Addressing End-to-End Memory Access Latency in NoC-Based Multicores*

Akbar Sharifi, Emre Kultursay, Mahmut Kandemir and Chita R. Das
The Pennsylvania State University
University Park, PA, 16802, USA
{akbar, euki139, kandemir, das}@cse.psu.edu

Abstract

To achieve high performance in emerging multicores, it is crucial to reduce the number of memory accesses that suffer from very high latencies. However, this should be done with care as improving latency of an access can worsen the latency of another as a result of resource sharing. Therefore, the goal should be to balance latencies of memory accesses issued by an application in an execution phase, while ensuring a low average latency value. Targeting Network-on-Chip (NoC) based multicores, we propose two network prioritization schemes that can cooperatively improve performance by reducing end-to-end memory access latencies. Our first scheme prioritizes memory response messages such that, in a given period of time, messages of an application that experience higher latencies than the average message latency for that application are expedited and a more uniform memory latency pattern is achieved. Our second scheme prioritizes the request messages that are destined for idle memory banks over others, with the goal of improving bank utilization and preventing long queues from being built in front of the memory banks. These two network prioritization-based optimizations together lead to uniform memory access latencies with a low average value. Our experiments with a 4 × 8 mesh network-based multicore show that, when applied together, our schemes can achieve 15%, 10% and 13% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively.

1. Introduction

From an architectural side, in parallel to increasing core counts, network-on-chip (NoC) is becoming one of the critical components which determine the overall performance, energy consumption and reliability of emerging multicore systems [14, 31, 35]. While NoC helps to improve communication scalability of data, it also contributes to memory access latencies. Consider as an example a data read access in an NoC-based multicore. Such an accesses first checks L1 cache, and if it misses there, accesses L2 cache by traversing the NoC (assuming an S-NUCA-based [15] cache space management). If it misses in L2, it takes another trip over the NoC to reach the target memory controller determined by the address of the requested data. Depending on the current status of the queue in the target memory bank, it waits in the queue for some amount of time before reaching the DRAM. After finally reading the data from the DRAM, the same set of components are visited in the reverse order until a copy of the data is brought into the core. Clearly, each of the components in this round-trip access path (L1, L2, network, bank queue, and memory access) contributes to the end-to-end latency of this data access.

From an application side, interferences across simultaneously executing applications on various shared resources (e.g., NoC, memory controllers) can lead to high variances across the latencies of the off-chip accesses made by an application [8, 18, 27]. Specifically, while one request can be lucky to retrieve its data very quickly, another request of the same application can get delayed in the network and/or the memory controller queue, suffering a much larger round-trip latency. This variance can in turn lead to degradation in overall system performance as requests that experience much higher latencies than the average can block the progress of the application. To achieve high performance, it is crucial to reduce the number of memory accesses that observe very high latencies, but this should be done with care as improving the latency of an access can worsen the latency of another as a result of resource sharing. Therefore, from an overall system performance point of view, instead of each application aggressively trying to minimize the latencies of all its memory accesses, it is better to have each application balance the latencies of its memory accesses.

Motivated by these observations, we propose two network prioritization schemes that can cooperatively reduce end-to-end memory access latencies in NoC-based multicores. One of these schemes targets memory response messages (i.e., messages from the memory controller), whereas the other targets memory request messages (i.e., messages from the core side). Our first scheme prioritizes memory response messages such that, in a given period of time, messages of an application that experience higher latencies than the average message latency of that application are expedited. This helps us reduce the number of off-chip requests with high latencies and achieve a more uniform memory latency pattern. While one might want to apply a similar optimization for memory request messages as well, it is hard to identify whether a data access will experience high latency at the time of the request generation as it has not even entered the network yet. Instead, our second scheme approaches problem from another angle, where it expedites request messages to improve memory bank utilization. More specifically, observing that at any given time frame some banks are heavily loaded whereas others are idle, our second scheme prioritizes (at network routers) the request messages that are destined for idle banks over the others. This helps to improve bank utilization and prevent long queues in front of the memory banks. These two network prioritization-based optimizations, when applied together, result in uniform memory access latencies with a low average value.

We implemented our schemes in a simulation environment and tested their effectiveness using a diverse set of multiprogrammed workloads. Our experiments with a 4 × 8 mesh network-based multicore show that our first scheme is very effective in reducing the number of off-chip requests with very high latencies. As a result, we are able to achieve 11%, 6%, and 10% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively. When both our schemes are used together, these improvements jump to 15%, 10% and 13%, in the same order. Our experimental evaluation also reveals that the proposed schemes consistently generate good results under different

---

*This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687 and a grant from Microsoft Corporation.
values of the main simulation parameters. The rest of this paper is structured as follows. We provide an overview of our target multicore architecture and the main motivations behind our optimizations in Section 2. In Section 3, we give the details of our proposed schemes and discuss our network prioritization implementation. Section 4 presents our experimental framework and discusses our results. We contrast our approach with related work in Section 5, and conclude the paper in Section 6.

