OSCAR: Orchestrating STT-RAM Cache Traffic for Heterogeneous CPU-GPU Architectures

Jia Zhan* Onur Kayıran† Gabriel H. Loh† Chita R. Das‡ Yuan Xie*
*University of California, Santa Barbara †Advanced Micro Devices, Inc. ‡The Pennsylvania State University

***Abstract*—As we integrate data-parallel GPUs with general-purpose CPUs on a single chip, the enormous cache traffic generated by GPUs will not only exhaust the limited cache capacity, but also severely interfere with CPU requests. Such heterogeneous multicores pose significant challenges to the design of shared last-level cache (LLC). This problem can be mitigated by replacing SRAM LLC with emerging non-volatile memories like Spin-Transfer Torque RAM (STT-RAM), which provides larger cache capacity and near-zero leakage power. However, without careful design, the slow write operations of STT-RAM may offset the capacity benefit, and the system may still suffer from contention in the shared LLC and on-chip interconnects.**

While there are cache optimization techniques to alleviate such problems, we reveal that the true potential of STT-RAM LLC may still be limited because now that the cache hit rate has been improved by the increased capacity, the on-chip network can become a performance bottleneck. CPU and GPU packets contend with each other for the shared network bandwidth. Moreover, the mixed-criticality read/write packets to STT-RAM add another layer of complexity to the network resource allocation. Therefore, being aware of the disparate latency tolerance of CPU/GPU applications and the asymmetric read/write latency of STT-RAM, we propose OSCAR to Orchestrate STT-RAM Caches traffic for heterogeneous ARchitectures. Specifically, an integration of *asynchronous batch scheduling* and *priority based allocation* for on-chip interconnect is proposed to maximize the potential of STT-RAM based LLC. Simulation results on a 28-GPU and 14-CPU system demonstrate an average of 17.4% performance improvement for CPUs, 10.8% performance improvement for GPUs, and 28.9% LLC energy saving compared to SRAM based LLC design.

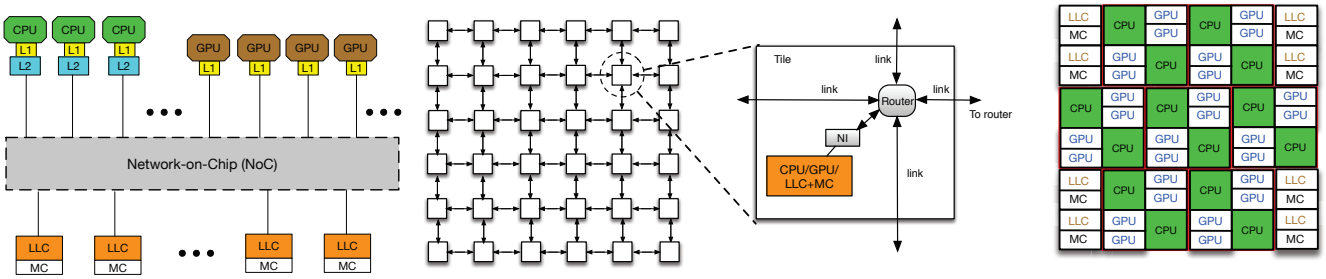
I. Introduction

Heterogeneous multi-cores that integrate CPUs and GPUs on the same chip have been recently used on all kinds of computing platforms such as handheld devices, personal computers, servers, and gaming consoles. Integrated designs (e.g., Intel Haswell [16] and NVIDIA Denver Project [8]) allow faster communication between CPU and GPU memory. Moreover, new integrated architectures such as HSA [24] (employed in some APU models such as the AMD A10-7850K APU [1]) provide a unified virtual address space and a programming model for CPU and GPU applications. These tightly integrated architectures lead to more efficient communication between CPU and GPU applications and better programmability. With such tight integration, it is beneficial to explore the possibility of sharing resources between CPUs and GPUs, such as the main memory, the last-level cache (LLC), and even the on-chip interconnect. However, this integration causes severe contention in the shared resources, and thus opens up new challenges in exploring the design trade-offs as well as new research problems in controlling and reducing the contention in shared resources.

