Improving Memory Efficiency in Heterogeneous MPSoCs through Row-Buffer Locality-aware Forwarding

YANG SONG and BILL LIN, University of California San Diego, USA

In heterogeneous multicore systems, the memory subsystem plays a critical role, since most core-to-core communications are conducted through the main memory. Memory efficiency has a substantial impact on system performance. Although memory traffic from multimedia cores generally manifests high row-buffer locality, which is beneficial to memory efficiency, the locality is often lost as memory streams are forwarded through networks-on-chip (NoC). Previous studies have discussed the techniques that improve memory visibility to reveal scattered row-buffer hit opportunities to the memory scheduler. However, extending local memory visibility introduces little benefit after the locality has been severely diluted. As the alternative approach, preserving row-buffer locality in the NoC has not been well explored. What is worse, it remains to be studied how to perform network traffic scheduling with the awareness of both memory efficiency and quality-of-service (QoS). In this article, we propose a router design with embedded row-index caches to enable locality-aware packet forwarding. The proposed design requires minor modifications to existing router microarchitecture and can be easily implemented with priority arbiters to integrate QoS support. Extensive evaluations show that the proposed design achieves higher memory efficiency than prior memory-aware routers, in addition to providing QoS support. On basis of extant QoS-aware routers, locality-aware forwarding helps to increase row-buffer hits by 58.32% and reduce memory latency by 14.45% on average. It also introduces a net reduction in DRAM and NoC energy cost by 27.82%.

CCS Concepts: • **Computer systems organization** → **Interconnection architectures**; **Heterogeneous (hybrid) systems**; *Multicore architectures*;

Additional Key Words and Phrases: Memory locality, row-buffer hit, energy efficiency, quality-of-service

ACM Reference format:

Yang Song and Bill Lin. 2020. Improving Memory Efficiency in Heterogeneous MPSoCs through Row-Buffer Locality-aware Forwarding. *ACM Trans. Archit. Code Optim.* 17, 1, Article 6 (March 2020), 26 pages.
<https://doi.org/10.1145/3377149>

1 INTRODUCTION

Modern heterogeneous multicore SoC architectures [22, 25] have been widely deployed in mobile devices thanks to their advanced energy efficiency. These multicore SoCs typically integrate a diverse collection of cores. Besides general-purpose cores like the CPU for running user applications, there are also real-time cores dedicated to specific functions, including the GPU, the DSP, multimedia cores (e.g., video codec and camera, etc.), and system cores (e.g., the GPS and

New paper, not an extension of a conference paper.

Authors' addresses: Y. Song and B. Lin, University of California San Diego, 9500 Gilman Dr., La Jolla, CA 92093; emails: {y6song, billlin}@eng.ucsd.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1544-3566/2020/03-ART6

<https://doi.org/10.1145/3377149>

Fig. 1. Architecture of the memory subsystem (including a network-on-chip and memory controllers) for heterogeneous multicore systems.

USB units, etc.). Unlike CPU-centric homogeneous systems where the cache hierarchy handles most core-to-core communications, heterogeneous systems conduct data sharing mostly through the main memory, because shared multimedia data are often too large to fit into any local cache without tremendously raising power and area overhead. Moreover, as DRAM becomes the major shared resource in heterogeneous multicore systems, its memory efficiency often has a direct and substantial impact on system performance.

Architecture of the shared memory subsystem in a heterogeneous MPSoC, including a network-on-chip (NoC) and memory controllers, is shown in Figure 1. The NoC collects memory requests from heterogeneous cores and directs them to their destination memory channels. Each memory channel is operated by an independent memory controller. Once memory requests arrive at the memory controller, they will be stored in request buffers, awaiting to be scheduled to DRAM. At each DRAM clock cycle, up to one memory request can be selected by the memory scheduler based on the scheduling policy. After a scheduling decision has been made, the selected request is sent to the command generator designated to the destination memory bank. Per-bank command generators generate DRAM commands to operate DRAM following memory-access protocol [11].

In a single DRAM channel, the storage structure is organized into multiple hierarchies, including (from top to bottom) ranks, banks, rows, and columns as shown in Figure 1. Channel interface and data bus are shared among ranks that can be operated in parallel when there is no data bus access conflict. Similarly, banks in the same rank are handled separately most of the time. However, only one row can be active in each bank. To read or write a column in an inactive row, the bank must be precharged (i.e., precharge operation) so that any previously active row is deactivated. Then the target row is loaded into the row-buffer (i.e., row activation operation) before column access is performed on this row-buffer. An inherent limitation of DRAM memory efficiency is that precharge and activation operations incur latency and energy penalties without contributing to actual data transfers. To cope with this limitation, the memory scheduler monitors row-buffer states in all banks and avoids triggering unnecessary precharges and activations.

Improving memory efficiency by increasing row-buffer hits per row activation is a major task of memory scheduling. First-Ready First-Come-First-Serve (FR-FCFS) policy [26] is prevalently used to prioritize requests for open rows if available and otherwise prioritize older requests. Minimalist scheduling policy [15] was proposed to garner open row-buffer hits between precharges and activations without causing unfairness among different cores/threads. Batching mechanism [1] has also been used to collect row-buffer hits and relieve the memory interference against latency-sensitive requests.

Besides memory scheduling policies, prior work have also explored improving memory visibility for the scheduler so that more row-buffer hit opportunities can be revealed. Several cache replacement and eviction policies in consideration of row-buffer states [13, 16, 27, 30] have been proposed for CPU-centric homogeneous systems. These work extend the candidate pool for memory scheduling into caches to provide more visibility of memory traffic to the scheduler. Nonetheless, when memory locality is low, potential hits for open rows are often waiting in NoC routers that are

beyond the visible range of the scheduler, even given local visibility extension. As the transporter of memory requests, the NoC plays a critical role in forwarding memory traffic and preserving row-buffer locality during transmission. However, while most extant designs for heterogeneous systems are focused on Quality-of-Service (QoS) support [9, 20, 23, 36], there have been very few studies [12, 24, 35] discussing the improvement of row-buffer locality in the NoC for the benefit of memory efficiency. One of the earliest such studies was presented in Reference [35]. The authors proposed a VC allocation policy that improves row-buffer locality in GPU shader memory traffic. Similarly, a network flow control protocol that considers DRAM timing constraints was introduced in Reference [12]. As one of the latest advancements, an NoC router design that keeps track of DRAM bank states was studied in Reference [24]. In addition, prior work are focused on either memory locality improvement or QoS support. It remains to be investigated how to combine these two often-conflicting goals during network traffic scheduling.

In this work, we propose a locality-aware NoC router design with lightweight row-index caches for row-buffer locality-aware traffic forwarding. The goal of the proposed design is to shorten the delay between requests for the same row so that they can arrive at the memory controller within a visible distance. Meanwhile, the proposed design is capable of providing QoS support. The contributions of our work can be summarized as follows.

- We show the challenge of improving memory efficiency in heterogeneous MPSoCs and identify the limitation of previous approaches. Without managing memory locality through the NoC, local extension of memory visibility introduces trivial improvement.
- We propose a new locality-aware NoC router microarchitecture. The proposed microarchitecture is equipped with row-index caches to track row indices that have appeared in recently forwarded packets. These lightweight caches enable efficient locality-aware forwarding in NoC routers and can be implemented with extant priority-based allocators.
- We introduce a locality-aware allocation scheme that improves row-buffer locality and provides QoS support at the same time.
- We evaluate the proposed design on several NoC topologies with the memory traffic of a state-of-the-art heterogeneous MPSoC [25]. Evaluation results show that our locality-aware NoC router achieves higher row-buffer hit count and lower memory latency than previous memory-aware methods, in addition to providing QoS guarantee. Compared with QoS-aware routers, the proposed design increases row-buffer hits by 58.32% and reduces average memory latency by 14.45%.

The rest of this article is organized as follows: Section 2 describes the background and motivation of our work. Section 3 details our router design with locality-aware forwarding and QoS support. Section 4 presents the evaluation results. A review of related work and conclusions follow in Sections 5 and 6.

2 BACKGROUND AND MOTIVATION

Memory subsystem design for heterogeneous MPSoCs has been extensively investigated by previous work. A major challenge being tackled by this body of literature is the delivery of quality-of-service (QoS) to heterogeneous cores in consideration of the disparity in their memory access patterns, data access rates, and performance targets. While QoS awareness in the NoC has been well explored [9, 20, 23, 36], memory performance, usually in conflict with QoS, is often ignored by NoC designs.