2. Background and Motivation

2.1. Target Multicore System

Figure 1 shows the high level view of an NoC-based multicore system that we target in this paper. As can be observed, this multicore system has a number of nodes arranged in a 2D grid (an example present day product using a similar organization is TILEPro36 by Tilera [1]). In this architecture, each node contains a processing element and a hierarchy of caches; only two levels of caches are shown in Figure 1. The shared last level cache (L2) has a banked organization where the L2 space in each tile represents a separate bank. The mapping of data to L2 cache banks is very important as far as performance is concerned, and in one mapping scheme (SNUCA [15]) which is also used in this paper, each cache block-sized unit of memory is statically mapped to one of the cache banks based on its address [16]. This address-based mapping results in an interleaving of cache blocks across cache banks. The main memory is connected to this multicore via memory controllers (MCs) on the corners (or sides), each controlling a single memory channel with possibly many memory modules (DIMMs) connected. Each channel consists of one or more ranks, and each rank is a group of memory banks. Different memory banks can be accessed at the same time, but they share common address and data buses. More details on the internal structure of the main memory can be found in [3, 9, 29]. Note that the target memory bank for each memory request is fixed and depends on the OS address mapping. If more than one memory request are queued in a memory controller trying to access the same bank, the memory scheduler decides which one should be served first [33, 34].

2.2. Memory Accesses in an NoC-based Multicore

Once a core issues a memory request, first the local L1 is checked for the presence of the requested data and if this fails, the request is forwarded to the corresponding L2 bank via the on-chip network. Finally, if the data is not found in the L2 bank, an off-chip memory access is made, again using the on-chip network. Figure 2 depicts this flow in our target architecture. One can make several critical observations from this figure. First, the L2 access latency is a function of the distance between the requesting core (L1) and the target L2. It is to be noted that, longer this distance gets, higher the chances for network contention. Second, the total latency to serve an L2 miss (i.e., the overall memory latency) includes memory controller queuing and memory service latencies (i.e., the memory access latency) as well as the latency of four trips on the network: L1 to L2, L2 to memory controller (MC), MC to L2, and L2 to L1. As a result, network latency can play a significant role in overall memory access latency (in fact, our experiments show that, even in a 4 × 8 multicore, cumulative network latency can be comparable to memory access latency). Third, the time spent in an MC depends on other requests that target the same MC as well as the (memory request) scheduling policy adopted by the architecture. Fourth, at any given time, there will be many memory requests traveling on the network. These requests contend for the shared network resources, resulting in additional delays. For example, a message from a core destined to an L2 may contend for the same link/Router with another message going from an L2 to an MC. Lastly, the time it takes to service a memory request (after it gets its turn from the memory scheduler) depends on whether it results in a row buffer hit or not.

2.3. Out-of-Order Execution

In our target system, each processing element (core) is a high performance out-of-order processor. It can issue multiple instructions by looking at its instruction window. As a result, if these instructions need to access the main memory, the memory requests can be issued at the same time with the hope of improving the overall performance by not stalling the processor for a single memory request. This is referred to as Memory Level Parallelism (MLP) [10, 11]. Note that although memory requests may return out-of-order, the instructions are committed in-order and therefore, a higher memory access latency for an instruction may affect the performance of an application significantly [10]. As an example, suppose that an application issues four load instructions (load-A, load-B, load-C and load-D, as shown in Figure 3) that need to access memory. As discussed, these instructions may be ready to commit at different times due to the network and memory latencies and also whether they are hits or misses...
in the on-chip caches. Note that in the instruction window shown in Figure 3, the load instruction (e.g., load-A) that needs to wait for a longer time to access the data (as compared to the other instructions) becomes a bottleneck for the application. The chances that a memory request will be a bottleneck depend on its delay and also other outstanding memory requests from the same application. For example, if an application is not memory intensive (an application with a low MPKI [misses per Kilo instructions]), the likelihood that one of its off-chip memory requests would be a bottleneck is high. This is why previously-proposed memory scheduling schemes [17, 18] and network prioritization [7], higher priorities are given to the requests coming from this type of applications. However, these prior works do not consider the network and memory access latencies together. For instance, in the application-aware network prioritization scheme proposed in [7], the time it takes to access the memory is assumed to be constant, whereas in reality memory requests (when they reach the memory controller in an NoC based multicore) may face different latencies due to queuing.

2.4. Motivations

Our proposed approach consists of two network prioritization based schemes (Scheme-1 and Scheme-2). These schemes are proposed based on two main motivations presented in this section. Specifically, Motivation-1 and Motivation-2 below motivate for Scheme-1 and Scheme-2, respectively.

2.4.1. Motivation 1: Some Memory Accesses Experience Much Higher Delays than Others As illustrated in Figure 2, a round-trip memory access has five paths/stages. The total delay of each access is equal to the sum of the delays on these paths. To have a better understanding of how much time each memory request spends on different stages/parts on average, we simulated a 4 × 8 NoC-based multicore with 4 memory controllers attached to the corners, and each core executing an application from the SPEC2006 benchmark suite [12] (see workload-2 in Table 2). More details on the experimental configuration and workloads are given in Section 4.1. Figure 4 plots the average round-trip delays of the off-chip memory accesses issued by one of the cores (executing milc in workload-2).