978-1-5090-3508-3/16/$31.00

An important shared resource in heterogeneous CPU-GPU architectures is the LLC. For example, the recent Intel i7 6700K processor [18] contains 4 CPU cores and an Intel HD graphics 530 (gen9), and there is a shared LLC that is not only used for sharing compute data, but also for graphics, which is usually not accessed by the CPU. The LLC reduces overall memory latency, which is critical for CPU performance, and acts as a filter to reduce the pressure on DRAM bandwidth, which is crucial for GPU performance. However, due to the high amount of thread-level parallelism (TLP) available in GPU applications, the high number of requests originating from GPUs exhaust the LLC capacity. Furthermore, CPU and GPU requests interfere with each other, leading to degradation in both CPU and GPU performance. While alternative solutions to this contention problem such as employing cache partitioning [27] or concurrency management [23] alleviate this problem or mitigate its effects, a more viable option to attack this problem is to eliminate it at its source, which is the limited LLC capacity. A naive approach to the problem posed by the limited LLC capacity is to employ a larger LLC, but it comes at the cost of extra access latency, higher dynamic, and leakage power as well as area overhead, and thus leads to design trade-offs.

Instead, we explore the replacement of SRAM based LLC by emerging non-volatile memories (NVM) for heterogeneous multicores. Unlike traditional SRAM that uses electric charges to store information, emerging NVMs (e.g., PCM, STT-RAM, ReRAM, etc.) use resistive storage in a cell, with considerably higher cell density and near-zero leakage in the data array. Therefore, NVMs can potentially replace SRAM to increase the LLC capacity without incurring significant area or power overhead. Among alternative NVM technologies, **STT-RAM** is particularly promising to replace SRAM in LLCs because of its high cell density, non-volatility feature, and low leakage power consumption. Moreover, STT-RAM shows higher endurance [6], [19] compared to other NVM, which makes it more attractive to design on-chip caches that must endure frequent accesses. Because of these benefits, industry has recently explored the opportunity of using STT-RAM as an SRAM LLC replacement for CPUs, such as Toshiba's recent efforts [36], [35]. Therefore, in this work, we take an initiative step to design STT-RAM based LLC for



(a) Heterogeneous multicore architecture (b) Network-on-Chip (c) Layout

Fig. 1: Baseline heterogeneous multicore system with 14 CPU cores, 28 GPU cores, and 8 LLC slices. They are connected by a 6×6 mesh network, with each router connected to either a CPU, two GPUs, or an LLC+MC slice.

heterogeneous multicores.

However, a direct replacement of SRAM LLC by STT-RAM will not work effectively for heterogeneous multicores. First, although STT-RAM can provide larger shared LLC capacity, it cannot completely prevent the large number of GPU requests from occupying the majority of LLC, which would hurt latency-sensitive CPU applications that heavily depend on LLC hits. Second, STT-RAM comes with latency and energy overhead associated with its write operations. This problem gets worse for heterogeneous multicores because, apart from CPU write requests, GPUs typically generate more frequent write requests. Due to these two main reasons, packets from both CPU and GPU applications, as well as read and write requests interleave in NoC. This interleaving causes the pattern of requests arriving at the LLC to be not favorable for STT-RAM, limiting the true potential of STT-RAM LLC in heterogeneous multicores.

Therefore, instead of simply replacing SRAM by STT-RAM as the shared LLC, we first analyze the interference of CPU and GPU traffic, and then propose necessary solutions at the NoC level to tackle the obstacles of employing STT-RAM LLC for heterogeneous multicores. Based on our key observation that expensive STT-RAM write requests are not as much performance critical in GPUs as in CPUs, and because of the latency tolerance disparity between CPU and GPU applications, we propose a customized NoC allocator design, called OSCAR, which consists of an **asynchronous batch scheduling** policy to dynamically batch CPU and GPU packets based on their traffic pattern, and a **priority-based allocator** to order different packets with mixed criticality inside each batch. OSCAR reshapes the pattern of requests to the STT-RAM LLC for better cache utilization. Overall, our goal is to enhance the performance of both CPUs and GPUs while reducing the energy consumption of LLC, thus designing an energy-efficient heterogeneous multicore system.

