An Application-Aware Heterogeneous Prioritization Framework for NoC based Chip Multiprocessors

Tejasi Pimpalkhute, Sudeep Pasricha
Department of Electrical and Computer Engineering
Colorado State University, Fort Collins, CO, U.S.A.
tejasi@rams.colostate.edu, sudeep@colostate.edu

Abstract
In chip multiprocessor (CMP) systems with multi-application workloads, communication and memory access both play an important role in influencing system performance. Intelligently prioritizing network packets and memory requests can notably improve system throughput. But with increasing workload diversity in CMPs, applying the same request prioritization rules across the chip can lead to sub-optimal results. In this paper, we propose a novel heterogeneous prioritization framework for CMPs in which two different packet prioritization approaches are proposed and applied to network-on-chip (NoC) routers. A new ranking scheme for classifying an application's criticality is also proposed. We evaluate our framework using a detailed cycle-accurate full system event-driven simulator. Our experimental results show that the proposed framework outperforms fair prioritization techniques by up to 12.6% as well as other application-specific techniques from prior literature by up to 6.9% for various multi-application workloads.

Keywords
Network-on-chip, network packet prioritization, memory level parallelism, memory-aware prioritization

1. Introduction
Emerging electronic chips are becoming increasingly complex with highly parallel processing architectures and tens to hundreds of components integrated on a single die. Network-on-chip (NoC) architectures have emerged as the most promising interconnection fabric for such chip multi-processor (CMP) systems. With the number of cores on a chip growing due to the benefits of shrinking nanometer technology, a lot of pressure is being put on the NoC fabric to handle communication flows efficiently in a system executing multi-programmed (i.e., multi-application) workloads. Much research has therefore been lately focusing on techniques to optimize NoCs for maximizing performance. Further, due to the well-publicized memory wall between processors and main memory, new techniques are needed to overcome mismatches between processor and memory performance. Given the increasing resource and workload heterogeneity in CMP designs, techniques to manage NoC and memory subsystems must exploit heterogeneity in application characteristics [1]-[2], NoC utilization [3], and memory level parallelism (MLP) [4]-[6].

Importance of Packet Scheduling: The performance of applications is increasingly impacted by communication and memory performance [30]. In particular, the throughput of an application is dependent on how fast its requested packets return to the core. It is very interesting to study the journey of a request packet originating at a core and back to the core, and how different parameters affect its end-to-end latency. When a request originates at a core, it first encounters the L1 cache where a miss may cause it to be sent to the next level in the memory hierarchy. In a CMP with a directory-based cache coherence protocol, the next level of the cache is the L2 cache, with L2 banks distributed across processing nodes. Requests often traverse the NoC to access data in remote L2 banks. The latency in these cases varies due to different hop distances and runtime NoC contention. Further, if a request encounters a miss at an L2 cache bank, it typically has to traverse the NoC again to reach a memory controller and then access main memory. Once a request reaches the memory controller, it can face delays due to factors such as bank conflicts, bus busy, etc. Thus, there are several factors contributing to the end-to-end latency of a request. At every step, a request packet experiences slowdown due to interference from other packets and contention. Hence, *it is critical to prioritize packets so that in situations of contention where several packets are competing for bandwidth, requests that are most crucial to the overall system throughput can proceed before other requests.*

Importance of Memory Level Parallelism (MLP): Modern memories are no longer monolithic structures but rather multi-bank architectures that can operate in parallel. Widely used SDRAMs use 3 main commands: precharge (PRE), activate (ACT), and Read/Write (R/W) [6]. For any request, the SDRAM memory controller needs to know its bank address, row address, and column address. When a request arrives at the controller, it checks if the bank being requested is already active, and if not it issues an ACT command. The desired row is then copied into a row buffer to access the requested address. Once the data is fetched, a PRE command is issued to restore the row in the row buffer and idle the bank depending on whether the memory implements a closed-page or open-page policy. Most memories implement an open-page policy to encourage row hits and minimize ACT and PRE commands. Another facet of SDRAMs is that while one bank is busy another bank can still be activated and accessed under certain conditions (called bank level parallelism). It is also desirable for requests approaching memory to do so in a manner that avoids consecutive requests to different rows of the same bank, to reduce memory latency. Typically, *exploiting bank-level parallelism and row locality can enable huge performance gains for memory accesses* [4].

Importance of Network Prioritization: As discussed earlier, it is vital to emphasize the scheduling of network packet requests to enhance system throughput. However, the key question to be answered is: at what point should request prioritization occur? One stage where request prioritization typically occurs in today's systems is in the memory controller. But out-of-order buffers for request scheduling in these controllers take up a lot of area and add notable latency, especially when congestion in the network is high or memory-intensive workloads are running. In contrast, NoC routers have simple FIFO input/output buffers, with multiple requests competing for the same output channel simultaneously. As the NoC becomes more and more congested, it is vital to analyze such requests as early as possible, to uncover their impact on individual workloads as well as system performance and appropriately adjust their priorities. Thus, it is more cost-effective to apply *workload-aware prioritization*

techniques in the NoC closer to the source rather than prioritizing the same requests at the memory controller. At the same time, improving main memory performance necessitates increase in row buffer hits and a reduction in memory bank conflicts. This information is more readily available closest to the memory. Rather than relying on costly re-ordering in the memory controller, *memory-aware prioritization* techniques in NoCs can enable more effective distinctions between requests.

In light of the above observations, in this paper we make the following contributions:

- We introduce the novel concept of heterogeneous multi-staged prioritization in NoCs. We demonstrate that by using different prioritization criteria in two different parts of the NoC, system performance can be notably improved.
- We propose new algorithms to be employed in each of these two stages of prioritization. The first algorithm, for routers closer to their originating cores, prioritizes request packets as per criticality of their applications. The second algorithm, which is for routers in proximity to memory controllers, aims to reduce bank conflict for memory requests as they proceed to the memory controller.
- We propose a new ranking scheme for determining criticality of applications. We evaluate previously proposed ranking metrics and show that our ranking scheme is more accurate.
- We compare our work with a baseline configuration frequently used in CMPs, and some of the best performing recent work on NoC and memory request prioritization, and demonstrate the effectiveness of our proposed staged prioritization technique over these prior efforts.