Multimedia data constitute a large portion of memory traffic in heterogeneous MPSoCs. This part of memory traffic typically possesses high spatial locality that is beneficial to row-buffer hit rate in DRAM. However, as traffic flows merge into a single stream to DRAM, they often dilute

Fig. 2. Average numbers of row-buffer hits per row activation with respect to different numbers of heterogeneous cores being served in the memory subsystem. A single arbiter applying round-robin policy is used to forward memory traffic to the memory controller. Emulated real-time cores are listed in Table 3. CPU test case 1 (Table 2) and high-performance mode (Table 3) are used for simulation. Experiment setup is detailed in Section 4.

Fig. 3. An example of memory locality of NoC traffic affecting memory efficiency in DRAM. Assume five request buffers are designated to each memory bank. Requests are labeled by indices of their target rows. Those for the same row are painted in the same color. FR-FCFS policy is applied by the memory scheduler.

each other. Without proper care, memory locality is quickly lost during the dilution, resulting in fewer row-buffer hits and lower memory efficiency in DRAM. Figure 2 shows the average number of row-buffer hits with respect to different numbers of heterogeneous cores sharing the memory subsystem. As more cores are included, the average number of row-buffer hits drops rapidly from 4.55 to 2.64.

To explain this phenomenon, Figure 3 illustrates two scenarios with different degrees of memory locality leading to different memory efficiencies. Assume there are five request buffers designated to each memory bank. Only requests in these buffers are visible to the scheduler that applies FR-FCFS policy for memory scheduling. In both scenarios, the memory controller receives the same requests from the NoC for memory bank K but in different orders. The memory traffic in Figure 3(a) has low row-buffer locality, meaning that requests for the same rows are separated far apart. As a result, the memory scheduler cannot find the opportunity to access any active row. Instead, the scheduler keeps precharging the row-buffer and serves memory requests in their arrival order. In the other scenario shown in Figure 3(b), memory requests for the same rows stay close enough so that they are visible to the scheduler at the same time. Two row-buffer hits are found after each row activation. By comparison, the second scenario is more desirable, because less delay and energy overhead are induced. However, the first scenario is more common in heterogeneous systems due to locality dilution.

Locality dilution can be tackled in two ways: (1) increase memory visibility and make more requests visible to the scheduler so that the dilution can be tolerated and (2) preserve memory locality and prevent requests for the same row from being dispersed. The first approach has been

Fig. 4. Average numbers of row-buffer hits per row activation with respect to different numbers of request buffers (memory visibility) in the memory controller and different maximum burst lengths (memory locality). Experiment setup is the same as in Figure 2.

well explored by previous work [13, 16, 27, 30], whereas the second approach has drawn little attention in recent years [12, 24, 35].

Figure 4 shows the results from varying memory visibility and locality. In simulation, a single arbiter fetches requests from cores to the memory controller in a round-robin fashion. Different extents of memory visibility are realized by different numbers of request buffers in the memory controller. To vary memory locality, we modify round-robin policy at the single arbiter so that a requestor can be served multiple times in consecutive cycles. The goal is to create bursty traffic so that memory locality can be partially preserved during arbitration. Figure 4 includes the results from applying different burst lengths. The maximum burst length refers to the maximum number of times that a requestor can be served in a row, if possible, before the next requestor takes its turn. A longer burst preserves more memory locality.

As shown, the average number of row-buffer hits is improved trivially as the buffers are increased from 10 to 80 regardless of burst length. This is because in a multicore system with numerous traffic flows, the memory locality is often too diluted to be recovered by a reasonable number of request buffers. Alternatively, we can serve the same core multiple times in a row to give it the exclusive access to memory fabrics without locality dilution. When the burst length is sufficiently high, requests from a heterogeneous core are able to make it to DRAM without locality dilution caused by others. Thus the average number of row-buffer hits is augmented quickly with increasing burst length. By comparison, creating bursty traffic provides a more noticeable impact on row-buffer hits than adding request buffers. However, real-time cores with small bandwidth footprints can be easily preempted by cores generating bursty traffic. Therefore, we need a more sophisticated approach to preserve memory locality during traffic forwarding.

Yuan et al. [35] investigated row-buffer locality dilution in GPU architectures and introduced two policies for virtual channel allocation in NoC routers to alleviate the dilution. One arbitration policy is Hold-Grant (HG), which grants the highest priority to the input port that was served in the previous cycle. The other policy, Row-Matching Hold-Grant (RMHG) holds the grant for the previous winner only if the pending request shows the same row index with the previous request. Although the authors showed that both policies are able to greatly improve the performance of an in-order first-in-first-out DRAM scheduler, it is not clear how these policies can integrate QoS support, which is increasingly important in modern heterogeneous systems. In particular, both HG and RMHG policies only keep a record of the most recent request, which does not allow interruptions from other mechanisms such as QoS support. For example, supposed a VC allocator stops a memory stream with high locality to serve a latency-sensitive stream, the allocator will not be able to continue the high-locality stream afterwards, since it has been “forgotten.” In fact, it is unnecessary to batch requests for the same row throughout the network, because

out-of-order memory schedulers, which are fairly common in state-of-the-art MPSoCs, are capable of harvesting row-buffer hits given the requests are sufficiently close to each other. Jang et al. [12] proposed a DRAM-aware flow control protocol. In this protocol, VC allocators prioritize the packets that travel to the same row as the preceding packet and avoid allocating a virtual channel to any packet, which can cause bank conflict with its predecessor. It facilitates batch formation of packets for the same row-buffer at source. However, it bares the same limitation as in Reference [35]. Namely, it does not leave room for the implementation of QoS awareness. Pimpalkhute et al. [24] introduced a DRAM-aware packet scheduling scheme in the routers that are one-hop away from the memory controller. Although the proposed scheme expands the memory visibility from the memory controller into its neighbor routers, it is still unable to prevent locality dilution that happens near the source. Moreover, this scheme cannot be easily integrated into extant QoS-aware designs.

In this work, we propose a locality-aware NoC router design to preserve row-buffer locality of memory traffic without degrading the QoS for heterogeneous cores.

3 LOCALITY-AWARE FORWARDING

3.1 Preserving Row-Buffer Locality

As discussed in the previous section, the memory traffic coming out of a single real-time core usually shows high row-buffer locality. However, the locality is often diluted when different memory streams are merged into one, as they travel through the same links in the NoC. To alleviate the dilution, we prefer requests for the same rows to stay close to each other during transmission. To achieve that, we introduce the *row-index caches* to NoC routers. A row index refers to the address bits that locate the target row in DRAM, including the bits indexing the channel, the rank, the bank, and the row of the destination address. As the name suggests, row-index caches record row indices of recently forwarded packets that are going to DRAM. By referring to these caches, allocations can be performed in consideration of row-buffer locality.

Caching of row indices is done separately for each output port to preserve the locality of the memory traffic traveling through the same link. When a packet is requesting for VC/switch allocation, the allocator checks on the cache with the row index of this packet. If there is a cache hit, then the allocator will prioritize this packet to shorten its queuing latency so that it can stay in a visible distance from the previous packet going to the same row. If it turns out to be a cache miss, then it means either this row index has not appeared before or it has appeared a while ago. In either case, this packet will not be prioritized. This simple approach reorders memory streams to improve row-buffer locality without keeping track of DRAM row-buffer states or timing constraints. In addition, it is oblivious of NoC topology. A locality-aware policy for packet forwarding in the network is specified as below.

- **Locality-aware RR policy:** If packet A hits the row-index cache and B does not, then choose A. Otherwise, perform round-robin arbitration among all requestors.

Since cache implementation techniques have been well studied so far, integrating caches into NoC routers does not require sophisticated architectural modifications. Unlike conventional caches that keep both tag and data for each cache line, row-index caches only store tags, which further lowers implementation complexity.