Our main contributions in this paper are as follows:

- We show that the capacity of LLC is a performance limiting factor in heterogeneous multicores. Therefore, we take an initiative step to replace SRAM based LLC with STT-RAM for higher capacity and lower leakage power.
- We reveal a two-dimensional interference problem with heterogeneous multicores with STT-RAM LLC: latency-sensitive CPUs vs. bandwidth-sensitive GPUs, and asymmetric latency of read/write packets due to STT-RAM. Thus, instead of simple adoption of emerging NVM technology, we focus on *re-shaping the network traffic* to favor STT-RAM caches, so that the full potential of STT-RAM LLC can be maximized to support heterogeneous multicores.
- Instead of strict isolation between CPU and GPU traffic by employing two separate sub-networks and statically partitioning the network bandwidth, we propose an asynchronous batch scheduling policy to provide fair bandwidth allocation in one shared network, which dynamically groups CPU and GPU packets in batches according to the traffic pattern.
- We further modify the switch allocator to provide fine-grained priority based allocation inside each batch of packets, taking into account the fact that GPU applications are more latency-tolerant than CPU applications, and that the high overhead of STT-RAM write requests has significant performance impact on CPU applications as opposed to little impact in GPU applications.
- We perform an extensive evaluation of our proposal and show that our techniques provide 17.4% and 10.8% average performance improvement for CPU and GPU applications, respectively, and 28.9% LLC energy saving, compared to the conventional SRAM based LLC design.

II. Analysis of Heterogeneous Multicores

In this section, we show an example heterogeneous multicore system with shared NoC/LLC and examine the key challenges we are concerned with. Specifically, we analyze the impact of GPU applications on CPU performance and explore potential solutions to address the problems.

A. A Case for Heterogeneous Multicore Systems

As shown in Figure 1a, the heterogeneous multicore architecture places throughput-optimized GPU cores and latency-optimized CPU cores on the same chip, and connects these cores to the shared LLC and memory controllers (MCs) via an interconnect. In order to provide a scalable design for such heterogeneous architectures, we use a tiled architecture and connect all the components via Network-on-Chip (NoC). Figure 1b depicts a 6x6 mesh network interconnected through

routers and links. All the core, cache, and MCs are then attached to the routers for communication.

In our baseline layout, as shown in Figure 1c, a total of 14 CPUs and 28 GPUs are connected to 8 LLC slices through a 6x6 mesh NoC. It consists of 7 processing tiles, where each tile has 4 GPU and 2 CPU cores. We choose a GPU to CPU core ratio of 2:1 because a single GPU core (i.e., streaming multiprocessor, SM) in Fermi GF110 GPU (45nm technology) occupies roughly half the area of one Intel Nehalem CPU core (45nm technology). The LLC, shared by both CPUs and GPUs, is distributed and each slice is directly attached to an MC. The detailed architecture configuration (Table I) and workload specification (Tables III and IV) will be discussed in Section V. Note that we categorized the CPU benchmarks based on their L2 cache MPKI (miss-per-kilo-instructions) and selected 14 applications with a wide range of MPKI values. The baseline layout is generic enough to represent tile-based heterogeneous multicores, and is not uniquely binded to our following analysis and methodology. An alternative layout is also explored in Section V-G.

B. Effects of GPU Traffic on CPU Performance

CPU applications tend to be latency-sensitive, whereas GPU applications are more bandwidth-sensitive and can often tolerate long cache/memory access latency because of sufficient thread-level parallelism [17]. These disparities may lead to unpredictable performance when CPU and GPU applications share the on-chip resources such as NoC and LLC. Due to the massive number of threads, it is quite common for GPU applications to access caches much frequently than CPU applications and make the network congested. Therefore, CPU performance is likely to be severely influenced.

To illustrate this, we run a mix of CPU and GPU applications on the baseline heterogeneous multicore system shown in Figure 1. Specifically, for each experiment, *we use the same set of multi-programmed CPU workloads representing different MPKI values with each CPU core running a different program, while all GPU cores cooperatively run a single workload.* Details of the workloads are described in Table IV in Section V-B. Then, we run these CPU applications standalone in the same system by disabling the GPU cores. In this way, we can compare the performance of CPUs with and without GPU interference. For example, Figure 2 shows the performance of individual CPU applications before and after enabling the GPU application `Blackscholes (BLK)`. Different CPU applications are affected by GPU execution at different degrees. Applications such as `povray`, `namd`, and `dealII` suffer much less because of high local L2 cache hit rate. On the contrary, those with high MPKIs (e.g. `omnetpp`, `mcf`, `soplex`) generate more LLC accesses and thus will be affected more severely by the GPU traffic. For example, the performance degradation of `mcf` reaches 75.5%. On average, the IPC drop is 53.7% across all CPU workloads.