2. Related Work

Several previous efforts have focused on improving performance of either NoC or memory requests. We summarize relevant and recent works in the area below.

Memory scheduling techniques: Several past efforts [1], [7]-[10] have proposed memory scheduling techniques that modify memory controllers to improve memory access performance. However, these techniques are unaware of the network state. The techniques proposed in [11]-[12] schedule memory requests based on the network state, but assume that information about each core is available in the memory controller. Such an assumption entails a huge communication overhead between the memory controller and cores, and as such these techniques are prohibitive and expensive to implement in practice. In [13], the authors propose to emulate memory controller functionality in routers, with the goal of reducing memory controller reordering overhead, minimizing memory latency, and increasing memory utilization. However, the approach is oblivious of application workload needs, and requires careful programming with memory-specific information (e.g., timing parameters) which reduces its portability, as well as scalability in emerging heterogeneous memory systems [14]. The technique in [15] extends the work in [13] by distinguishing between demand packets and prefetch packets while reordering requests as per memory state, but the fundamental drawbacks mentioned for [13] still remain.

Network prioritization techniques: In [16]-[17], application-aware packet prioritization mechanisms for NoCs are proposed based on amount of NoC traffic [16] and packet slack [17]. However, these works use simplified memory delay models for evaluating their techniques, and are agnostic of main memory access performance. In [17], a complex prediction mechanism is used in NoC routers for predicting L2 cache misses and packet slack. We discuss these ranking metrics and their inefficiencies in later sections. In [18], the authors attempt to balance loads on memory banks and reduce packet latencies by reacting to request packets and response packets differently. However, the prioritization in this approach leads to an increased number of delayed packets thereby rendering the technique ineffective in many situations. This technique is also application oblivious and tries to make latencies of all the packets uniform irrespective of their criticality, which can limit system throughput. In [19] the concept of NoC sub-networks is proposed based on application categories to reduce inter- application interference and increase system throughput. The authors use a fine grain application classification method which suffers from some key drawbacks as will be discussed in section 3.

All these prior works employ homogeneous prioritization at each node. Our paper is similar to these works in that we also focus on improving performance of requests as they journey through the NoC. However, ours is the first work to propose heterogeneous multi-staged NoC prioritization to achieve more aggressive system performance optimization.

3. Characterizing Application Packet Criticality

In multi-programmed CMPs, system throughput is governed by the harmonious execution of multiple applications running together. In such systems, routers face a lot of contention due to request/response packets of different applications interfering with each other for the same output channels. It is very important to distinguish between packets of different applications as opposed to treating all packets equally, to optimize system performance. The problem of categorizing packet criticality has been studied in a few recent works [16][17][19] that propose various classification metrics. These metrics are summarized below:

Packet slack estimation: In [17], the authors compute the *slack* of a request packet to determine a packet's criticality. Slack is defined as the maximum delay a packet can tolerate in the network without affecting performance at its source core. For example, consider the scenario where a processor has two outstanding load misses that result in successive network request packets Pkt1 and Pkt2. If Pkt2 encounters lower NoC latency and returns with data before Pkt1, it cannot commit its corresponding load until Pkt1 returns. Thus Pkt2 has some slack for which it can be delayed without reducing application performance. The slack is used in arbitration decisions to prioritize packets with smaller slack over others. But network contention is not considered during slack estimation which can lead to incorrect values of slack and hence flawed determination of packet criticality.

Network episode height and length estimation: In [19], the authors use network episode length and episode height to perform fine grain classification of applications and their requests. The *episode length* is the number of cycles for which an application injects packets in the network till all requests for that core are satisfied, and *episode height* is the number of outstanding L1 misses during this episode length. The *episode length* and *height* values are an indicator of the memory intensity of the network phases of an application and its packet criticality. To determine episode length, a count of cycles is kept for which the MSHR (Miss Status Handling Registers) queue [20] is occupied. However, MSHR queues remove an entry only when the request is satisfied [21]. Hence, if the request gets delayed in the network due to contention, it will result in longer network episodes. As a result, there is a high probability of incorrectly projecting that the application is more network-intensive with this approach.

Private cache miss count estimation: In [16], the authors use private cache misses per kilo instructions (MPKI) as a ranking metric for determining stall time criticality of an application packet. They form 8 clusters of ranks based on relative MPKI of

all the applications, and the application with the lowest MPKI gets the highest rank. This metric is unaffected by the network state and is a reliable indicator of an application's memory intensiveness. Along with [16], several other application-aware techniques proposed previously [1][10][19] have considered this MPKI metric for classification purposes. However, this approach only gives us a very broad idea about the nature of an application in terms of its latency criticality and does not capture the MLP (Memory Level Parallelism) of an application.

Need for dynamic fine-grain application classification: As applications typically demonstrate behavioral variations over the span of their execution, their network demands and therefore criticality also varies over time. To better understand this phenomenon, we captured the L1 MPKI (misses per thousand instructions in the L1 cache) and number of outstanding requests in the L1 MSHR queue for several applications from the SPLASH-2 and SPEC2K suites. MSHR queues are structures that are typically checked on an L1 cache miss. The current miss is added to the queue only if a miss to the same block as the current miss does not already exist. The greater the MSHR queue occupancy, the larger is the number of blocks demonstrating MLP in an application [21]. This MSHR queue occupancy also indicates the instantaneous network demands of an application. We term the MSHR queue occupancy as *MLP index*. Figure 1(a)-(b) show varying L1 MPKI and MLP index values for the various applications over a period of 5M cycles. An interesting observation is that even though L1 MPKI may be constant over an interval for an application, its MLP index can change significantly during that time (e.g., for *ammp, lucas,* etc). The results demonstrate a need for a dynamic fine-grained classification of applications based on their time-varying network demands and MLP characteristics.