3.2 Accessing Row-Index Cache

A straightforward way to implement row-index caches is to insert a cache at every output port. Packets requesting for an output port access the associated cache. However, when there is more

than one packet requesting for the same output port, multiple accesses will happen to the same cache, which cannot be done in a single clock cycle. To avoid cache access conflicts, we adopt per-input row-index caches instead so that cache access can be performed immediately at input once a packet arrives. Inside the cache, each cache set is designated to a distinct output port to allow independent caching per output. These input caches are identical in that cache sets for the same output port remain the same in all caches. Once an allocation is determined by the allocator, all row-index caches will be refilled with the same entry, i.e., the allocatee's row index, in the designated cache set.

When multiple allocation decisions are made with respect to different output ports, these decisions will trigger multiple refills to every input cache. To circumvent frequent cache refills, only VC allocation decisions are broadcast to row-index caches. A cache refill queue is located before every row-index cache to buffer refills from VC allocator that cannot be written into the cache at current cycle. In addition, to allow concurrent cache access and refill, an SRAM with one read and one write ports is used to implement a row-index cache.

3.3 Integrating QoS Support

Without regard to the QoS, locality-aware forwarding will easily delay memory streams with low memory locality but high time criticality, leading to worsened system performance. To ensure that locality-aware forwarding is not performed at the expense of system performance, it is necessary to integrate QoS support into our locality-aware router.

Previous work on QoS-aware NoC [9, 20, 23, 36] have used priority arbiters to realize QoS support in NoC routers. Specifically, in these approaches packets are categorized in distinct frames or classes that are served based on their priority levels. There have also been works [28, 32] adjusting priority levels of network packets according to requestors' QoS experiences. In Reference [28], the priority of a packet is set to reflect the requestor's actual performance relative to its target QoS. The resulting priority level is attached to the packet and can be used for priority-based round-robin (specified below) in the memory subsystem.

- **QoS-aware RR policy:** Suppose P_A and P_B are priorities for transactions A and B. If $P_A > P_B$, then choose A; if $P_A < P_B$, then choose B; otherwise, choose between A and B in round-robin manners.

To combine QoS support with locality awareness, we need to introduce row-index cache into extant priority schemes. This can be done by assigning extra credits to the priority levels of requestors that hit row-index caches. An example of such a locality-aware priority scheme is shown below.

- **Locality-aware QoS policy:** Suppose packet A hits the row-index cache and B does not. If $P_A, P_B < \delta$ or $P_A = P_B$, then choose A. Otherwise, perform QoS-aware RR.

The parameter δ is an adjustable threshold used to balance locality-aware forwarding and QoS-aware allocation. When the priority level is lower than δ , the allocator focuses on row-buffer locality; otherwise, QoS support comes first. A higher δ value gives more favor to row-buffer locality but also potentially causes more disturbance to the QoS. Given a priority range of 0 to 7, we found $\delta = 6$ a good setting to improve memory efficiency without causing QoS degradations.

3.4 Microarchitecture

The proposed microarchitecture is based on a canonical NoC router design [4]. The pipeline of the canonical NoC router consists of four stages: buffer write (BW)/route computation (RC), VC allocation (VA), switch allocation (SA), and switch traversal (ST). Link traversal (LT) typically takes

Fig. 5. Microarchitecture of the locality-aware NoC router with per-input row-index caches.

another cycle as the flit is forwarded to the next router. The router adopts lookahead routing to determine output ports one hop earlier. Figure 5 shows the microarchitecture of the locality-aware NoC router. Compared with the canonical router design, the locality-aware router is equipped with per-input row-index caches to track row-buffer locality in recent packets. It also requires priority arbiters to enable locality-aware VC/switch allocations. The updated pipeline stages will be specified in the rest of this section.

Row-Index Cache Access: Once a head flit arrives at the input buffer, its destination information, including output port index and DRAM row index, is used to access the row-index cache associated with this input port. Cache access by the newly buffered head flit is concurrent with route computation, which means no extra cycles are introduced to the router pipeline by cache access. Since a row-index cache only performs tag comparisons without data transfers, cache access can be finished in one cycle and the result will be available for VC allocation in the next cycle. When there is no incoming head flit, cache access can still be performed on extant head flits in case the results will be different due to recent cache refills from VC allocator.

Locality-aware Allocation: Cache access result for a packet is encoded into a one-bit flag and collected by the VC/switch allocator, along with the priority level of this packet. Given above information, allocation policies with the awareness of both locality and QoS can be applied at this stage.

Row-Index Cache Refill: After VC Allocation is finished, allocatees' row indices are broadcast to every input cache and buffered in the cache refill queue if they cannot be written back in current clock cycle. In parallel with cache access, cache refill is performed in every cycle as long as the refill queue is nonempty. The classic Least-Recently Used (LRU) replacement policy is applied in row-index caches. In case of overflow at a cache refill queue, the oldest refill will be discarded to be consistent with the LRU policy.

3.5 Implementation Cost

Compared with canonical routers, the extra implementation cost of the locality-aware router mainly comes from the storage overhead of row-index caches. In a memory subsystem with 2-GB memory and 2-KB page size, we need $\log(2G/2K) = 20$ bits to encode a row index. Thus, a row-index cache stores $20N_{way}N_{port}$ bits in total, in which N_{way} is the number of cache ways per set/output and N_{port} is the number of router ports. Assume the same amount of storage is implemented in the cache refill queue; the locality-aware router requires an extra storage of $40N_{way}N_{port}$ bits per input port.

Table 1. Memory Subsystem Simulation Configurations

NoC	1-GHz clock rate, 5×5 mesh and asymmetric networks, dimension-order routing, 128-bit link width, 4 VCs per port, five buffers per VC, separable VC/switch allocator
MC	40 request buffers, FR-FCFS policy, address mapping scheme (from physical address to DRAM address): channel-rank-row-bank-column
DRAM	1866-MHz clock rate, 2-GB memory, 2-KB page size, Channels-Ranks-Banks: 1-2-8, CL-tRCD-tRP(cycles): 36-34-34, tWTR-tRTP-tWR(cycles): 19-14-34, tRRD-tFAW(cycles): 19-75

Table 2. CPU Benchmark Descriptions

case 1	art-mcf-sjeng-sphinx3	case 3	mcf-sjeng-sphinx3-hmmer
case 2	art-mcf-sjeng-hmmer	case 4	art-mcf-sphinx3-hmmer

4 EVALUATION

In this section, the proposed NoC design with locality-aware forwarding will first be compared with prior memory-aware routers without integrating QoS support. Performance metrics of memory efficiency, including row-buffer locality and memory latency, will be evaluated. Furthermore, the proposed design will be demonstrated in its capability of improving row-buffer locality while providing QoS guarantee. In addition, the energy savings and overhead by the proposed design will be assessed.

4.1 Methodology

For evaluation, the proposed architecture shown in Figure 5 is implemented in a cycle-accurate NoC simulator. Dimension-order routing and wormhole flow control are applied in the NoC. We assume a memory request, i.e., an NoC packet, contains 64 bytes and an NoC flit contains 128 bits. Once injected into the NoC, a packet is converted into one head flit encoded with destination information and another four flits carrying actual data. In NoC routers, we adopt the four-stage pipeline. Each input port of the router contains four virtual channels and a row-index cache. Each virtual channel buffers up to five flits. The row-index cache contains a four-way cache set for each output port. A memory controller (MC), simulated by DRAMSim2 [33] with a LPDDR4 timing model, is integrated into the NoC simulator. A single memory channel with 2-GB address space is shared by all heterogeneous cores. Detailed simulation configurations are depicted in Table 1.

Memory traffic of real-time cores is produced in the same way as in Reference [28]. Specifically, memory requests are generated per-core and per-DMA according to memory specifications of Qualcomm Snapdragon 845 [25]. Our memory traffic generator models distinct memory access patterns of each DMA in Snapdragon 845, including their address access patterns and request sizes. We also take into account per-core configurations such as frequency, the number of memory ports, and the number of outstanding requests per port. Peak bandwidth consumptions by these DMAs are set by a camcorder application running at 30 fps. For example, the frame rotator reads and writes 1080p YUV420 images at 30 fps, consuming 89 MB/s for read and write requests, respectively, and 178 MB/s in total. For the CPU, we emulate four cores sharing a 2-MB L2 cache. Each CPU core executes an application from SPEC 2000/2006 benchmarks and injects memory traffic into the NoC. Different combinations of CPU applications are tested as listed in Table 2.