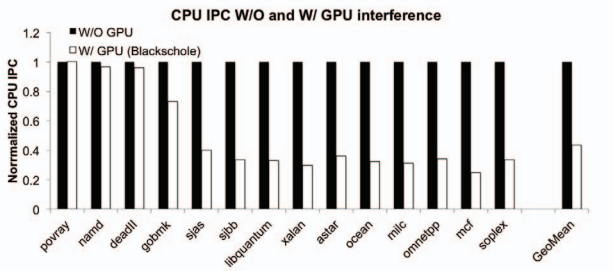


Fig. 2: The IPC of each CPU program when executing without a GPU application, and when executing with `Blackscholes` (a GPU application). These CPU programs represent applications with different L2 MPKI values.

C. Breakdown of Memory Requests

To understand why CPU performance is severely affected by GPU applications, we collect the number of LLC/MC requests generated by CPUs and GPUs. We run the same multi-programmed CPU workloads (See Table IV) with a different GPU workload in all the experiments. Because the number of GPU requests can be several orders of magnitudes larger than that of CPU requests, we show the request count in **logarithmic** scale in Figure 3. Note that we run the simulation until the slowest CPU core issues 5 million instructions or the GPU application finishes execution, whichever comes first. Therefore, even though we are running the same CPU mix, the actually simulated instructions for CPUs will vary with different GPU workloads. We observe that the LLC/MC requests are dominated by GPU applications. On average, there are over 25× more GPU requests than that of CPU requests. In the worst case, when the CPU applications are running with GPU application `BFS`, almost 97% of the requests come from GPUs. Therefore, the majority of the on-chip shared resources will be occupied by GPU requests. Furthermore, these GPU requests might occupy the LLC without actually obtaining substantial benefit.

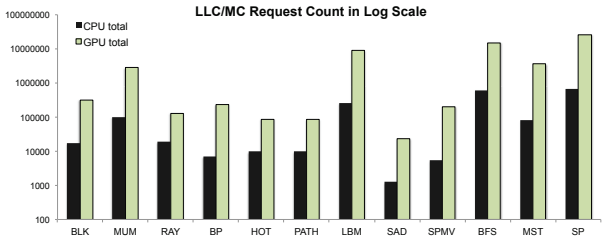


Fig. 3: The total number of LLC/MC requests generated by both CPUs and GPUs, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis). Note that the Y axis is in logarithmic scale.

Note that these requests are generated by local cache misses in CPUs/GPUs. These misses are delivered to the destination nodes for LLC access, and sent off-chip for memory access in case of LLC misses. Correspondingly, the reply messages will be sent back on-chip for cache updates. Therefore, the LLC/MC requests shown in Figure 3 will not only stress the LLC, but also the request and reply NoC that connect all the on-chip components.

D. Ideal Case Analysis: A Larger LLC at No Cost

With a large amount of GPU traffic exhausting the LLC and leaving little share for CPU applications, a straightforward solution is to increase the LLC capacity in order to accommodate more traffic. Here we explore an ideal LLC design which "magically" increases the SRAM based LLC capacity without any constraint (i.e., the increased capacity does not incur additional area/power overhead and can be operated at the same speed as the baseline).

Since we run multi-programmed CPU applications, we use weighted speedup (*WS*) [40] to measure the overall CPU performance. *WS* is obtained by $\sum_{i=1}^{n}(IPC_{i,multiprogram}/IPC_{i,alone})$, where n is the number of CPU applications in the workload. We observe in Figure 4 that CPU performance improves in most of the cases except when running CPUs with GPU applications such as `BlK`, `Ray`, and `SPMV`. On average, we can achieve 8.0%, 14.8%, and 18.6% CPU performance improvement with 16MB, 32MB, and 64MB LLC, respectively. Moreover, the highest performance gains at these three ideal LLC designs are 32.0%, 65.6%, and 76.2%, respectively.