Figure 4: The average delays of the off-chip memory accesses issued by one of the cores (executing application milc in workload-2).

Figure 5 plots the distribution of the memory accesses across different delay regions. The values on the x-axis correspond to different latencies observed during execution, and the y-axis shows the percentage of the total memory requests for different latencies. The area under the curve between x = d1 and x = d2 gives the fraction of the memory accesses that have a total latency between d1 and d2. As can be observed from this plot, the latencies of most of the accesses are around the average, but there are few accesses with very large delays (600 cycles or more). Unfortunately, these high latency accesses can be very problematic and degrade overall application performance significantly.

Our first scheme identifies these (high latency) accesses (after the DRAM access is completed) and attempts to reduce their latencies with the goal of improving the overall system performance. Specifically, our proposed scheme tries to expedite these slow messages on their return path by giving them a higher priority in the on-chip routers. In other words, we try to make the latencies of the different memory accesses issued by the same application as uniform as possible.
In this example, the memory utilization could be improved if R-2 bank-1 is a common bottleneck for all co-running applications, in a bank queue suffers long queuing delay, it is most likely that other late requests in the same bank. In other words, if one request at the bank queues. However, prioritizing these requests would harm other late requests in the same bank. In other words, if one request in a bank queue suffers long queuing delay, it is most likely that this bank is a common bottleneck for all co-running applications, and consequently prioritizing the requests from one application over others would not help in a global sense.

Motivated by these observations, Scheme-1 attempts to reduce the delays coming from the third factor, since right after the memory controller the delays accumulated so far can give a good estimate of whether a memory access is going to be “late” or not. In this scheme, each memory message (packet) has a field that maintains the age (“so-far delay”) of the corresponding access (based on the delay at each router/ stage, this field is updated over the round-trip of the memory access). Note that the variations in network latencies (the first factor) can come from two contributions: (1) the distances in the network from L1 to L2 and from L2 to MC, and (2) the network traffic. Both these sources are captured by our so-far delay field, and consequently handled by our scheme.

To illustrate how Scheme-1 works, let us explain it through a simple example. In Figure 8, suppose that MC1 has received a memory request from core-1, and R0 is the corresponding response message

![Graph of different banks idleness](image)

**Figure 6:** The idleness of different banks of a memory controller.

**Figure 7:** A snapshot that shows the states of the three banks controlled by one of the memory controllers.

### 3. Our Approach

In this paper, we propose two network prioritization schemes that consider the states of the memory bank queues and the overall memory access delays in NoC-based multicores. In this section, we first explain how our schemes work and then discuss the implementation details of the specific network prioritization method we employ. In our proposed approach, Scheme-1 focuses on the response messages coming back from the memory and Scheme-2 considers request messages destined for the off-chip memory. We assume a one-to-one mapping between applications and cores.

#### 3.1. Scheme-1: Reducing Variations across Memory Accesses

This scheme is based on the motivation discussed in Section 2.4.1. Recall from Figure 5 that some memory accesses issued by an application can take much longer times to complete compared to other accesses from the same application (we call them late accesses). As an example, in Figure 5, about 10% of the accesses have delays larger than 600 cycles (while the average latency is about 350 cycles). As mentioned earlier, these late accesses can severely degrade the performance of applications. Suppose for example that an application executes \( a = b + c \). In this instruction, if one of the operands becomes ready only after a long time, this will delay the completion of this instruction as well as all instructions that depend on it. The main goal behind our Scheme-1 is to mitigate the impact of the late accesses by helping them reach their destinations faster.

As shown in Figure 4, three different factors can cause memory accesses to be late in the system: (1) the network latency from the source cache bank to the corresponding memory controller, (2) the memory access latency (which includes both queuing delay and the time it takes to access the DRAM), and (3) the network latency from the memory controller to the source cache bank. Therefore, in order to reduce the delay experienced by the late accesses, one can potentially target these three delays. The first and third delays can be reduced by giving higher priorities to the request and the response messages in the network. However, for the first part (network latency after an L1 miss, from the L1 bank to the memory controller), it is not known that a memory request is going to be a late memory access, since the total delay depends mainly on the memory response time (factor 2) and the network delay of the response messages (factor 3), as illustrated in Figure 4. To reduce the response time of the memory (factor 2), late memory requests could be scheduled faster at the bank queues. However, prioritizing these requests would harm other late requests in the same bank. In other words, if one request in a bank queue suffers long queuing delay, it is most likely that this bank is a common bottleneck for all co-running applications, and consequently prioritizing the requests from one application over others would not help in a global sense.