Since most real-time cores do not operate all the time, we emulate real-time traffic in two separate modes including high-performance and low-power modes. In high-performance mode, more

Table 3. Specifications of Emulated Real-time Cores and Their Statuses in High-Performance (HP) and Low-Power (LP) Modes

Real-time cores		Clock frequency	Request size	Access pattern	# of DMAs	Bandwidth usage	High perf	Low perf
GPU		533 MHz	128 B	linear	2	2 GB/s	on	on
DSP		533 MHz	32 B	random	2	365 MB/s	on	on
Multimedia	Video Codec	406 MHz	64-256 B	linear	6	374 MB/s	on	off
	JPEG Encoder	406 MHz	256 B	linear	2	432 MB/s	on	off
	Display	406 MHz	128 B	linear	5	3928 MB/s	on	on
	Camera	406 MHz	128 B	linear	14	1169 MB/s	on	on
	Frame Rotator	406 MHz	128 B	linear	2	178 MB/s	on	off
	Image Processor	406 MHz	256 B	linear	5	445 MB/s	on	off
System	USB	240 MHz	128 B	linear	2	372 MB/s	on	on
	GPS	240 MHz	32 B	random	3	71 MB/s	on	on
	Modem	466 MHz	128 B	linear	2	705 MB/s	on	off
	WiFi	240 MHz	32 B	linear	1	33 MB/s	on	on
	Audio DSP	466 MHz	128 B	linear	1	250 MB/s	on	off

real-time cores are active and more traffic is generated than in the other mode. Table 3 shows specification summary of emulated real-time cores and their statuses in different emulation modes. The listed specifications include clock frequency, memory request size, number of DMAs per core, and bandwidth usage required to achieve performance target. Most of the emulated real-time DMAs access sequential (linear) addresses, except for DSP and GPS. Memory requests are generated at each core at a constant rate that is determined by the frequency and bandwidth usage, except for video codec, JPEG, frame rotator, and image processor. In these multimedia cores, memory requests are issued as fast as possible in the beginning of every frame period until the maximum number (32) of outstanding requests has been reached.

Evaluations are conducted upon different network topologies, including mesh and asymmetric networks, as shown in Figure 6. For the mesh networks, heterogeneous cores are evenly distributed on a 5×5 mesh with a MC in the center. Some major real-time cores are divided into smaller pieces with equal bandwidth to balance traffic load on the mesh. We test two layouts (Figure 6(a) and (b)) of heterogeneous cores on the mesh network. Note that in heterogeneous MPSoCs, DMAs belonging to the same core are usually clustered and share the same interface with the NoC. Therefore, we also test asymmetric networks with undivided real-time cores. In Figure 6(c) real-time cores are placed based on their functions. In Figure 6(d), heterogeneous cores are positioned in imitation of the layout of Snapdragon 845.

In addition to RR and FCFS policies, we will test several network scheduling policies, including the proposed locality-aware schemes, which are listed in Table 4. The Dual Scheduled Packet (DSPK) scheme was introduced in Reference [24], which prioritizes requests to open rows and idle banks to improve row-buffer hit rate and bank-level parallelism at the same time. It is implemented in routers within one-hop distance from the MC. In other routers, RR policy is applied. In Reference [24], it was shown that DSPK outperforms previous approaches such as in Reference [12]. On the basis of DSPK, we introduce a simplified version of DSPK in which routers (that are located within a one-hop radius of the MC) track row-buffer states and prioritize requests for open rows such that potential row-buffer hits have the highest priority. We refer to this method as row-buffer-aware round-robin (RB-aware RR) policy. Furthermore, we apply the same scheduling

Fig. 6. Summary of tested NoC topologies, including two 5×5 mesh networks and two asymmetric networks.

policy in all routers assuming row-buffer states are broadcast in the network. We call this scheme as Full RB-aware RR. HG and RMHG policies were proposed in Reference [35] to improve row-buffer locality of GPU shader traffic. In addition, we introduce a revised version of RMHG policy with QoS support, i.e., QoS-aware RMHG. Last, we will also demonstrate Locality-aware RR, QoS-aware RR, and Locality-aware QoS policies, which have been explained in Section 3. Note that aforementioned policies are only applied to VC allocation in NoC routers. Round-robin policy is adopted for switch allocation in all cases.

4.2 Row-buffer Locality

In this section, we will first evaluate the effectiveness of our locality-aware router in terms of locality preservation without considering QoS guarantees.

Although memory locality is a well-known concept, it does not come with a standard metric for evaluation. Therefore, to visualize row-buffer locality, we collect the first 10,000 memory requests arriving at the MC. For each request, we record its arrival time and destination row in DRAM. We also check whether the destination row has been accessed by previous requests. If so, then we calculate arrival time difference between the current request and the last request that has accessed the same row. This arrival time difference associates with the likelihood of open row-buffer hits, as a smaller time difference means a higher chance for these two requests to hit the same row before it is closed. Furthermore, we collect arrival time differences from 10,000 memory requests and show their distribution over a window period of 300 cycles in Figure 7. More occurrences of time differences in small cycles is an indication of higher row-buffer locality. Mesh network I and CPU test case 1 are used in simulation. Real-time cores are set to the high-performance mode. For comparison, four network scheduling policies are implemented, including RR, FCFS, DSPK, and Locality-aware RR policies.

As shown in Figure 7, RR policy results in the least number of occurrences within 100 cycles. That means RR arbitration leads to the least number of requests arriving at the same row within

Table 4. Network Traffic Scheduling Policies

Dual Scheduled Packet (DSPK) [24]	In the routers that are connected to MC or one-hop away from MC, prioritize requests whose destination rows are currently open. Otherwise, prioritize the requests going to idle banks that are ready for activation (not limited by four-bank activation window [11]). In other routers, round-robin arbitration is applied.
Row-Buffer-aware Round-Robin (RB-aware RR)	In the routers that are connected to MC or one-hop away from MC, prioritize requests whose destination rows are currently open. If there are no such requests, then perform round-robin among all.
Full RB-aware RR	In all routers of the network, prioritize requests whose destination rows are currently open. If there are multiple such requests, then perform round-robin among them.
Hold-Grant (HG) [35]	In all routers, repeat (up to eight times) the choice by the last arbitration if the previous requestor is still available. Otherwise, perform round-robin.
Row-Matching Hold-Grant (RMHG) [35]	In all routers, repeat the choice by the last arbitration if the previous requestor is still available and its destination row index matches with the previous request. Otherwise, perform round-robin.
Locality-aware RR	In all routers, prioritize row-index cache hits. Perform round-robin if there are multiple hits. If there are no hits, then perform round-robin among all requests.
QoS-aware RR	In all routers, choose requests with the highest priority level. ¹ If there are multiple requests sharing the highest priority, then perform round-robin among them.
QoS-aware RMHG	In all routers, repeat the choice by the last arbitration if the previous requestor is still available and its destination row index matches with the previous request. Otherwise, perform QoS-aware RR.
Locality-aware QoS	In all routers, if there are requests with priorities ≥ 6 of $[0, 7]$, choose the one with the highest priority. In the case of multiple requests sharing the highest priority, choose the one that hits the row-index cache. If there are no requests with priorities ≥ 6 , then perform QoS-aware RR among requests that hit the row-index cache.

a window period of 100 cycles. This is because RR policy causes the most severe dilution to row-buffer locality by interleaving traffic flows. FCFS policy leads to slightly more occurrences within 100 cycles than RR, because FCFS preserves the arrival order of NoC packets in each router, which helps to relieve dilution to memory streams with high data rates. In contrast, DSPK and Locality-aware RR both obtain much higher row-buffer locality as many more occurrences happen within 100 cycles. We will conduct further analysis on these two policies and others in the following experiment.

A direct result from preserving row-buffer locality is the increase of row-buffer hits per row activation in DRAM. Figure 8 summarizes the average row-buffer hits per activation of all test cases/modes and networks. Several observations can be made from Figure 8. First, CPU applications show trivial impacts on row-buffer hits as different applications lead to very similar results given the network topology and real-time traffic mode, because most memory traffic comes from real-time cores. Second, under most policies, more improvement on row-buffer hits happens in highperformance mode. This is because in this mode more memory requests are generated, as well as the locality dilution. In addition, mesh networks show more potential in row-buffer hit

¹Priority levels are calculated at sources according to end-to-end QoS of real-time cores. More details can be found in Reference [28].