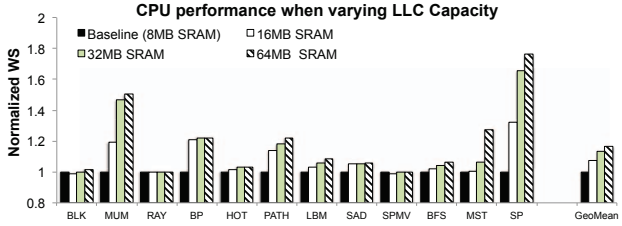


Fig. 4: The impact to CPU performance (weighted-speedup) when increasing the LLC capacity ideally. Each experiment runs the same multi-programmed CPU workloads (see Table IV) with a different GPU workload (X axis).

Similarly, we observe GPU performance under different LLC capacities. On average, the total GPU IPC improvement is 22.2%, 24.5%, and 25.7% for a 16MB, 32MB, and 64MB LLC, respectively, as shown in Figure 5.

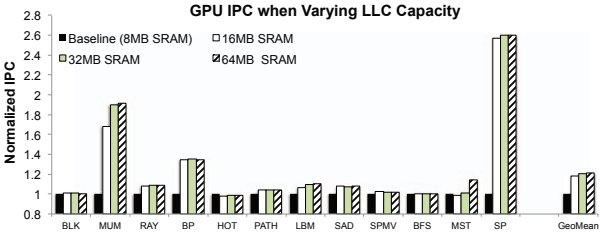


Fig. 5: The impact to GPU performance when increasing the LLC capacity ideally. Each experiment runs the same multi-programmed CPU workloads (see Table IV) with a different GPU workload (X axis).

Therefore, if we can ideally scale the LLC capacity, the performance benefits we can achieve for both CPUs and GPUs are quite promising. However, (1) LLC already occupies a dominant portion of chip area and thus it is impractical to double or even further increase the LLC capacity with traditional SRAM technology. (2) The increase of cache capacity comes with power overhead. This is especially true for leakage power, which is proportional to the cache area. (3) The access latency to each LLC slice will increase as its capacity increases. Alternatively, if we increase the number of LLC slices in our tiled architecture, the size of the NoC will also increase, which in turn hurts the system performance due to the longer communication distance.

III. Replacing SRAM LLC by STT-RAM

As we can see from the previous section, designing a larger LLC can potentially mitigate the CPU/GPU contention problem. However, this approach is not practical for conventional SRAM based LLC design. Therefore, as a replacement of SRAM, we explore emerging non-volatile memories like STT-RAM for LLC design in heterogeneous multicores.

A. Motivation for Leveraging STT-RAM Technology

Unlike the traditional SRAM and DRAM technologies that use electric charges to store information, STT-RAM uses Magnetic Tunnel Junctions (MTJs) for binary storage. STT-RAM has the following advantages over SRAM or other emerging non-volatile memories, which make it a good candidate to replace SRAM as the last-level cache.

- Like other resistive memories, STT-RAM relies on non-volatile, resistive information storage in a cell, and thus exhibits near-zero leakage in the data array. Figure 6 shows the structure of an STT-RAM cell. It uses a 1T1J structure which comprises of an access transistor and a Magnetic Tunnel Junction (MTJ) for binary storage. An MTJ contains two ferromagnetic layers (reference layer and free layer) and one tunnel barrier layer (MgO). The directions of these two layers determine the low/high resistance of the MTJ, which indicate the "0"/"1" state.

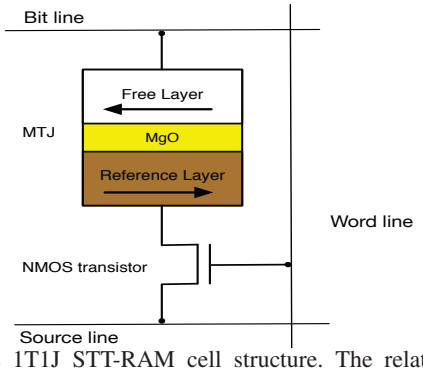


Fig. 6: The 1T1J STT-RAM cell structure. The relative direction of free layer to the reference layer (parallel/anti-parallel) determines the low/high resistance of the MTJ, which indicates "0"/"1" binary storage.

