Network-on-Chip for Managing Shared Caches in Chip Multiprocessors," in *EMC*, 2009.
[12] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-Chip Networks," in *ISCA*, 2009.
[13] B. Nayak *et al.*, "SLIDER: Smart Late Injection DEflection Router for mesh NoCs," in *ICCD*, 2013.
[14] R. Parikh *et al.*, "Power-Aware NoCs through Routing and Topology Reconfiguration," in *DAC*, 2014.
[15] J. Jose and A. Das, "An Adaptive Deflection Router with Dual Injection and Ejection Units for Mesh NoCs," in *VLSID*, 2018.
[16] J. Rettkowski and D. Gohringer, "Data Stream Processing in Networks-on-Chip," in *ISVLSI*, 2017.
[17] A. Das *et al.*, "Critical Packet Prioritisation by Slack-Aware Re-routing in On-Chip Networks," in *NOCS*, 2018.
[18] K. Sangaiah *et al.*, "SnackNoC: Processing in the Communication Layer," in *HPCA*, 2020.
[19] N. Binkert *et al.*, "The gem5 Simulator," *SIGARCH CAN*, 2011.
[20] N. Agarwal *et al.*, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator," in *ISPASS*, 2009.
[21] C. Sun *et al.*, "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic NoC Modeling," in *NOCS*, 2012.