To explain how this observation can be exploited, consider Figure 7 which shows the states of three banks controlled by one of the memory controllers at time \( t_1 \). As can be seen, Banks 0 and 1 have requests to be serviced, whereas Bank-2 is idle. Assume further that there are three memory requests (from L2 misses) in the on-chip network destined for these banks as shown in Figure 7 (R-0, R-1 and R-2 which are destined for Bank 0, 1 and 2, respectively). In this example, the memory utilization could be improved if R-2 could reach its memory controller faster so it can be serviced by Bank-2 immediately. Our proposed scheme exploits this load variation across different banks by using the local history at each node to accelerate the memory requests on the network destined for the idle banks.
for that request after it is serviced by MC1. Once R0 is ready, the age field (the so-far delay) in R0 gets updated based on the delay that has been imposed by the memory. After that, the updated so-far delay is compared with a threshold value Th. If it is larger than Th, this memory access is considered to be late and, on the return path, R0 will have a higher priority to reach its destinations (the paths from MC1 to L2 and L2 to L1 illustrated in Figure 8). Note that in the memory controllers, each core/application is associated with a threshold value. These threshold values are determined and sent to the MCs by the cores periodically (every 1ms) over the execution and they are also prioritized over other requests (each MC has storage to keep the threshold values). Since this happens every 1ms, the overhead on the other messages is small. In our default configuration, each core sets its threshold value to $1.2 \times \text{Delay}_{\text{avg}}$, where $\text{Delay}_{\text{avg}}$ is the average delay of the off-chip memory accesses that belong to that application/core. Once a response message for an off-chip memory access comes back, the round-trip latency of that off-chip memory access is read from the message (the age field) and $\text{Delay}_{\text{avg}}$ is updated accordingly. Consequently, during the course of execution, the threshold value used for each application changes dynamically.

Note also that we compute the priority of a response message when the message is about to be injected into the network by the memory controller. However, at that time instant, the memory controller only knows the so-far delay of the request, and therefore, any delay threshold that will be used in calculating the priority must be defined based on the average delay up to and including the memory access latency $\text{Delay}_{\text{so-far-avg}}$. For this reason, we set our threshold to $1.2 \times \text{Delay}_{\text{avg}}$ (which is almost equivalent to $1.7 \times \text{Delay}_{\text{so-far-avg}}$; both averages are marked in Figure 9 using vertical solid lines). Figure 9 plots two delay distributions of the memory accesses (issued by the core executing application milc from workload-2). Specifically, the dashed curve shows the distribution of the round-trip delays of the messages, while the solid curve plots the distribution of the so-far delays at the point right after the memory controller (i.e., when the data is read from the memory, and is about to be injected into NoC).

**Implementation Details.** As discussed earlier, Scheme-1 needs each memory access message to have a field (called “age” field) capturing the so-far delay of the message in the NoC. We assume that this field is 12-bit long and its value is in cycles. We believe a 12-bit field is sufficient in our work, since it is very rare that the round-trip of an off-chip message takes more than $4096 - 1$ cycles in the system. We assume that the header flit has room for the so-far delays (12-bit) in a 128-bit header. At each router/memory-controller, once the message is ready to be sent out, the age-field value is updated as follows:

$$
\text{age} = \text{age} + \frac{(\text{local}_{\text{time}} - \text{message}_{\text{entry}}_{\text{time}}) \times \text{FREQ}_{\text{MULT}}}{\text{local}_{\text{frequency}}},
$$

where $\text{local}_{\text{time}}$ is the local router/MC clock cycles, $\text{message}_{\text{entry}}_{\text{time}}$ is the time when the message enters the routers/MCs, $\text{FREQ}_{\text{MULT}}$ is a constant value (used to work in integer domain, instead of fractions of cycles) and $\text{local}_{\text{frequency}}$ is the local operating frequency. As one can observe, in each hop, first, the local delay of each incoming message is computed ($\text{local}_{\text{time}} - \text{message}_{\text{entry}}_{\text{time}}$) and then this value is added to the age-field of the message. Note that dividing the local delay by the $\text{local}_{\text{frequency}}$ allows our scheme to work when routers/MCs operate at different frequencies. Note also that, the entry time of each message is not transferred over the network, and each router keeps track of its own local delay and computes the so-far delay locally. Instead of updating the age field at each hop, one may choose to employ a global clock to compute the so-far-delays. In this method, each message is labeled with the starting time and, at the destination, the starting time is subtracted from the current time to obtain the so-far-delay. However, in this mechanism, all the nodes need to have clocks and these clocks should be synchronized.

**3.2. Scheme-2: Balancing Memory Bank Loads**

Based on Motivation-2 discussed in Section 2.4.2, the main goal behind this scheme is to increase the utilization of the memory banks in the system. To do this, the requests destined for the idle banks are given higher priorities in the network to reach their target memory controllers faster. However, the number of requests queued in each memory bank varies over the execution, and more importantly, a core in our 2D mesh does not know about the states of the memory banks. In other words, no global information is available to the
nodes to help them decide whether different memory requests they issue are destined for idle banks or not. To address this problem, in our proposed scheme, each node exploits “local information” to estimate the pressure imposed by the requests on the memory banks. Specifically, each router keeps and updates a table (called “Bank History Table”) that records the number of off-chip memory requests sent from this router to each bank in the last $T$ cycles. An off-chip request is given a higher priority in the on-chip network if no message has been sent to the same bank according to the value in the bank history table. In Section 4, we present results that show how bank idleness is reduced when Scheme-2 is employed.