- STT-RAM uses smaller 1T1J cells as opposed to the typical six-transistor SRAM cells. An SRAM cell size is about 120 - 200 F^2 whereas an STT-RAM cell size is about 6 - 50 F^2 [6]. As a result, for the same capacity as SRAM, the dense STT-RAM cells can cut the cache area budget.
- The endurance of STT-RAM (10^{15} writes [6], [19]) is significantly higher than other NVMs (e.g., 10^9 writes for PCM). This makes STT-RAM superior than other NVMs to handle frequent cache accesses in heterogeneous multicores.

However, although STT-RAM read operations achieve comparable read latency and energy as SRAM, **the write latency and the write energy of STT-RAM are significantly higher than an SRAM access**, because a strong current is required to reverse the magnetic direction of the MTJ in order to write "0" or "1" into an STT-RAM cell.

B. Same Area LLC Replacement

Here, we analyze the performance benefit of replacing SRAM LLC by STT-RAM for heterogeneous multicore systems. We assume the density of STT-RAM is 4× of SRAM (see Table II in Section V). Therefore, for the same area budget as the 8MB baseline SRAM LLC, we can approximately design a 32MB STT-RAM based LLC. For comparison, we also analyze an ideal LLC design with the same capacity (32MB) as STT-RAM but the same speed as SRAM.

Figure 7 shows the overall CPU performance with the STT-RAM replacement. Compared to the ideal LLC, there is a significant performance gap due to the write latency overhead of STT-RAM. For some cases like `BLK` and `HOT`, the overall performance of CPU applications even drops below the baseline, meaning the write latency overhead outweighs the capacity benefit of STT-RAM. Overall, a simple replacement of the baseline SRAM by STT-RAM increases the overall CPU performance by about 5.8%. On the other hand, due to the long write latency of STT-RAM, we observe a 6.5% performance loss compared to the ideal LLC design.

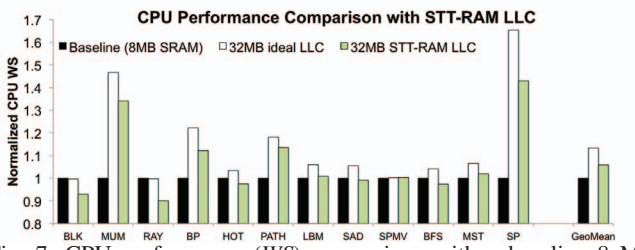


Fig. 7: CPU performance (*WS*) comparison with a baseline 8 MB SRAM LLC, an ideal LLC with the capacity (32 MB) of STT-RAM and the speed of SRAM, and a 32 MB STT-RAM based LLC. Each experiment runs the same multi-programmed CPU workloads (see Table IV) with a different GPU workload (X axis).

As for GPU performance, we conduct similar analysis and present the results in Figure 8. Different from our observation for CPUs, the impact of write latency overhead to GPUs is rather random. Some applications (e.g., `SP`) experience performance drop as compared to the ideal SRAM design. Some applications (e.g., `RAY`, `BFS`) drop below baseline slightly. Interestingly, there are several benchmarks (e.g., `MUM`, `SPMV`) that even gained a bit performance improvement compared to the ideal LLC case.

Overall, a simple replacement of the baseline SRAM by STT-RAM will increase the GPU performance by 11.2%. When compared to the ideal LLC case, the performance degradation due to the long write latency is only 1.1% on average. The observation validates the latency tolerance of GPUs, as their performance is determined by the abundant thread-level parallelism which keeps the core busy and hides some of the cache/memory access latency. *In summary, we demonstrate that the high overhead of STT-RAM write requests have a significant performance impact on CPU applications as opposed to little impact in GPU applications.*

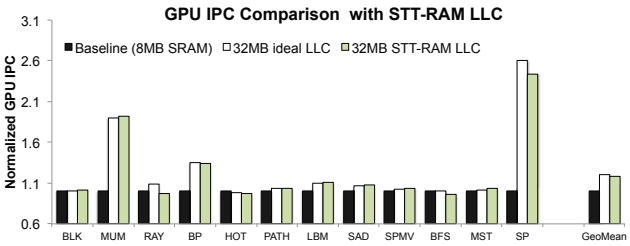


Fig. 8: GPU performance comparison with a baseline 8 MB SRAM LLC, an ideal LLC with the capacity (32 MB) of STT-RAM and the speed of SRAM, and a 32 MB STT-RAM based LLC. Each experiment runs the same multi-programmed CPU workloads (see Table IV) with a different GPU workload (X axis).