3.1 Our Proposed Criticality Ranking Scheme

Different applications contribute differently to the overall system throughput. If an application requires a lot of its operands to be fetched from memory, it spends most of its time stalling and waiting for the requests to be satisfied by the main memory. Such applications cannot contribute much to the overall system throughput (defined as total number of instructions executed per unit time). We can thus classify such memory-intensive applications as *latency-tolerant* applications that do not impact performance notably even if their packets face a small delay. In contrast, there are some applications that have a very high CPU utilization due to their high compute intensity and require very few operands to be fetched from memory over long periods. Such compute-intensive applications can be classified as *latency-critical*, as they are very crucial for the performance of the system, i.e., any additional wait time for a core executing this application can make the CPU stall which otherwise it could have spent executing instructions.

In our proposed criticality ranking scheme, applications (and their corresponding packets) are classified dynamically into four categories based on their latency criticality and MLP. The pseudo-code below summarizes the classification.

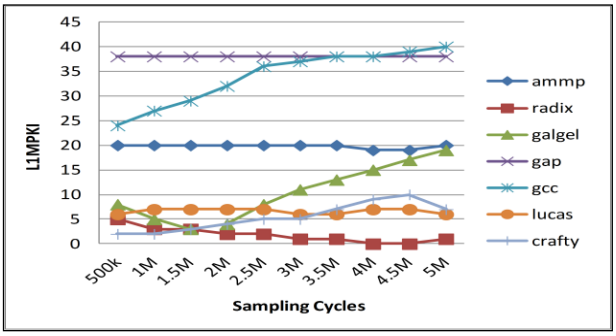
Dynamic application criticality ranking scheme

Counters at core's network interface:
cL1MPKI: counter for average L1 MPKI
cMLPIndex: counter for MLP index

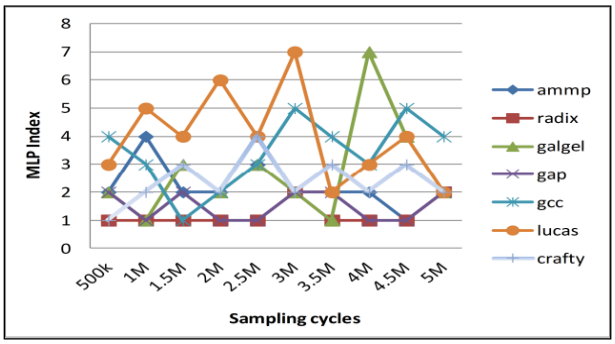
Rank assignment for application packets at core network interface:
At the end of every τ cycles-
case ((cL1MPKI $\leq \tau_0$) AND (cMLPIndex $\leq \tau_1$)) : Rank 0
case ((cL1MPKI $\leq \tau_0$) AND (cMLPIndex $> \tau_1$)) : Rank 1
case ((cL1MPKI $> \tau_0$) AND (cMLPIndex $\leq \tau_1$)) : Rank 2
case ((cL1MPKI $> \tau_0$) AND (cMLPIndex $> \tau_1$)) : Rank 3

cL1MPKI = 0; cMLPIndex = 0 //reset counters

At each core's network interface, a count of L1 cache misses is kept to broadly classify an application as latency-critical or latency-tolerant. To perform fine-grain classification of applications as per their memory access patterns and MLP index exhibited, we also keep a count of entries in the L1 MSHR queue. These counters are queried for the purposes of dynamic classification and then reset at each application profiling interval of τ cycles. At the profiling intervals of τ cycles**,** L1 miss counters and MLP index counters at a core's network interface are checked. If the L1 cache miss count is equal to or below a threshold τ_0 then that application is identified as latency-critical and further checked for its MLP index. Intuitively, in the latency-critical class, applications with MLP index equal to or lower than a certain threshold τ_1 are indicative of multiple row hit requests or high criticality for an application's compute phase, so we give them the highest priority i.e., rank 0. If a latency-critical application has MLP index greater than τ_1 then it comes immediately next in priority because its MLP is likely to enhance the performance of the system. Similarly, if the L1 cache miss count exceeds the threshold τ_0, then that application is put in the latency-tolerant category and further classified depending on its MLP capabilities as mentioned for the latency-critical class. In this manner, our approach performs a fine-grain online classification that captures the changing dynamics of application execution over time.



(a)



(b)

Figure 1: (a) L1 MPKI; (b) number of outstanding requests in L1 MSHR queue over time, for SPLASH-2 and SPEC2K workloads.

4. Heterogeneous Prioritization Framework (HEPI)

Our proposed heterogeneous prioritization (HEPI) framework is inspired by the insight that there is always a race among request packets originating at different cores to reach their destinations. The overall system throughput is impacted by which requests get served faster, and allocation of NoC resources plays a significant role in determining this. Request/response packets also often interfere with each other at NoC routers to acquire an output channel. It is the responsibility of router *arbitration* mechanisms to determine which packets get access to an output channel at every cycle. In Round Robin (RR) based arbitration schemes that are widely used in NoCs, the arbitration is fair but packet

criticality oblivious. Moreover, for off-chip memory requests, when packets reach the memory controller, a great deal of power and latency is expended in re-ordering these requests to optimally access memory in contemporary memory controller nodes.

To overcome both of these limitations, we propose a novel heterogeneous two-stage prioritization technique for packets traversing the NoC. The first stage applies application-aware prioritization, with packets being prioritized as per their criticality, which in turn depends on their latency sensitivity and MLP characteristics. The second stage applies memory-aware prioritization, with packets being reordered as per memory state to minimize stalls. Figure 2 shows how the two types of prioritizations are applied to NoC routers. The routers in close proximity (i.e., at one or less hop distance) from the memory controller employ the second stage prioritization mechanisms while all other routers, relatively farther from memory employ the first stage prioritization mechanisms. As we show later, such a framework can notably improve system throughput and memory access performance.

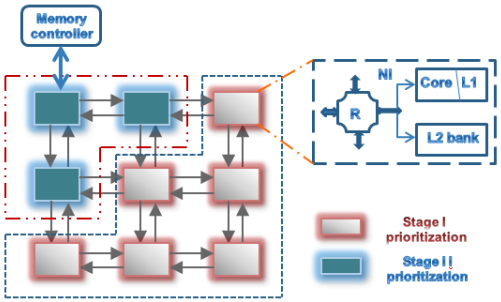


Figure 2: Proposed HEPI Framework

In the following subsections, we first describe a time-based batch prioritization mechanism used in all NoC routers to prevent packet starvation, followed by details of the Stage I application-aware and Stage II memory-aware arbitration algorithms.