Fig. 7. Distribution of arrival time differences between requests accessing the same rows, ranging from 1 to 300 cycles. Arrival time differences above 300 cycles are not included in the figure.

Fig. 8. Average row-buffer hit counts per row activation by different network scheduling policies. One million DRAM clock cycles are simulated.

improvement, because memory traffic travels through more routers and suffers from more dilution in meshes.

Among the tested policies in Figure 8, RR and FCFS policies have the fewest row-buffer hits, because both of them are ignorant of memory locality. FCFS policy has slightly better results than RR, because it maintains the issuance order of memory requests, which helps to preserve memory locality of bursty traffic from multimedia cores. Row-buffer hits by DSPK and RB-aware RR policies stay close to each other in all cases, because these two policies are similar and only implemented in part of the NoC. Compared with RR policy, DSPK and RB-aware RR policies improve row-buffer hits by 18.8% and 18.5%, respectively, on average. Full RB-aware RR policy improves RB-aware RR by 22.3% through implementing memory-aware VC allocations in all routers. It achieves 45.0% more row-buffer hits than RR policy. However, the effectiveness of this method is limited,

because it can be pointless to prioritize a memory request for an open row if it is far from an MC. In that case, the row may be closed by the time when this request finally arrives at the MC. Therefore, Full RB-aware RR policy achieves fewer row-buffer hits compared with HG, RMHG, and Locality-aware RR policies. Instead of tracking row-buffer states, these three policies focus on improving row-buffer locality, which is effective regardless of router location and NoC topology. In all test cases, RMHG policy performs better than HG, because it is more aware of row-buffer locality instead of blindly serving the same requestor. On average, HG and RMHG policies outperform RR by 67.1% and 96.4%, respectively. Comparisons between RMHG and Locality-aware RR policies show different results on different NoC topologies. While Locality-aware RR achieves the highest row-buffer hit counts on mesh networks, RMHG outperforms Locality-aware RR on asymmetric networks. On average, Locality-aware RR policy increases row-buffer hits by 99.3% in comparison to RR. Since RMHG is essentially Locality-aware RR with one cache line per set in row-index caches, the difference between their performances indicates the impacts of row-index cache size over locality preservation. Specifically, memory traffic in the mesh networks travels through more routers than that in the asymmetric networks, thereby suffering from more locality dilution. Having more cache lines in row-index caches helps to record more requests in the past and provides better locality preservation. Thus Locality-aware RR policy delivers better performance on mesh networks. However, if locality dilution is not severe, then adding more cache lines can introduce too much competition. As an extreme case, a row-index cache with infinite capacity records all the rows that have been activated in the past. Thus, it would be ineffective in detecting temporal locality of memory traffic. For a similar reason, Locality-aware RR is outperformed by RMHG on asymmetric networks. More analysis results on row-index cache size will be presented in Section 4.5.

Increasing row-buffer hits helps to reduce memory latency by avoiding the latency penalty caused by unnecessary activation and precharge operations. The summary of normalized average memory latency is shown in Figure 9. Locality-aware RR results in the lowest memory latency on average, which is 17.44% lower than RR and 12.31% lower than DSPK.

To further evaluate row-buffer locality and memory latency for the longer term, we simulate memory system for 10 ms, as shown in Figure 10, where CPU test case 1 is applied in all cases. Row-buffer hit rate is calculated as the ratio of the number of open row accesses over total number of DRAM accesses. Similarly to Figure 8, Locality-aware RR and RMHG policies achieve similar row-buffer hit rates among all cases, although Locality-aware RR shows slight advantage on mesh networks and RMHG performs somewhat better on asymmetric networks. On average, both Locality-aware RR and RMHG policies achieve around 37% higher row-buffer hit rate than RR. As for average memory latency, Locality-aware RR achieves shorter average latency than RMHG by tracking memory locality of more rows and thus enabling higher inter-bank parallelism. On average, Locality-aware RR and RMHG reduce latency by 16.51% and 12.96%, respectively, compared with RR policy.

4.3 Row-buffer Locality with QoS Support

So far, we have demonstrated the efficacy of our locality-aware NoC router in improving memory efficiency. However, in heterogeneous MPSoCs, QoS is a critical metric for user experience that can be harmed by memory efficiency improvement. For example, the DSP generates random memory access requests, which means it is subject to extra delay when linear memory accesses are prioritized to improve memory efficiency. What is worse, the DSP is latency-sensitive and requires its average memory latency to be limited within a certain threshold. Without regard to the DSP's QoS, i.e., memory latency, improving row-buffer locality could bring more damage than benefits.

Fig. 9. Normalized average memory latencies by different allocation policies. All results are normalized by the average latency achieved by RR policy. One million DRAM clock cycles are simulated.

Fig. 10. Row-buffer hit rates and normalized (to RR policy) average memory latencies by different allocation policies.

As discussed in Section 3.3, prior work have proposed frameworks to adjust priority levels of memory requests based on their QoS targets. We have shown that QoS support can be integrated into locality-aware routers so that memory efficiency is improved without violating QoS guarantees. To evaluate memory efficiency improvement under QoS support, we implement the Locality-aware QoS policy from Section 3.3, as well as the QoS-aware RR policy. In our implementation, QoS priority levels are generated at source as proposed in Reference [28]. The latency sensitivity and the random memory access pattern make the DSP an ideal example for the demonstration of achieving QoS target during locality-aware forwarding. Therefore, we use DSP memory latency as the QoS target to be evaluated. In our evaluation, the average memory latency limit of the DSP is set to $0.8\mu\text{s}$.

Figure 11 shows real-time DSP memory latency and row-buffer hits per row activation in DRAM under different scheduling policies, including DSPK, RMHG, Locality-aware RR, QoS-aware RR, and Locality-aware QoS. Asymmetric network I and CPU test case 4 are used in simulation. Without regards to memory latency of the DSP, prioritizing memory streams with high locality

Fig. 11. DSP memory latency and row-buffer hit count over time. The maximum limit of DSP latency is $0.8\mu s$. Asymmetric network I and CPU test case 4 are used for simulation. Real-time cores are set to HP mode.

to garner row-buffer hits de-prioritizes DSP requests. Therefore, DSP memory latency increases as more row-buffer hits are collected. As QoS-ignorant policies, DSPK, Locality-aware RR, and RMHG lead to increasing row-buffer hits as well as increasing DSP latency. All of them fail to limit DSP latency within $0.8\mu s$. As for QoS-aware RR policy, it achieves the lowest latency for the DSP, since QoS is its sole focus but also results in the lowest row-buffer hit count, since it does not take into account memory efficiency. Last, Locality-aware QoS limits DSP latency below the given threshold while obtaining a decent amount of improvement on row-buffer hit count compared with QoS-aware RR.

Figure 12 summarizes DSP latency violations in all test cases. Latency violation is defined as the portion of average latency exceeding the maximum latency limit. Thus, a positive value indicates longer latency than the maximum limit and a negative value indicates shorter latency within the limit. Without QoS support, DSPK, RB-aware RR, HG, RMHG, and Locality-aware RR often fail to meet the latency requirement of the DSP. This is especially obvious in mesh networks where memory requests travel through more routers to DRAM. DSP memory latency also tends to be higher when real-time cores are in high-performance mode due to the intensified memory competition.

To demonstrate the necessity of implementing QoS support in the network, we include another method in Figure 12 where Locality-aware RR is applied in the NoC to improve memory efficiency while QoS-aware memory scheduler proposed in Reference [1] is implemented in the MC. In the QoS-aware MC, memory requests are batched based on their target rows. These batches are scheduled to DRAM based on their priority levels. As shown, this method shares very similar results with Locality-aware RR when QoS awareness is not introduced to the MC. This is because most memory latency happens in the NoC. When DSP memory traffic suffers from too much delay in the network, the MC has limited capability in reducing memory latency for the DSP. Therefore, it is necessary to integrate QoS awareness into the network to enable end-to-end QoS support.

QoS-aware RMHG policy achieves much lower DSP latency than RMHG. However, latency violations still happen in Mesh network II, because QoS-aware RMHG only prioritizes DSP requests when a router output port has been released by row matching logic. In contrast, both QoS-aware RR and Locality-aware QoS policies limit DSP latency under the given threshold for all cases. In addition to meeting QoS target, Locality-aware QoS also brings considerable

Fig. 12. Summary of DSP memory latency violations. Latency violation is calculated by subtracting maximum latency limit ($0.8 \mu s$) from average memory latency.