C. Challenges with STT-RAM Based LLC

Asymmetric Read/Write Performance: As shown in Figure 7, the STT-RAM write overhead costs a 6.5% performance drop on average for CPUs, with the worst-case performance drop being 13.5%. To better understand the behavior of reads and writes in the system, we divide the total LLC/MC accesses (as shown in Figure 3) into reads and writes. The distribution of these requests are shown in Figure 9. We observe that, for the CPU mix, 24% of LLC/MC accesses come from write operations on average. For GPU workloads, the ratio of writes varies from 0.2% to 93.4%. Therefore, the impact of slow writes cannot be ignored.

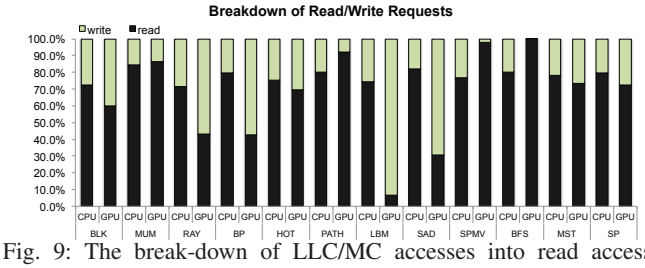


Fig. 9: The break-down of LLC/MC accesses into read accesses and write accesses, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

Although writes are not in the critical path during execution, the long write duration will block the critical reads from accessing the shared LLC [42], [33], which in turn hurts the system performance. This is especially true for the latency-sensitive CPU applications, because the critical CPU read requests will not only interfere with CPU write requests, but also will be severely blocked by the enormous GPU requests.

Requirements for NoC support: As illustrated before, the true potential of STT-RAM LLC is still limited due to the write latency overhead associated with STT-RAM, especially for CPUs. However, addressing the problem of asymmetric read/write performance is not straightforward for heterogeneous multicores, as these asymmetric read/write operations belong to heterogeneous CPU/GPU applications that generate heavily unbalanced number of requests. Now that the cache hit rate can be improved by the increased STT-RAM capacity,

the communication backbone (NoC) that routes all these requests can potentially become the performance bottleneck. Specifically, the high amount of GPU traffic may quickly occupy most of the NoC resources such as buffers, and even lead to network congestion or saturation.

To analyze the NoC performance under STT-RAM based LLC, we evaluate the average network latency for the entire system. As shown in Figure 10, there is clearly a significant latency gap between running CPUs W/ and W/O GPU applications. On average, the NoC latency is increased by 4.0× with the involvement of GPUs. Compared to the 8MB SRAM baseline, the 32MB STT-RAM based LLC can reduce the average NoC latency by 17.4%, but still experience a significant 3.3× NoC performance degradation compared to the W/O GPU case.

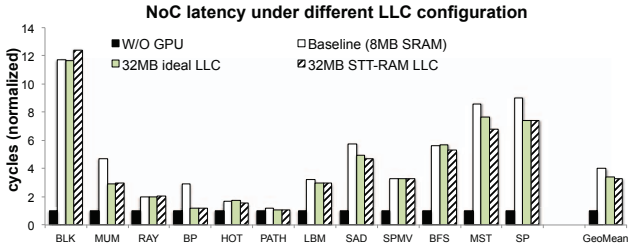


Fig. 10: Average NoC latency under different LLC configurations, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

This means, even though the replacement of SRAM LLC by STT-RAM achieves some performance gains (5.8% for CPUs, and 11.2% for GPUs), the latency reduction in the NoC is still negligible. Specifically, NoC will potentially become the performance bottleneck because LLC/MC accesses will be delayed in the network for 3.3× longer time compared to a network without GPU traffic. Note that we started off with a relative large baseline NoC bandwidth. Our link width is 32 bytes, which is a reasonable design and offers 2× to 4× network bandwidth of many NoC configurations with 16B [33], [28], [37] and 8B [46] link width. Therefore, to fully utilize the potential of STT-RAM based LLC for larger performance speedup, a robust NoC support is required to reshape the pattern of requests to the STT-RAM LLC, being aware of the heterogeneous CPU/GPU traffic with asymmetric STT-RAM read/write requests.