4.1 Time Based Batching

We employ a time-based batching technique that is loosely adapted from [16] to prevent starvation of packets in the NoC. The time-based batching technique is summarized below.

Time-based Batching scheme

Parameter Definitions:
β: number of cycles in a batching interval
K : Maximum Batching levels
CBID: Current Batch Injection ID counter
PBID: Packet's tagged Batch ID

Batch assignment for application packets at core network interface:
At the end of every β cycles-
1: CBID = CBID + 1
2: if (CBID >= K)
3: CBID = CBID % K //wraps around for compact ids
4: PBID = CBID

Relative batch priority of packet p at router during arbitration at cycle T:
RBP (p) = (CBID at T – PBID) % K

After each batching interval β, the packets originating in that interval from a network interface are assigned a batch id. This batch id is a timestamp that can be used in routers to determine how long the packet has spent in the NoC. The batch id counter is incremented at each batching interval until K levels, after which it starts over again (steps 1-3) to maintain compact sized ids. The network interface (NI) of each core tags a packets' header flit with a batch id before it is injected into the NoC. When any packet *p* arrives at a NoC router, a relative batch priority of *p* (RBP) is computed based on current batch injection id (CBID) and packet batch id (PBID) for *p*. This RBP value is used to include starvation information as part of the prioritization scheme in both stage I and stage II NoC routers as discussed next.

4.2 Stage I Application-Aware Algorithm

This algorithm prioritizes packets in NoC routers as per their application-specific criticality and also ensures freedom from starvation. We use our novel ranking scheme presented in section 3.1 to determine application-specific packet criticality, and the time-based batching technique (Section 4.1) to prevent starvation of packets. The network interface (NI) of each core uses our proposed ranking scheme to compute application rank, and then tags a packets' header flit with its rank before injecting it into the NoC. The pseudo-code at the end of this subsection summarizes the Stage I algorithm. We discuss this algorithm below.

For every output port *o*, the arbiter checks input ports to see if there are packets that need to go to port *o* (steps 3-4). For an input port *i* with a packet *p* destined for output port *o*, the RBP value of *p* is compared against the value stored in the Maximum Batch Priority (MBP) register (steps 5-15). If RBP for *p* is higher than MBP (step 5), it indicates a starved packet that must be given the highest priority, which is done by updating the *priority_port* register with *p*'s corresponding input port *i* (step 6). The value of MBP is also updated with the priority of the oldest batch (step 7). If RBP is equal to MBP (step 8), prioritization occurs as per packet criticality (steps 9-14). The rank of *p* ($rank_p$) is first compared against the value stored in the *min_rank* register. If $rank_p$ is less than *min_rank*, the value in *min_rank* is updated and *i* becomes the priority port (steps 9-11). If $rank_p$ is equal to *min_rank* (step 12), then we have a case where the priority of two packets destined for *o* is the same. In this case, *p* is prioritised only if it is an L2 packet, as prioritizing such on-chip packets that can finish quickly over off-chip requests improves system throughput (step 13). Finally, if RBP is less than MBP (step 15), it indicates a situation where another packet set the MBP value and is more severely starved; therefore packet *p* at the current input port *i* is not given higher priority (i.e., *priority_port* register is not updated). Once all input ports have been checked, the arbiter grants the priority port access to the output port *o* (steps 18-20). These steps are performed in parallel in stage I NoC routers for all output ports at every cycle, to determine which packets from the input ports can proceed to their destination output ports.

Stage I Application-Aware Algorithm

Parameter Definitions
O: set of NoC router output ports
I: set of NoC router input ports
MBP: Maximum Batch Priority //*priority of oldest batch*
PBP: Packet Batch Priority //*batch priority for current packet*

Prioritization of packets at stage I NoC router:
1: **for all** o: o ∈ O **do**: {
2: Initialize min_rank to 5; priority_port to -1; MBP to -K.
3: **for each** i: i ∈ I **do**: {
4: **if** (packet p ∋ p at i ↦ o) { // *packet p at i going to o*
5: **if** (RBP (p) > MBP) {
6: MBP = RBP (p)
7: priority_port = i
8: } **else if** (RBP (p) == MBP) {
9: **if** ($rank_p$ < min_rank) {
10: min_rank = $rank_p$
11: priority_port = i
12: } **else if** ($rank_p$ == min_rank) {
13: priority_port = (type (p) == L2 packet)? i : priority_port
14: } *// if $rank_p$ > min_rank, do nothing*
15: }*// else if (RBP(p) <MBP), do nothing*
16: }
17: }
18: **if** (!(priority_port == -1)) {
19: priority_port ← grant port *o*
20: }
21: }

4.3 Stage II Memory-Aware Algorithm

This algorithm prioritizes packets in NoC routers so as to avoid bank conflicts and encourage *row hits* and *bank level parallelism* (BLP). One of the major reasons that off-chip memory requests are delayed is due to stalls experienced as a result of bank conflicts. We observed that in addition to prioritizing applications in the network, system performance can be improved if the prioritized requests do not spend time stalling due to such bank conflicts. We thus propose to use the arbitration inherent in NoC routers to prepare DRAM memory requests to proceed to the memory controller in a manner that prevents bank conflicts and relieves the memory controller from costly reordering of requests to minimize bank conflicts and improve row-buffer hits. We impart this memory-awareness to routers in the proximity of a memory controller (Stage II routers; Figure 2). Much like in Stage I routers, the Stage II routers also use a time-based batching technique (Section 4.1) to prevent packet starvation. The pseudo-code below summarizes the Stage II algorithm.