Note that due to Scheme-2 and the variation in the network latencies, different requests that are destined for the same memory bank may not reach the bank queue in order (based on their ages). This may cause more delays for the late accesses. Message ordering can be handled by the memory scheduler, since the scheduler knows the timing information for all arriving messages.

### 3.3. Network Prioritization Implementation

Further, that would be very costly to use stronger routers and more network resources. Our schemes can be used along with other architectural solutions to provide additional benefits. In this section, we give the implementation details of our prioritization method. In our NoC-based multicore, we assume a typical architecture with 5-stage pipeline for the on-chip routers [21]. We further assume flit-buffering and VC (virtual channel) credit-based flow control at every router [4]. In this architecture, each network message is split into several flits with fixed-lengths and the flow of the flits follows the wormhole switching protocol [6]. For further details, we refer the reader to [21].

Once a header flit enters a router, in the first stage of the pipeline, which is called buffer write (BW), the flit is written into the allocated input buffer. In the next stage (routing computation (RC)), the routing algorithm is invoked and the right output port is determined for the packet based on the position of the current router and the destination address extracted from the header flit. The next stage, called virtual channel allocation (VA), is where a free output virtual channel is reserved for the packet. After this stage, the flit needs to get the permission to use the crossbar switch (switch allocation (SA)). In the last stage, the flit traverses the switch (ST) to be sent over the physical link (LT). Note that only the header flit passes through these stages, and the body and tail flits skip the RC and VA stages as they simply follow the header flit.

In our approach, the messages to be expedited have higher priorities in VC and SW arbitrations (for virtual channel and switch allocations). The VC arbitration is done when an output VC is preferred by more than one input VC and, one of them should be granted. The SA arbitration on the other hand has two phases. In the first phase, only one of the ready VCs is selected per input port and, in the second phase, one VC is selected for each output port. The reason for these two arbitrations is that, at each cycle, only one flit can be sent from an input and each output is able to receive a single flit.

Although prioritization in the arbitration stages expedites the messages and helps them reach their destinations faster, there is a limit in this message acceleration, since it takes at least 5 cycles for the data flits to pass through a router in the network. To tackle this limitation, we also employ a mechanism called “pipeline bypassing” [21], which gives the late messages the opportunity to go through fewer number of pipeline stages in the routers. In this mechanism, once the header flit of a message (with high priority) enters a router, in the same cycle, after it is written to the input buffer, one free output VC is allocated to it (VC allocation) and the switch is reserved for this flit (SW allocation) (in other words, the BW, RC, VA and SA stages are combined and performed in the first stage which is called the “setup stage”). Figure 10 illustrates the pipeline stages for the baseline router as well as the one with pipeline bypassing). If in this cycle, there is a conflict in selecting the free output VC and the switch allocation between this flit and a flit (with normal priority) in another VC, the flit with the higher priority is prioritized over the others. Body flits also use this “bypassing” only when the input buffer is empty once they enter the router (Note that the input buffers are always idle when the header flits enter).

To avoid starvation in the system, we also use “age fields” in the messages (which are also used in Scheme-1). In our prioritization scheme, flit A is prioritized over flit B in the cases discussed above, if (1) flit A has a high priority but flit B has the normal one, and (2) the age of message B is not more than $T$ cycles greater than that of message A (flits A and B belong to messages A and B). Note that the routers also consider the local delays in addition to the age fields in the messages. “Batching” [8] is another method that can be used to avoid starvation. In this method, time is divided into intervals of $T$ cycles and there is a field in each message that indicates the batching interval in which the message was injected to the network. The packets are prioritized if they belong to the same batch. However, this method requires a synchronization among the cores since they need to access a common global time.

### 4. Experimental Evaluation

In this section, we first explain our simulation framework and the different workloads that we used in our evaluations and then, present and discuss our experimental results.

#### 4.1. Setup, Metrics, and Workloads

**Setup:** We use GEMS [25] as our simulation framework to evaluate our proposed schemes. This framework allows us to simulate an NoC-based multicore system. GEMS consists of two main components called Opal and Ruby, and employs Simics [24] as the base simulator. Opal implements an accurate model for out-of-order cores, and Ruby implements the network modules (the routers and the links) and the memory controllers. The instructions are executed by Opal and, if they require a memory access, a request message is injected to Ruby. Opal receives the response message once the requested data comes back from an L2 bank or one of the memory controllers (simulated by Ruby). Table 1 gives our baseline configuration. As shown in this table, our baseline system has 32 cores connected by a $4 \times 8$ mesh-based NoC, and also 4 memory controllers are connected to the four corners of the network. In our evaluation, we employ cache line interleaving where the consecutive lines of an OS page are mapped to different memory controllers. Note that this strategy helps to avoid creating hot spots in the memory controllers. In the results presented below, the overheads incurred by our proposed schemes are included.