Fig. 13. Row-buffer hit counts by Locality-aware QoS policy, normalized to the results by QoS-aware RR policy. A normalized hit count higher than 1 indicates improvement in row-buffer locality.

improvement on row-buffer hit rate. Figure 13 shows the row-buffer hit counts by Locality-aware QoS policy normalized by those from QoS-aware RR. All normalized hit counts are above 1 meaning row-buffer hit rate improvement is seen in every test case. On average, row-buffer hit count is improved by 58.32%. In the best case, the improvement is up to 81.88%.

As discussed in previous section, row-buffer locality improvement helps to reduce average memory latency for all cores and thus facilitates their workload progress. Figure 14 summarizes normalized memory latencies by Locality-aware QoS policy. These results are normalized to those achieved by QoS-aware RR policy. Locality-aware QoS achieves lower latency in all cases, leading to an average latency reduction by 14.45%. As demonstrated in Figures 13 and 14, our

Fig. 14. Average memory latencies by Locality-aware QoS policy, normalized to the results by QoS-aware RR policy. A normalized latency lower than 1 indicates reduction in memory latency.

Fig. 15. Summary of normalized DRAM energy cost (operation energy and background energy) by different allocation policies after one million bytes have been accessed in DRAM. All results are normalized by the energy cost achieved by QoS-aware RR policy.

locality-aware router design is capable of providing QoS support and improving memory efficiency at the same time.

4.4 Energy Saving and Overhead

The energy consumed by DRAM can be split into two parts: operation energy and background energy [6]. Operation energy refers to the energy cost related to the execution of DRAM commands, including activation, precharge, and column read/write, whereas background energy is consumed at a constant rate regardless of DRAM operations, such as the energy cost by phase-locked loop. With more row-buffer hits, fewer activation and precharge operations are needed to finish the same amount of memory accesses, which means less operation energy cost. Furthermore, since memory latency is reduced as a result of increased row-buffer hits, it takes shorter time to serve the same number of memory requests in DRAM, which leads to lower background energy.

Figure 15 summarizes the DRAM energy by Locality-aware QoS policy. The energy costs are normalized to those by QoS-aware RR policy to show the energy reduction introduced by locality-aware forwarding. CPU test case 1 is used for all test cases. We adopt an open source tool named DRAMPower [2] for command trace-based DRAM energy estimation. The memory subsystem is simulated until one million bytes have been accessed in DRAM. The energy cost during this period is shown, including operation energy (Figure 15(a)) and background energy (Figure 15(b)). Compared with QoS-aware RR policy, Locality-aware QoS lowers operation energy by 26.70% and background energy by 29.84% on average, respectively.

The proposed locality-aware router design achieves substantial DRAM energy savings by creating more row-buffer hits and reducing the energy required to serve the same number of memory requests. However, the locality-aware design also adds extra energy cost to router

Fig. 16. Combined energy cost (in DRAM and NoC routers) by Locality-aware QoS policy in locality-aware routers after one million bytes have been accessed in DRAM. All results are normalized by the combined energy cost by QoS-aware RR policy applied in canonical routers.

Fig. 17. Row-index cache hit rate and average row-buffer hits per activation over different numbers of cache ways in row-index caches.

microarchitecture. To estimate the energy overhead in NoC routers, we use DSENT [31] with a 45nm technology node library. We extend the power model in DSENT to include row-index caches. Compared with a canonical NoC router, the locality-aware design increases area and power by 11.54% and 12.10%, respectively. Nonetheless, given that DRAM energy consumption is far higher than the energy consumption of an NoC router, we still see a substantial net improvement in energy savings. In Figure 16, we show the combined energy cost in DRAM and NoC routers when locality-aware RR policy is applied in our locality-aware NoC routers. The results are normalized by the combined energy cost when the QoS-aware RR policy is applied in canonical NoC routers. Although locality-aware NoC routers incur some energy overhead, the DRAM energy savings are substantially more. Therefore, we still see a substantial net improvement in energy savings when considering both DRAM and NoC energy. As shown in Figure 16, the combined energy cost by locality-aware RR still shows an average net reduction by 27.82%.

4.5 Sensitivity Analysis

To have a better understanding of row-index caches, we further explore impacts of row-index cache size and address mapping scheme on the efficacy of the proposed locality-aware router design.

4.5.1 Row-Index Cache Size. First, we vary the size of row-index caches by adjusting the number of cache ways per set from 1 to 32. Figure 17 shows row-index cache hit rate and average row-buffer hits per activation in DRAM under varying cache size. Mesh network I and Asymmetric network I are used for the evaluation. Locality-aware RR policy is applied in network routers. As shown in Figure 17(a), cache hit rate grows gradually with the increase in cache size for all

Fig. 18. Average row-buffer hits per row activation with respect to different address mapping schemes.

test cases regardless of network topology. However, in Figure 17(b) the numbers of row-buffer hits behave differently on different networks.

To understand the results in Figure 17(b), we need to analyze the impact of row-index cache size on row-buffer locality preservation. Specifically, the size of row-index caches influences locality preservation in two ways. First, a larger size allows to record more recent requests and thus helps to capture more temporal locality in network traffic. However, more cache storage does not always help, because when there are multiple traffic streams with high memory locality, they may all end up with cache hits if the cache is large enough to “remember” all of them. As a result, none of the competing memory streams get prioritized over the others, and Locality-aware RR policy regresses to RR policy. Therefore, the performance of Locality-aware RR policy degrades if we have more cache storage than necessary.

The optimal size of row-buffer index caches is determined by the layout of heterogeneous cores and the network topology. In the case of Mesh network I, the average number of row-buffer hits achieves its peak when row-index caches have four-way cache sets, whereas for Asymmetric network I, the best performance is achieved with one cache line per set. The major difference between these two networks is that heterogeneous cores are more evenly distributed in Mesh network I, resulting in more locality dilution. Slightly increasing cache size helps to recover the diluted locality. However, Asymmetric network I places multimedia cores close to each other. While these multimedia cores issue memory streams with high locality to the same routers, increasing cache size in these routers will quickly degrade locality-aware forwarding to round-robin arbitration.

4.5.2 Address Mapping Scheme. So far we have assumed the address mapping scheme depicted in Table 1. However, more sophisticated address mapping schemes are often used to balance bank-level parallelism and row-buffer locality. Such schemes interleave column bits with bank bits so that memory streams with high locality will not only access a single row and preempt others from accessing the same bank. Therefore, we test three different address mapping schemes as shown in Figure 18. CPU test case 1 is applied in all cases.

Address mapping is a process of breaking down a physical address into a series of indices for ranks, banks, rows, and columns. For example, in Figure 18, $rank[0:0]$ indicates a 1-bit index for rank, while $col[9:0]$ represents a 10-bit column index. In Figure 18(a), the same scheme as in Table 1 is applied. In this case the bank index is not interleaved with the column index, and the column index is mapped to the lowest bits in physical address to maximize row-buffer locality. In

both Figure 18(b) and (c), the bank index and the column index are interleaved to increase bank-level parallelism at the expense of row-buffer locality. Additionally, in Figure 18(c) the bank index is mapped to lower bits than in Figure 18(b) to achieve higher parallelism.

Among the evaluated network scheduling policies, RR shows trivial changes under different mapping schemes, because this policy is oblivious of DRAM microarchitecture. The difference between RR and DSPK in terms of row-buffer hits diminishes quickly as the bank index moves to lower physical address bits. As for RMHG and Locality-aware RR policies, they achieve similar numbers of row-buffer hits regardless of address mapping schemes. Although their improvements on row-buffer hits are reduced as the bank index moves to lower bits, both of them outperform RR by more than 40% in all cases as shown in Figure 18(c).

Alternatively, we can improve bank-level parallelism by simply adding more banks to the memory subsystem. Our evaluations so far have been based on 16 memory banks (2 ranks and 8 banks per rank). In Figure 18(d), we double the number of banks to 32 in total without expanding memory space (by incrementing bank index bitwidth and decrementing row index bitwidth). In comparison to Figure 18(a), row-buffer hits by both Locality-aware RR and RMHG policies diminish in Figure 18(d) with more banks available. However, we still see at least 45% more row-buffer hits being enabled by Locality-aware RR. On average, Locality-aware RR and RMHG policies increase row-buffer hits by 77.47% and 76.91%, respectively.