Stage II Memory-Aware Algorithm

Parameter Definitions
O: set of NoC router output ports
I: set of NoC router input ports

Prioritization of packets at stage II NoC router:

```
1:  for all o: o ∈ O do :{
2:    Initialize min_rank to 5; priority_port to -1; MBP to -K.
3:    for each i: i ∈ I do: {
4:     if (packet p ∋ p at i ↦ o) {                  // packet p at i going to o
5:       if (RBP (p) > MBP) {
6:            MBP = RBP (p)
7:            priority_port = i
8:       } else if (RBP (p) == MBP) {
9:            if ((type (p) == DRAM request) {
10:             if ((dest.bank(p) ∉ RUB) ||          // bank level parallelism
                    (dest.{bank, row}(p) ∈ RUB)) { //row hit
11:                   min_rank = rank_p
12:                   priority_port = i
13:             } else if ((dest.bank(p) ∈ RUB) && valid==0)
                      &&(rank_p < min_rank)) {      //bank conflict
14:                   min_rank = rank_p
15:                   priority_port = i
16:             } // else do nothing
17:           } else {                                  // for network packets
18:             if (rank_p < min_rank) {                //packet criticality
19:                   min_rank = rank_p
20:                   priority_port = i
21:             } // if rank_p ≥ min_rank, do nothing
22:           }
23:       } // if RBP < MBP do nothing
24:     }
25:   }
26:   if (!(priority_port == -1)) {
27:       priority_port ← grant port o
28:   }
29: }
```

Stage II routers share a small *recently used bank (RUB) table* with information about the main memory state that includes the N distinct recently used banks, their corresponding rows, and a valid bit to indicate if the corresponding request has been processed. A valid bit value of 0 for an entry indicates that the bank is free whereas 1 indicates that the bank is busy servicing the request. For every output port o, the arbiter checks input ports to see if there are packets that need to go to port o (steps 3-4). For an input port i with a packet p destined for output port o, we first check if the packet has been starved by applying an RBP check as we did for the stage I prioritization algorithm (steps 5-7), to ensure that no packet suffers from starvation. If packets of the same batch are competing with each other (step 8), then memory packets (i.e. DRAM requests) and network packets (e.g., DRAM responses, shared cache requests/responses or cache coherence data) are handled separately. For a DRAM request, the router queries the RUB table and prioritizes packet p under the following conditions (steps 10-12): i) if the entry for p's destination bank is absent in RUB; or ii) if p's destination bank is present in RUB and matches the corresponding row (valid or invalid). The first condition enhances bank level parallelism by accessing multiple banks in parallel; the second condition indicates a row hit. If these conditions are not satisfied, then if a more critical packet (i.e. with lower rank than min_rank) is encountered, it is prioritized if the bank's RUB entry has been invalidated, which indicates that the bank is not busy (steps 13-15). These conditions ensure freedom from stalls at the DRAM. The priority_port and min_rank registers are updated with the packet's i and $rank_p$ values if the above conditions are satisfied.

If none of the above conditions are satisfied by memory packets at input ports, then the arbiter prioritizes network packets even if they have higher ranks (lower priority) than memory packets. By de-prioritizing the bank conflict/row conflict memory packets in this manner, head of queue stalls at the memory controller are avoided. The network packets are prioritized based on their ranks (steps 17-20) compared with the value in min_rank. The input port of the highest priority (lowest rank) packet is stored in the priority_port register (step 20). After iterating through all input ports, the output port o is granted to the priority_port (steps 26-28). These steps are performed in parallel for all output ports in stage II routers at every cycle, to determine packets from input ports that can proceed to their output ports.

All stage II routers have read access to the shared RUB table; however, only the router directly connected to the memory controller node has write access to the table. When a DRAM request wins the arbitration in this router, the RUB table is updated with the request bank and row entries, and the valid bit for the entry is set to 1. If the bank entry for the winner is already present in the RUB table, the row entry is overwritten and the valid bit is set to 1. The memory controller sends notifications to this router when a bank becomes free; following which this router invalidates the bank entry by setting its valid bit to 0. If the table is full, the oldest bank entry with a valid bit of 0 is overwritten with the new request. The size of the table is kept small to allow for fast table lookup and avoid delays during arbitration.

Table 1: Baseline CMP system configuration

CPU configuration	1 GHz; out of order
L1 Cache	I/D-cache 16 KB, 2-way, 2-cycle latency, cacheline 128B, 16 MSHRs
L2 Cache	Unified, 128kB bank shared, 4-way set associative, cacheline 128B
Main Memory	2GB, DDR2-667, 2 ranks/DIMM, 8 banks per rank, detailed memory model using open-row policy and row-interleaving
Router	5-stage Virtual Channel Router; Credit-based flow control, 4 VCs per port, 4 buffers/data virtual channel, 1 buffer/ctrl virtual channel
Network Topology	3×3 2D Concentrated Mesh, Concentration factor 4
Routing scheme	Deterministic X-Y

5. Experiments

5.1 Experimental Setup

Evaluation Platform: We use the cycle-accurate event-driven GEM5 full system simulator [22] for validating our proposed prioritization framework. GEM5 is a full system simulator providing support for state-of-the-art out-of-order processor cores as well as models for a detailed NoC architecture, cache hierarchy, and main memory subsystem [23]. We use the directory-based MESI coherence protocol as our default coherence protocol for the on-chip cache hierarchy. Table 1 shows the configuration of our baseline CMP consisting of a 3×3 concentrated mesh NoC with a concentration factor of 4

(i.e., 4 cores connected to each router) for a total of 36 cores on the die. We assume one-to-one application to core mapping with requests from a core allowed to access L2 cache banks at remote cores. Each NoC router has a state-of-the-art 5-stage pipelined implementation. For our baseline CMP, we use one memory controller connected to node 0 and assume a NUMA-based system to support scalability. The baseline memory controller serves requests in a first come first served manner.