<table>
<thead>
<tr>
<th>BW</th>
<th>RC</th>
<th>VA</th>
<th>SA</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipeline Bypassing</td>
<td>setup</td>
<td>ST</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation Metric: We use normalized weighted speedup metric to quantify the benefits of our proposed scheme. Weighted speedup is defined as: \( WS = \sum_{i} \frac{IPC_{i}(\text{shared})}{IPC_{i}(\text{alone})} \), where \( IPC_{i}(\text{shared}) \) is the IPC of application \( i \) when it is executed with the other applications in the workload and \( IPC_{i}(\text{alone}) \) is the IPC of the same application when it is executed alone without having any contention with the other applications. In this section, we present the weighted speedup values that are normalized to that achieved by the default case where no prioritization is employed.

Workloads: We formed our workloads using the applications from the SPEC2006 benchmark suite [12]. Table 2 gives the 18 workloads that we used in our experiments on our 32-core system (the numbers in the parenthesis represent the number of copies for each application in the workload). The workloads listed in Table 2 cover all applications and are categorized into three different groups based on the memory intensity of the applications: (1) Workloads 1 through 6 (mixed workloads): in these workloads, half of the applications are memory intensive applications (with high MPKI) and the remaining ones are memory non-intensive. (2) Workloads 7 through 12: all applications in this category are memory intensive. (3) Workloads 13 through 18: none of the applications in this group is memory intensive. In other words, to form a workload, we first categorize the applications into three groups based on the memory intensity (since our scheme is proposed for memory accesses) and then, for each workload, we randomly pick 32 applications from the specific category (note that this strategy may choose some applications multiple times, but all the applications in a workload belong to the same memory-intensity region). MPKI values representing the memory intensities of the applications in the SPEC2006 suite can be found in [17].

4.2. Results

We now present the experimental results collected using various configurations. In executing a workload, the simulation was fast-forwarded for 1 billion cycles, and then we collected our statistics in the next 100-million cycle run. As stated earlier, we employed a one-to-one mapping between applications and cores, that is, each core executes one application and the application-to-core mapping does not change during the execution.

Results for a 32-core system with the baseline configuration: As mentioned earlier, in this configuration, 32 cores are connected by a 4 × 8 2D mesh-based NoC and 4 memory controllers are placed in four corners of the mesh (other parameters are as given in Table 1).

Figure 11 plots the normalized weighted speedup values achieved for our workloads, which are categorized into three groups as shown in Table 2. As discussed in Section 3, our approach to reducing end-to-end latency employs two complementary schemes. In Scheme-1, if the so-far delays of the response messages are larger than a threshold value (the default value of this threshold is 1.2 × Delay\text{avg}) after the memory controller stage, they are given a higher priority in the network on the return paths to reach their destinations faster, whereas Scheme-2 accelerates the request messages that are destined for the idle memory banks. As stated earlier, in this scheme, the decision of whether a request message is destined for an idle bank or not is made based on the local information. When an L2 miss occurs, if over the last T cycles, the number of requests sent to the same bank is less than a threshold (th), the request is given high priority (the default values for T and th are 200 cycles and 1, respectively).

In Figure 11, two bars are presented for each workload. The first bar shows the performance improvement achieved when Scheme-1 is employed in the system and the second one is for the case where Scheme-1 and 2 are employed together. Note that the weighted speedup values presented in Figure 11 are normalized to the base case in which the applications are running together but no prioritization scheme is employed.

As can be observed from Figure 11, our approach (Scheme-1 + Scheme-2) improves performance by up to 13%, 15% and 10% for the mixed, memory intensive and memory non-intensive workloads, respectively. Further, in general, higher performance improvements are achieved when the applications are more memory intensive. This is because when the co-running applications are more memory intensive, the traffic on the network is higher and, as a result, the impact of our network prioritization based approach on accelerating the late messages becomes more pronounced. However, our proposed scheme does not improve the performance for all the workloads tested. For instance, the speedup values achieved by Scheme-1 for workloads 2 and 9 are slightly less than 1. The reason for this behavior is that giving higher priorities to some of the messages in the network hurts the other messages and, this can offset the benefits of our scheme in some cases.

The impact of employing Scheme-1 and Scheme-2 can be observed in the distribution of the end-to-end latencies and the average idleness of the memory banks, respectively. Figure 12a plots 8 “cumulative distribution functions” (CDFs) of the off-chip memory accesses generated by the first 8 applications in workload-1. In this figure, the values on the x-axis are the total latencies (in cycles) and the y-axis shows the fraction of the total number of off-chip memory accesses. Each curve corresponds to one of the applications and the \( F(x) \) value gives the fraction of the total accesses with delays less than x. Figure 12b plots the new CDFs of the off-chip accesses for the same 8 applications when Scheme-1 is employed. It can be observed from Figure 12a that 90% of the messages have an average total latency about 700 cycles (shown with dashed lines), whereas Scheme-1 reduces it to about 600 cycles (see Figure 12b).

Figure 12c plots the “probability density function” (PDF) of the memory accesses issued by one of the applications (lbm) in workload-1 before and after Scheme-1 is employed (the area under the curve shows the fraction of total number of accesses). Employing Scheme-1 reduces the number of messages with large delays and move them from Region-1 to Region-2. This results in about 8%
performance improvement for this application. As one can observe from this figure, not all the messages are transferred to Region-2 and there are still messages in Region-1. This is because Scheme-1 reduces a portion of the total memory access latency and, for instance, if a message spends a long time in the memory queue (e.g., 700 cycles), Scheme-1 will not be able to move this message to Region-2.