5 RELATED WORK

QoS-aware Networks-on-Chip: Globally Synchronized Frames (GSF) [20] is one of the earliest approaches that adopts a frame-based scheduling scheme to provide throughput guarantees to network traffic. In GSF, time is coarsely quantized into frames and each flow can reserve a fraction in the frame to inject data flits. Frames are prioritized according to their ages and the flits belonging to older frames are chosen over those in younger frames. The limitation of GSF is twofold: the hardware complexity due to the recycling of frames and source buffering and the coarse granularity of global scheduling. To overcome the limitations of GSF, LOFT scheme [23] localizes the frame-based scheduling. In LOFT, each output link of a router manages frame recycles its own frames independently, which enables more flexibility and less hardware cost. Preemptive Virtual Clock (PVC) [9] takes a different approach to circumvent the issues in GSF. In PVC, frames are defined as a fixed number of cycles so that no frame recycling is needed. Moreover, each router maintains a table of bandwidth counters for network traffic flows. Rate-based scheduling is performed at each router in reference to the bandwidth counters. Aéria [5] is a slack-aware NoC architecture that quantizes packet slacks at injection and prioritizes urgent packets with small slacks during VC/switch allocations. To prevent starvation, batching mechanism similar to the framing in Reference [9] is adopted to sort packets into batches that are prioritized based on their age. The OSCAR scheme [36] was recently proposed for CPU-GPU systems. To accommodate the vastly different traffic from the CPU and the GPU, the authors deployed asynchronous batch scheduling. Every router bundles packets into mini batches, inside which specific provisions are allocated to the CPU and the GPU. Packets from older batches are scheduled ahead of younger batches. Inside a batch, packets are reordered so that higher priority is given to CPU requests and read requests. Although priority-based arbitration is commonly used to implement QoS support in NoC routers, weighted round-robin has also been proposed for QoS-aware scheduling [10]. In Reference [8], the scalability bottleneck of richly connected low-diameter networks is analyzed. The authors pointed out that QoS-related buffer cost scales poorly with network radix. To reduce the substantial QoS overheads, a hybrid NoC architecture is proposed, in which shared resources are consolidated within a portion of the network and QoS support is only implemented within the subnetworks containing shared resources.

Memory-aware Networks-on-Chip: One of the earliest studies on row-buffer locality dilution issue was presented in Reference [35]. The authors proposed two policies for VC allocation to preserve row-buffer locality while scheduling memory traffic from multiple GPU shaders. Hold-Grant policy holds the highest priority for the previous winner, whereas Row-Matching Hold-Grant policy holds the grant for the previous winner as long as the row index of the new request matches with its predecessor's. To reduce hardware overhead, the authors also proposed Hash-Matching Hold-Grant policy, which stores and compares row indices by their hash codes. These policies were proven to help a simple in-order memory scheduler to achieve much higher performance. A DRAM-aware flow control protocol was proposed in Reference [12], assuming the memory controller does not possess out-of-order scheduling capability or request buffers. In the proposed protocol, VC allocators compare the destinations of buffered packets with the last scheduled packet. The priority level of a packet is evaluated based on the hypothetical memory scheduling interval after the previous packet. The packet that goes to the same row with its predecessor needs the least scheduling time interval and thus achieves the highest priority. This way, packets coming from the same source and to the same row in DRAM naturally form into batches along the route. The authors in Reference [24] introduced the Dual Scheduled Packet (DSPK) framework, which includes two types of routers. The first type of router (DSPK-I) deals with core-to-core communication traffic and prioritizes latency-sensitive packets according to their L1 cache miss rates and memory-level parallelism. The second type of router (DSPK-II) is particularly designed for the traffic to off-chip memory. A DSPK-II router collects bank state information and prioritizes memory packets that are beneficial to DRAM efficiency. Since a DSPK-II router requires real-time information on bank states, it has to be placed near the memory controller. In the proposed framework, DSPK-II routers are used within a one-hop radius of the memory controller. A packet-coalescing NoC router was proposed in Reference [3] to improve memory locality on GPGPU platform. The proposed router design coalesces packets for the same DRAM row into a large packet. Specifically, when a new packet arrives at a router, it will be held for a certain number of cycles to wait for others going to the same row. When the hold time limit has been reached, collected packets will be coalesced into a single large packet and then scheduled to the next router. Similarly to RMHG, packet-coalescing ignores end-to-end QoS and hinders the performance of latency-sensitive cores by adding constant coalescing delays at each hop.

Among extant memory-aware network designs, RMHG [35] achieves the highest memory efficiency with the least complexity, because the design only requires a register per port to record row index (or hash) of the previous request and simple comparator logic to perform row matching. However, based on our evaluations in Section 4.5.1, RMHG may suffer from the lack of "memory" issue on a network with numerous traffic flows and arbitration stages. It is also inflexible to be incorporated into other prioritization schemes such as QoS awareness. DRAM-aware router design in Reference [12] is very similar to RMHG except that it takes into account more DRAM timing constraints. This can be an advantage in a memory system with high bank-level parallelism and write-to-read turnaround penalty. Nonetheless, it suffers from the same limitations as RMHG's. DSPK-II routers [24] are only intended to be implemented close to the memory controller. Evaluation results have shown that it is far less effective than RMHG due to the limited memory visibility. By comparison, the proposed locality-aware router design achieves the same level of memory efficiency as RMHG without sacrificing the flexibility to integrate other prioritization schemes. Moreover, row-buffer size in the proposed design can be tuned according to the severity of network traffic interference to achieve the optimal performance.

Memory Visibility Extension: Besides NoC designs, there have also been many research about increasing the visibility for the memory scheduler to improve DRAM efficiency. As the stage right before the memory controller, the last-level cache is often seen as an opportunity for visibility

extension. One of the earliest DRAM-aware last-level cache replacement policies was introduced in Reference [19]. The authors proposed to evict cache lines that are likely to become row hits in DRAM or exploit bank-level parallelism when they are brought into the cache again later. Virtual Write Queue (VWQ) [30] was designed to coordinate memory scheduling with cache writebacks. As an extension to the write queue in the memory controller, the LRU cache lines in the last-level cache are visible to the memory controller and available for memory scheduling along with write requests. Writebacks are triggered when the target row of these dirty LRU cache lines is being accessed so that these scheduled writebacks will not cause extra row activations in DRAM. Last-Write Predictor Guided (LWPG) writeback policy was proposed in Reference [34]. The authors defined last-write blocks as cache blocks that will not be written again before being evicted to memory. A low overhead last-write predictor keeps track of last-write blocks that are available for memory scheduling without incurring extra memory write requests. Eager Read/Write Clustering (ERWC) [13] further reduces the number of row activations by clustering both write and read requests with cache writebacks to the same row-buffer. Eager writebacks are triggered by row activations regardless of the cause. Recently a unified memory controller [27] was proposed to orchestrate cache request arbiter with memory scheduler so that potential row-buffer hits can be harvested during cache request arbitration.

QoS-aware Memory Scheduling: Providing QoS support in the memory controller has been vastly investigated in the past decade. Parallelism-aware batch scheduling [21] was proposed to balance between thread-level fairness and DRAM efficiency. Memory requests are formed into batches that are served in time order. Within a single batch, memory requests can be re-ordered to improve row-buffer hit rate and bank-level parallelism. ATLAS [17] was introduced to prioritize applications with low memory intensity to improve system throughput. Furthermore, thread cluster memory scheduling [18] dynamically clusters applications into low and high memory-intensity clusters and improves system performance and fairness at the same time. A parallel application memory scheduling scheme [7] was later introduced to handle the inter-thread memory interference within a parallel application. Staged memory scheduler [1] was the first application-aware scheduler design proposed for CPU-GPU systems. The proposed scheduler architecture consists of separate queuing stages for memory transactions and DRAM commands. A QoS-aware scheduler is located between two queuing stages. The single-tier virtual queuing memory controller [29] overcomes the inefficacy of two-tier controllers in QoS-aware scheduling. Virtual queues are adopted to perform per-source per-bank queuing once memory requests are received by transaction buffers. QoS-aware scheduling is performed as the last step in the memory controller to avoid unwanted latency between the scheduler and DRAM. In Reference [14], a QoS-aware scheduling policy was proposed to dynamically balance CPU and GPU bandwidth. The proposed policy tracks GPU workload progress and allocates sufficient bandwidth to the GPU to achieve the target frame rate. Meanwhile best-effort service is provided to the CPU. Recently, a deadline-aware scheduler [32] extends the concept of frame progress tracking and takes into account more multimedia cores with target frame rates while improving CPU performance. Memory controller designs for new storage technology other than DRAM have also raised attention. For example, a fair and high-performance memory control scheme for persistent memory-based system that runs both persistent and non-persistent applications was introduced in Reference [37].