HEPI Implementation: For our proposed heterogeneous prioritization (HEPI) framework, the network interface for a core requires counters to keep track of a core's L1 misses and outstanding requests in the MSHR queue at each ranking interval. We empirically choose the ranking interval as τ =100K cycles considering the tradeoff between ranking accuracy and its overhead. The network interface has the straightforward logic to compute ranks of applications over a ranking interval, as discussed in Section 3.1, and assigns a 2 bit rank to the header flit of each packet for prioritization in downstream routers. Unlike prior work such as STC [16] where at each predefined interval the L1 MPKI values of each core are communicated to a centralized decision logic (CDL) to rank applications, our approach is more scalable and possesses lower overhead as ranking decisions are made locally. Based on the application profiling done for SPEC2K and SPLASH2 benchmarks, the median value of average L1 MPKI was found to be 15 whereas MLP Index was found to be 3. Hence, we set the classification threshold τ_0 as 15 and classification threshold τ_1 as 3. The batching interval value (β) was empirically set to 16K cycles and the maximum batching levels (K) set to 8. Finally, for stage II routers, we chose the RUB table size (N) as 8 entries per rank. This table size is small enough to allow for fast lookup, while also providing sufficient visibility into recently used banks given that modern DRAMs typically possess 8 banks per rank.

Comparison with Prior Work: We modeled our proposed HEPI framework as well as the best known prior works on NoC and memory prioritization for comparison. The prior works and configurations we compare against include: 1) a baseline localized round robin technique (Local RR) that uses a fair round-robin prioritization algorithm in all NoC routers; 2) SDRAM-aware router [13] that uses SDRAM-specific timing parameters at specific SDRAM-aware routers (SDAR) to optimize delay and priority of an off-chip memory request; and 3) Stall Time Criticality (STC) [16] that employs application-aware prioritization based on batching and application ranking as per L1 MPKI at all the NoC routers. The parameters used for STC are 1) batching interval=16K cycles; 2) ranking interval= 350K cycles; 3) batching as well as ranking levels=8. For SDAR, we use DDR2-667 timing parameters from Micron datasheet [24] and replace three conventional NoC routers in the vicinity of the memory controller with SDRAM-aware routers. These prior techniques have used a trace-based simulator for their evaluation purposes while we use an event-driven simulator with detailed micro-architecture models. All the simulations in our studies ran for at least 180M instructions.

Table 2: SPLASH-2 and SPEC2K classifications

Compute-Intensive	Memory-Intensive
ammp, apsi, gap, galgel, gcc, mgrid, ocean	barnes, crafty, fft, fmm, lucas, radix, sixtrack, swim, equake, applu

Workloads: We profiled 17 diverse applications from the SPLASH-2 and SPEC2K benchmark suites and divided them up into two categories based on their compute- and memory-intensity as shown in Table 2. We used different combinations of these applications to form multi-programmed workloads to compare HEPI with the baseline system and prior work.

Evaluation Metrics: We focus on three major performance metrics: overall system throughput, average memory latency and weighted speedup. We define overall system throughput as the number of instructions executed in the entire system over a duration of time (in terms of number of cycles). Weighted speedup [25] is defined as $\sum_i \frac{\text{IPCmp}_i}{\text{IPCsp}_i}$ over all applications, where IPC_{mp} and IPC_{sp} are the instructions executed per cycle when running multiple workloads and running a single workload, respectively. This metric is useful in that it captures inter-application interference. Average memory latency is measured from the time a request enters a memory controller till it is serviced by DRAM and enqueued back in the controller.

5.2 Experimental Results

This section presents experimental results that compare our proposed HEPI framework with the following prioritization techniques: 1) baseline Local RR; 2) SDAR [13]; and 3) STC [16]. Modern CMP systems are often multi-programmed, executing either heterogeneous workloads (a mix of compute- and memory-intensive) or homogeneous workloads (only compute-intensive or only memory-intensive). Hence, we perform experiments for each of these workload cases. Table 3 shows how various benchmarks are combined to create the three workload cases. The number next to a benchmark corresponds to its parallelization degree (number of cores it runs on).

In the following section (Section 5.2.1), results for system throughput, memory latency and weighted speedup for the comparison against prior prioritization frameworks are presented across the three workload categories. We show results for the scalability of our framework for three different network sizes with varying number of cores in Section 5.2.2. Finally, we compare the overhead of our proposed approach and other techniques in Section 5.2.3.

Table 3: Three workload cases for a 36 core CMP

Heterogeneous Workloads (Case I)	ocean(4), gcc(8), apsi(6), swim(8), equake(10)
	ammp(5), apsi(9), radix(10), gap(8),lucas(4)
	Crafty(7), gap(9), applu(11), gcc(4), barnes(5)
Compute-Intensive Workloads (Case II)	lucas(12),barnes(12), radix(12)
	fft(9), swim(9), barnes(9), fmm(9)
	equake(12), crafty(12), applu(12)
Memory-Intensive Workloads (Case III)	ocean(9), apsi(9), mgrid(9), gcc(9)
	gcc(16), ammp(8), galgel(12)
	apsi(9), gap(9), mgrid(9), gcc(9)

5.2.1 Performance comparison across workload types

Figure 3 shows the comparison between HEPI and other techniques, with averaged results shown for the three workload cases shown in Table 3. The numbers over the bars in the figures indicate percentage improvement of the techniques over the baseline Local RR technique. For case I heterogeneous workloads, HEPI shows an improvement over the baseline/ SDAR/STC techniques of 8.4%/4.6%/5.9% for system throughput, 7.3%/3.3%/5.4% for memory latency, and 9.3%/4.9%/6.1% for weighted speedup, motivating the need for packet prioritization over fairness. The SDAR technique supports memory-aware packet prioritization closer to the memory controller but is oblivious to application-specific packet criticality. Hence, it does not contribute towards improving the system throughput as effectively as the multi-stage HEPI technique does. In contrast, STC enables application-specific packet prioritization but relies on L1 MPKI only to determine criticality of a packet and does not consider its memory access characteristics. Also, it is inefficient in handling DRAM requests as it is memory-unaware. Hence, it is outperformed by HEPI which not only exploits the unique characteristics of each application to determine packet criticality and better prioritize them upstream in the network via stage I routers, but also

maximizes row hits and prevents bank-conflicts downstream in the network (closer to the memory) via stage II routers.

For case II compute-intensive workloads, we can observe that HEPI again outperforms the other techniques, but by a smaller magnitude than for heterogeneous workloads. HEPI shows an average improvement of 6.1%/2.9%/3.9% for system throughput, 4.7%2.8%/3.7% for memory latency and 6.5%/ 3.2%/4% for weighted speedup over the baseline/SDAR/STC techniques. The smaller magnitude of improvement for HEPI with compute-intensive workloads in contrast to heterogeneous workloads is because for compute intensive workloads, network traffic is much lower, which reduces opportunities for applying prioritization and consequently diminishes the benefits of using the HEPI technique. Moreover, as all the workloads have similar behavior, there is little scope for distinguishing between applications in this case.