To illustrate the impact of Scheme-2, we plot in Figure 13 the idleness values for different banks of a memory controller when Scheme-2 is employed and when no scheme is used (workload-1). As can be seen, Scheme-2 reduces the idleness of our banks (resulting in an overall system performance improvement of more than 5%). Figure 14 plots dynamic reduction in bank idleness over the course of execution for one of our workloads (workload-1).

### Results for a 16-core system:

We also ran our experiments in a smaller system with 16 cores connected by a 4 × 4 mesh-based NoC. In this configuration, two memory controllers with the same parameters as in Table 1 are attached to two opposite corners of the mesh.

Figure 15 plots the speedup values achieved by our proposed scheme when our workloads are executed on this smaller system. We picked the first half of the applications in each workload shown in Table 2 for this experiment (for the mixed workload, the first half of the memory intensive and memory non-intensive applications are selected). Averaged over all the workloads, the speedups achieved in these experiments are about 8%, 10% and 5% for the mixed, memory intensive and memory non-intensive workloads, respectively.

A comparison between Figures 11 and 15 reveals that the speedup values for the 32-core system are generally higher than those for the 16-core system. This is because as the network size increases, the network delay contributes more to the round-trip latencies of the off-chip accesses and, as a result, the impact of the network prioritization is amplified.

**Impact of the threshold value in Scheme-1:** As discussed earlier, to determine whether a response message is late or not, after a request is serviced by the corresponding memory controller, the so-far delay of the request is computed. The so-far delay is compared against a threshold and if it is larger than this threshold value, the corresponding memory access is considered to be “late”. Recall also that the default value for this threshold is $1.2 \times Avg_{delay}$. Note that $Avg_{delay}$ is the average round-trip latency of the off-chip requests that belong to the same application (issued by the same core) and is computed dynamically by the source core. To study the impact of the threshold values on the achieved speedups, we performed experiments for two other threshold values: $1.1 \times Avg_{delay}$ and $1.4 \times Avg_{delay}$. Figure 16a plots the speedup values achieved for the three cases when workloads 1 through 6 are used. As can be observed, when the threshold is set to a larger value, the speedup values reduce since fewer messages are considered to be
Plots the speedup values for $T$ is less than a threshold ($t_{th}$) values for $T$ over the last $T$ cycles, the number of requests sent to the same bank increases, prioritizing too many messages in the network hurts other messages, leading eventually to network congestion and performance degradation.

**Impact of the $T$ value (history length):** In Scheme-2, the decision of whether a request message is destined for an idle bank or not is made based on the local information. When an L2 miss occurs, if over the last $T$ cycles, the number of requests sent to the same bank is less than a threshold ($t_{th}$), the request is given high priority (the default values for $T$ and $t_{th}$ are 200 cycles and 1, respectively). Figure 16b plots the speedup values for $T = 100, 200,$ and 400. As can be observed, for $T = 400$ the speedup values are not as high as the case where $T=200$ since the number of late requests decreases. However, reducing the $T$ length ($T = 100$) does not necessarily increase the speedups either (see workload-2 and workload-4) due to the imprecision in finding the idle banks and the slow down imposed on the other messages in the network.

**Impact of the number of memory controllers:** Finally, Figure 16c plots the performance improvements for our mixed workloads when there are two and four memory controllers in the system. One can observe that, the performance improvement is slightly increased when there are fewer number of memory controllers in the system. The reason for this can be explained as follows. When there are fewer number of memory controllers in the system, due to the pressure increase on the bank queues, there will be more critical and late accesses in the system that can be enhanced by Scheme-1, and this results in higher performance. Note also that, although the benefits form Scheme-2 may reduce with fewer number of memory controllers (since there will be less idle banks), due to the improvement achieved by Scheme-1, the overall improvement (Scheme-1 + Scheme-2) is slightly better (see Figure 16c). However, the performance improvement is reduced for w-2 and w-3 due to the lower improvements achieved by Scheme-2.

**Sensitivity to the structure of the routers:** Although many schemes have been proposed by prior research to reduce message latency and contention in the on-chip networks, they do not completely eliminate the network contention and we are far from achieving the ideal performance. Therefore, our proposed schemes can be employed with the presence of the other proposed strategies to speed up the critical off-chip messages by prioritizing them in the network. One of the techniques to reduce the network latency is to employ routers with the fewer number of pipeline stages. As given in Table 1, in our default configuration, the routers are implemented as five-stage pipelines. However, the number of stages can be reduced to two (as discussed in Section 3.3, but here, all the flits have the opportunity to traverse each router in two cycles if there is no contention). Figure 17 compares the performance improvement achieved by our proposed schemes (for w-1 to w-6) when the routers have two and five pipeline stages. As can be observed from this plot, the performance is still improved up to 10% for the 2-stage pipeline. However, the improvement is 25-40% lower compared to the 5-stage pipeline case. This is because the percentage of the network latency that

**Figure 12:** (a) The CDFs of the off-chip accesses for the first 8 applications in workload-1. The values on the x-axis are the total (end-to-end) memory access latencies (in cycles), and the y-axis shows the fraction of the total number of off-chip memory accesses. (b) The CDFs of the off-chip accesses for the first 8 applications in workload-1, when Scheme-1 is employed. (c) The delay distributions of the memory accesses for one of the applications (lbm) in workload-1 before and after Scheme-1 is employed. This distribution change results in 8% performance improvement.