6 CONCLUSIONS

In this article, we show the deficiency of heterogeneous MPSoCs in memory efficiency due to memory locality dilution. Previous work have studied memory visibility extension, while the preservation of memory locality in the NoC remains to be explored. In response to the challenge, we proposed a NoC router design with locality-aware packet forwarding. Row-index caches

are introduced to track memory locality in recent packets. Locality-aware forwarding can be implemented on basis of priority arbiters. It is compatible with extant QoS-aware router designs. In our evaluations, the proposed design is proven to be more effective in improving memory efficiency compared with prior memory-aware approaches, while providing QoS support. Compared with QoS-aware routing without special care to row-buffer locality, the proposed design greatly improves row-buffer hits by 58.32% and reduces memory latency by 14.45% on average. Considering both energy overhead and savings, the locality-aware router brings about an average net reduction in energy cost by 27.82%.

REFERENCES

- [1] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 416–427. <http://dl.acm.org/citation.cfm?id=2337159.2337207>.
- [2] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. 2014. DRAMPower: Open-source DRAM Power and Energy Estimation Tool. Retrieved from <http://www.drampower.info>.
- [3] Chien-Ting Chen, Yoshi Shih-Chieh Huang, Yuan-Ying Chang, Chiao-Yun Tu, Chung-Ta King, Tai-Yuan Wang, Janche Sang, and Ming-Hua Li. 2014. Designing coalescing network-on-chip for efficient memory accesses of GPGPUs. In *Proceedings of the International Conference on Network and Parallel Computing*, Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura (Eds.). Springer, Berlin, 169–180.
- [4] William Dally and Brian Towles. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA.
- [5] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2010. Aergia: Exploiting packet latency slack in on-chip networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 106–116. DOI: <https://doi.org/10.1145/1815961.1815976>
- [6] Howard David, Chris Fallin, Eugene Gorbato, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC'11)*. ACM, New York, NY, 31–40. DOI: <https://doi.org/10.1145/1998582.1998590>
- [7] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt. 2011. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 362–373. DOI: <https://doi.org/10.1145/2155620.2155663>
- [8] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. 2011. Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 401–412. DOI: <https://doi.org/10.1145/2000064.2000112>
- [9] Boris Grot, Stephen W. Keckler, and Onur Mutlu. 2009. Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 268–279. DOI: <https://doi.org/10.1145/1669112.1669149>
- [10] Jan Heibwolf, Ralf KÄünig, and Jurgen Becker. 2012. A scalable NoC router design providing QoS support using weighted round robin scheduling. In *Proceedings of the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications*. 625–632. DOI: <https://doi.org/10.1109/ISPA.2012.93>
- [11] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, San Francisco, CA.
- [12] Wooyoung Jang and David Z. Pan. 2009. An SDRAM-aware router for networks-on-chip. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*. ACM, New York, NY, 800–805. DOI: <https://doi.org/10.1145/1629911.1630117>
- [13] Myeongjae Jeon, Conglong Li, Alan L. Cox, and Scott Rixner. 2013. Reducing DRAM row activations with eager read/write clustering. *ACM Trans. Archit. Code Optim.* 10, 4, Article 43 (Dec. 2013), 25 pages. DOI: <https://doi.org/10.1145/2555289.2555300>
- [14] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 850–855. DOI: <https://doi.org/10.1145/2228360.2228513>
- [15] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 24–35. DOI: <https://doi.org/10.1145/2155620.2155624>

- [16] Sukki Kim, Soontae Kim, and Yebin Lee. 2012. DRAM power-aware rank scheduling. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'12)*. ACM, New York, NY, 397–402. DOI : <https://doi.org/10.1145/2333660.2333751>
- [17] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. 1–12. DOI : <https://doi.org/10.1109/HPCA.2010.5416658>
- [18] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE Computer Society, Washington, DC, 65–76. DOI : <https://doi.org/10.1109/MICRO.2010.51>
- [19] Chang Joo Lee, Eiman Ebrahimi, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. 2010. DRAM-aware last-level cache replacement. High Performance Systems Technical Report TR-HPS-2010-007. University of Texas, Austin, TX.
- [20] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. 2008. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Washington, DC, 89–100. DOI : <https://doi.org/10.1109/ISCA.2008.31>
- [21] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, 63–74.
- [22] NVIDIA. 2015. Tegra X1. Retrieved from <http://www.nvidia.com/object/tegra-x1-processor.html>.
- [23] Jin Ouyang and Yuan Xie. 2010. LOFT: A high performance network-on-chip providing quality-of-service support. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 409–420. DOI : <https://doi.org/10.1109/MICRO.2010.21>
- [24] Tejas Pimpalkhute and Sudeep Pasricha. 2014. NoC scheduling for improved application-aware and memory-aware transfers in multi-core systems. In *Proceedings of the 27th International Conference on VLSI Design and the 13th International Conference on Embedded Systems*. 234–239. DOI : <https://doi.org/10.1109/VLSID.2014.47>
- [25] Qualcomm. 2017. Snapdragon 845. Retrieved from <https://www.qualcomm.com/products/snapdragon/processors/845>.
- [26] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 128–138. DOI : <https://doi.org/10.1145/339647.339668>
- [27] Yand Song, Olivier Alavoine, and Bill Lin. 2018. Row-buffer hit harvesting in orchestrated last-level cache and DRAM scheduling for heterogeneous multicore systems. In *Proceedings of the Design, Automation Test in Europe Conference (DATE'18)*. 779–784. DOI : <https://doi.org/10.23919/DATE.2018.8342112>
- [28] Yang Song, Olivier Alavoine, and Bill Lin. 2018. SARA: Self-aware resource allocation for heterogeneous MPSoCs. In *Proceedings of the 55th Annual Design Automation Conference (DAC'18)*. ACM, New York, NY, Article 113, 6 pages. DOI : <https://doi.org/10.1145/3195970.3196110>
- [29] Yang Song, Kambiz Samadi, and Bill Lin. 2016. Single-tier virtual queuing: An efficacious memory controller architecture for MPSoCs with multiple realtime cores. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*. ACM, New York, NY, Article 6, 6 pages. DOI : <https://doi.org/10.1145/2897937.2898093>
- [30] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 72–82. DOI : <https://doi.org/10.1145/1815961.1815972>
- [31] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT—A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the 2012 IEEE/ACM 6th International Symposium on Networks-on-Chip (NOCS'12)*. IEEE Computer Society, Washington, DC, 201–210. DOI : <https://doi.org/10.1109/NOCS.2012.31>
- [32] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Archit. Code Optim.* 12, 4, Article 65 (Jan. 2016), 65:1–65:28 pages.
- [33] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAM-Sim: A memory-system simulator. In *SIGARCH Computer Architecture News*. 100–107.
- [34] Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. 2012. Improving writeback efficiency with decoupled last-write prediction. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 309–320. <http://dl.acm.org/citation.cfm?id=2337159.2337195>
- [35] George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. 2009. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 34–44. DOI : <https://doi.org/10.1145/1669112.1669119>

- [36] Jia Zhan, Onur Kayiran, Gabriel H. Loh, C. R. Das, and Yuan Xie. 2016. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. DOI : <https://doi.org/10.1109/MICRO.2016.7783731>
- [37] Jishen Zhao, Onur Mutlu, and Yuan Xie. 2014. FIRM: Fair and high-performance memory control for persistent memory systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 153–165. DOI : <https://doi.org/10.1109/MICRO.2014.47>

Received December 2018; revised December 2019; accepted December 2019