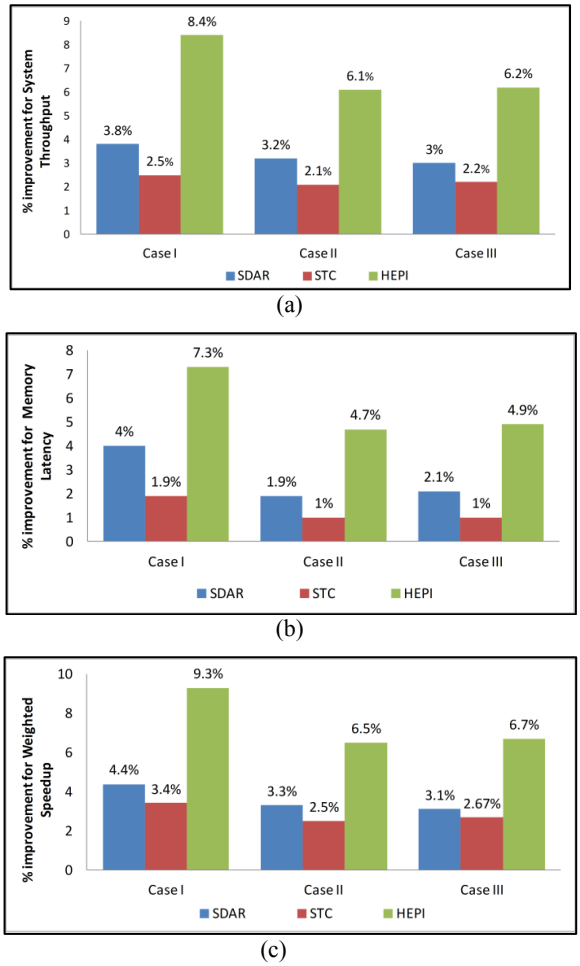


Figure 3: % improvement over Local RR: (a) system throughput (b) average memory latency (c) weighted speedup

Finally, for case III memory-intensive workloads, we see an average improvement of 6.2%/3.2%/4% for system throughput, 4.9%/2.8%/3.9% for memory latency and 6.7%/3.6%/4.1% for weighted speedup over the baseline/ SDAR/STC techniques. Intuitively, with higher network traffic under memory-intensive workloads, the magnitude of improvement for HEPI is also slightly larger than for compute-intensive case II workloads with low network traffic. The improvements for case III workloads are however lower than that for case I workloads. This is because case I workloads are heterogeneous and thus there is more opportunity to identify packets from different types of applications, rank them and exploit stage I routers to benefit criticality of applications; which is not as effective with homogeneous case III workloads.

5.2.2 Platform scaling analysis

In this set of experiments, we were interested in observing the impact of scaling the platform complexity (mesh size, core count, memory controller count) on the performance of HEPI and other prioritization techniques. We explored performance for three different network sizes: i) 3×3 mesh with a concentration factor of 3 (27 cores) and 1 memory controller, ii) 4×4 mesh with concentration factor of 3 (48 cores) and 1 memory controller; and iii) 8×8 mesh with concentration factor of 1 (64 cores) with 2 memory controllers on diagonally opposite sides. Figure 4 shows the percentage improvement for system throughput, average memory latency, and weighted speedup over the baseline Local RR technique, for the three different platform complexities. The results are averaged over the three heterogeneous workload configurations in case I from Table 3 (results for case II and case III workloads are omitted for brevity, but showed similar trends). It can be observed that HEPI scales well with increasing core counts, network complexity, and memory controller complexity, showing notable improvements over other techniques. With higher network traffic and more opportunity to prioritize requests, the results are more pronounced. Hence, we get the best improvements for the 4×4 configuration with 48 cores as it has the most NoC traffic.

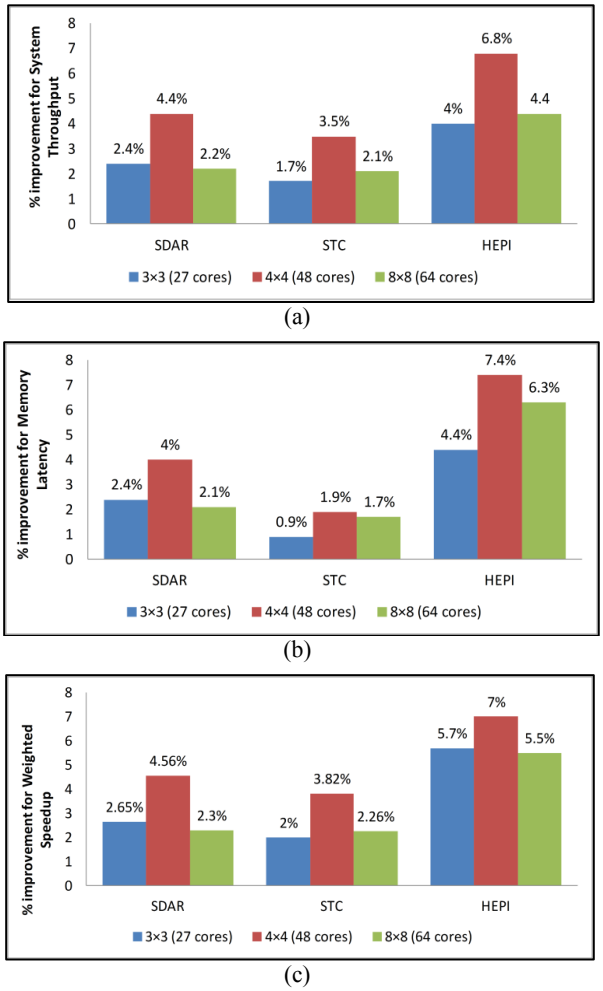


Figure 4: Percentage improvement over Local RR for different platform complexities: (a) system throughput (b) average memory latency (c) weighted speedup