**Figure 13:** Idleness values of different banks of a memory controller when Scheme-2 is employed and when no scheme is used. As can be observed, Scheme-2 reduces idleness in most of the banks.

**Figure 14:** Average idleness of the memory banks (workload-1) over the execution.
can be reduced (for the selected messages) by the network prioritization decreases for the 2-stage pipeline case (note that in this case no router bypassing is employed and the critical messages are prioritized in the message arbitration done in the routers). However, other techniques such as express channels proposed in [21] can be employed to increase the latency savings.

5. Related Work
Main memory accesses have been identified as a critical bottleneck in both multiprogrammed and multithreaded workloads [8, 18, 27]. Prior studies [2, 5, 22] have shown that packet latency variance can be reduced by using age-based prioritization at the network routers. These schemes added a new field to each packet to track its current network latency and assigned a higher priority in the router to packets that suffered higher network latencies during switch allocation. Lee et al. [23] proposed the use of a different metric with age-based prioritization, where the hop count of a packet is used to assign its network priority. However, livelocks can occur when such a static metric is used for prioritization, and hence, this approach required the implementation of complex probabilistic priority arbiters to prevent livelocks. In contrast to these age-based schemes, in our work, we compare the latency of a request coming from a core only against the average latency of the requests from the same core. Therefore, priorities for each core are calculated independently, which enables per-core optimization of the memory access latency variance.

Two recent approaches to network prioritization-based latency optimization are proposed by Das et al. [7, 8]. In [7], packets are prioritized based on the types of the co-running applications and in [8], the slack of a network packet is defined as the number of cycles the packet can be delayed without affecting the execution time of the application. Although these works also employ network prioritization, they do not have a detailed memory model (they assume fixed delay). In contrast, we have a detailed memory model (with queuing and row-buffer modeling) and use accurate so-far delay information dynamically updated during execution. Further, in [7, 8], latency/slack calculation is done at the core side based on indirect parameters such as hit/miss status of the packet and number of hops. However, it becomes extremely difficult to accurately predict latency/slack at the cores when multiple requests from different cores compete for the same MCs. Lastly, these prior works do not consider memory bank idleness. Consequently, our schemes are different from these prior works.

As mentioned earlier, in our work we use an S-NUCA [15] based cache space management. In [13], a novel scheme called Reactive NUCA is proposed for data placement in the distributed caches. In the proposed method, data is placed in the local L2 slices for the private data patterns seen in multi-programmed workloads. This scheme reduces the network latency of the memory accesses by elimi-
We proposed two network prioritization schemes that reduce the end-

Another important component of off-chip memory access latency variance is the variations in memory queuing latency. In [29, 30], the authors targeted at reducing the variance of memory access latencies at the memory controller. These approaches coordinate multiple on-chip memory controllers to equalize the performance penalty suffered by each application due to interferences coming from other applications. The effectiveness of these schemes on a large multicore with high network latencies is unclear. On a large network, communicating information across memory controllers itself can easily have a very high latency, thus increasing the reaction time to dynamic changes in execution behavior.

Clearly, directly improving off-chip memory access latencies can also improve overall system performance. Memory queuing latencies can be improved by finding a better scheduling of memory requests in the memory bank queues [17, 18], designing lower latency routers [19, 26, 32], or employing better flow-control [20, 21]. We would like to note that the network prioritization schemes we propose in this work can be used with any memory scheduling algorithm, router micro architecture, or flow control scheme. For instance, while our Scheme-1 is orthogonal to the memory scheduling scheme employed (as it optimizes network latency component of the memory), the increase in the average number of entries in memory bank queues due to our Scheme-2 can enable these complex memory schedulers to find even better schedules as they can see a larger window of memory requests. We postpone the investigation of such interactions to a future study.

6. Conclusion

We proposed two network prioritization schemes that reduce the end-to-end memory access latency in multicores. Our first scheme addresses the latency variance in memory accesses that belong to the same application, and expedites memory response messages that experience high latencies by prioritizing them. This reduces the number of off-chip requests with high latencies and achieves a more uniform memory latency pattern. Our second scheme improves memory performance by prioritizing memory request messages that are destined for idle banks over other requests. This optimization increases bank level parallelism and improves memory utilization. Through an extensive evaluation of our schemes on a 4 × 8 mesh-based multicore, we show that our first scheme is very effective in reducing the number of off-chip requests that experience very high latencies and achieves 11%, 6%, and 10% performance improvement on memory intensive, memory non-intensive, and mixed multiprogrammed workloads, respectively. When both our schemes used together, these improvements jump to 15%, 10% and 13%, in the same order.

References