5.2.3 Overhead analysis

This final set of results attempts to quantify and compare the overheads of implementing the HEPI prioritization technique with the implementation overhead for other prioritization techniques. Figure 5 shows the overall NoC fabric area for the

baseline Local RR, SDAR, STC and HEPI frameworks for the 32nm CMOS process technology node and the 36 core CMP configuration. Not surprisingly, the baseline technique having a NoC router with a simple round robin prioritization mechanism has the lowest area overhead. For the SDAR technique, the SDRAM-Aware NoC router has memory-specific information tables at each SDRAM-aware router and also requires buffers to store complete addresses of the previous arbitration winners. These additional components drive up its area overhead compared to the baseline case. STC routers require buffers to store L1 MPKI information for all the cores and central decision logic to calculate ranks for each application dynamically, hence this technique has a very high area overhead. In contrast, HEPI has a lower area overhead than for STC and SDAR as it performs ranking and application-centric prioritization locally with simple circuitry in stage I. HEPI stage II routers have low overhead as they require only a small RUB table shared by multiple routers and minimal circuitry to query/update the table.

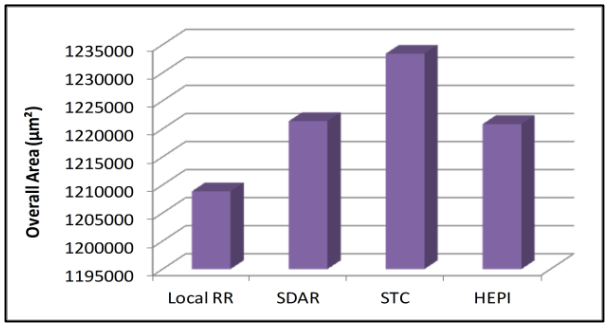


Figure 5: Overall area estimation for different techniques

6. Conclusion

In this paper, we propose a novel heterogeneous prioritization framework (HEPI) for CMPs in which two different packet prioritization approaches are proposed and applied to NoC routers, depending on whether the routers are nearer or farther away from the off-chip memory subsystem. A new ranking scheme for classifying an application's criticality was also proposed. Our experimental results show that the proposed HEPI framework outperforms fair prioritization techniques by up to 12.6% as well as other application-specific techniques from prior work by up to 6.9% for various multi-programmed workloads. HEPI also shows consistent improvements with increasing platform complexity. Given its competitive area overhead compared to other competing prioritization techniques, HEPI provides an attractive alternative for inclusion in emerging CMPs with multi-programmed workloads.

Acknowledgement

This research is sponsored in part by grants from NSF (CCF-1252500), SRC, and AFOSR (FA9550-13-1-0110).

References

[1] Y. Kim et al., "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior.", Proc. MICRO-43, 2010.

[2] Y. Jin et al., "Thread Criticality Support in On-Chip Networks", Proc. NoCArc 2010.

[3] S. Pasricha and N. Dutt, "On-Chip Communication Architectures", Morgan Kauffman, Apr 2008.

[4] A. Glew, "MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC", Proc. ASPLOS WACI Session, 1998.

[5] N. Dutt, "Memory-Aware NoC Exploration and Design", Proc. DATE 2008, pp. 1128-1129.

[6] S. Rixner et al., "Memory Access Scheduling," Proc. ISCA 2000.

[7] O. Mutlu, T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors", Proc. MICRO, 2007.

[8] Y. Kim et al., "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers", Proc. HPCA-16, 2010.

[9] R. Ausavarungnirun et. al., "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems", Proc. ISCA 2012.

[10] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems", Proc. ISCA-35, 2008.

[11] Z. Fang et al., "Core-Aware Memory Access Scheduling Schemes", Proc. IEEE International Parallel and Distributed Processing Symposium, (IPDPS) 2009, pp. 1-12.

[12] D. Kim et al., "A Network Congestion-Aware Memory Controller", Proc. NOCS 2010.

[13] W. Jang and D. Pan, "An SDRAM-Aware Router for Networks-on-Chip", Proc. DAC 2009, pp. 800-805.

[14] S. Phadke and S. Narayanasamy, "MLP-aware Heterogeneous Memory System", Proc. DATE 2011, pp. 1-6.

[15] W. Jang and D. Pan, "Application-Aware NoC Design for Efficient SDRAM Access", Proc. DAC 2010, pp. 453-456.

[16] R. Das et al., "Application-Aware Prioritization Mechanisms for On-Chip Networks", Proc. MICRO-42, 2009, pp. 280-291.

[17] R. Das et al., "Aérgia: Exploiting Packet Latency Slack in On-Chip Networks", Proc. ISCA 2010.

[18] A. Sharifi et al., "Addressing End-to-End Memory Access Latency in NoC-Based Multicores", Proc. MICRO 2012.

[19] A. K. Mishra, et al., "A Heterogeneous Multiple Network-On-Chip Design: An Application-Aware Approach", Proc. DAC 2013.

[20] D. Kroft, "Lock-up Free Instruction Fetch/pre-fetch Cache Organization", Proc. ISCA 1981, pp. 81-87.

[21] N. Jerger and L. Peh, "On-Chip Networks", Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2009.

[22] N. Binkert et al., "The Gem5 Simulator", ACM SIGARCH Computer Architecture News, vol. 39, Issue 2, May 2011.

[23] N. Agarwal et al., "GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator", ISPASS 2009.

[24] Micron Datasheet DDR2-667, http://download.micron.com/pdf/datasheets/modules/ddr2/.

[25] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads", IEEE Micro, May-June 2008, pp. 42-53.

[26] Micron Datasheet DDR3-1333, http:// download.micron.com/pdf/datasheets/dram/ddr3/.

[27] J. W. Lee, et al., "Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks", Proc. ISCA 2008.

[28] S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations", Proc. ISCAS 1995.

[29] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A Power-Area Simulator for Interconnection Networks", TVLSI 2012.

[30] S. Pasricha, and N. Dutt, "A Framework for Cosynthesis of Memory and Communication Architectures for MPSoC," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(3), pp. 408-420, Mar. 2007.