IV. Network-on-Chip Management Policies

In this section, we propose the key design parts of OSCAR to fully explore the potential of STT-RAM based LLC in heterogeneous multicores through NoC optimization, which consists of a *strong ordering* of network packets through asynchronous batch scheduling, and a *weak ordering* of packets inside each batch through priority-based allocation.

A. Strong Ordering with Batch Scheduling

A straightforward way to mitigate the interference of CPU and GPU packets is network isolation. In this section, we will first discuss a strong isolation of network packets by employing separate sub-networks, which statically partitions the network bandwidth. Then we propose a novel asynchronous batch scheduling which can **dynamically** allocate network bandwidth to CPU and GPU packets using a single shared network. It groups CPU and GPU packets into batches and enforces strong ordering between batches to support differentiated bandwidth allocation.

1) Multiple Independent Networks

A simple scheme to handle network interference is to enforce network isolation between different applications. Similar to the multi-network designs [34], [11] for CPUs, we can design separate networks for CPU and GPU traffic. As shown in Figure 11, when the network interface (NI) receives a local packet, it decides which network this packet should be steered to, based on the packet type (CPU or GPU) indicated in the packet header. For fair comparison, we assume a fixed total channel width (aka. phit width) for CPU and GPU network combined, thus the limited channel bandwidth should be statically partitioned to each network.

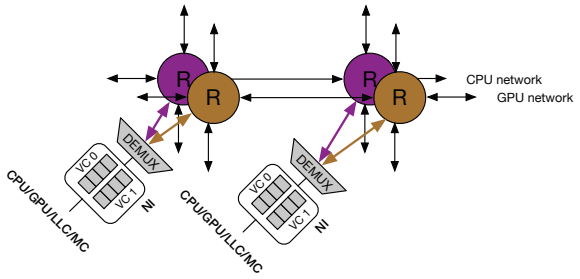


Fig. 11: Schema of network isolation: Separate CPU and GPU networks are connected to the local nodes through shared network interfaces (NIs). Packets will be steered into different networks based on their packet types.

2) Asynchronous Batch Scheduling

The aforementioned multi-net design provides strict isolation between CPU and GPU traffic, but may fail to satisfy various application demand due to static partitioning of network bandwidth. Instead, we still use a single shared network, but design a novel technique, called asynchronous batch scheduling, to allow dynamic bandwidth allocation to CPU and GPU packets. Specifically, our asynchronous batch scheduling policy consists of two techniques: (1) At each input port, always reserve one virtual channel (VC) for CPU traffic only, while the rest of the VCs are accessible to both CPU and GPU packets, and (2) batch CPU and GPU packets for fair scheduling with strong ordering between different batches. Specifically, the reserved VC mitigates starvation of CPU packet, because GPU packets are orders of magnitude more than CPU packets as shown in Figure 3. CPU and GPU packets are grouped into mini-batches to enable differentiated bandwidth allocation according to the traffic pattern. The strong ordering is between different batches, namely one batch has to finish before the other batch starts. However, inside each batch, flits are scheduled *asynchronously* to avoid starvation (i.e., a flit does not have to wait for the entire batch to be filled out before it can proceed).

Figure 12 demonstrates our router microarchitecture with asynchronous batch scheduling. Incoming flits are buffered in the input port which consists of multiple virtual channels (VCs), and then go through VC allocation (VA) and switch allocation (SA) before traversing the switch (ST) and reaching the output port. A batch is a collection of flits (fixed size F) that is delivered in entirety. We use four VCs per input port, with VC 0 reserved for CPU packets. In each batch, a fixed share is allocated to CPU (N_C) and GPU (N_G). In this example, $N_C = 1$, $N_G = 4$, and $F = 5$, which indicates 20% of network bandwidth is allocated to CPU traffic.

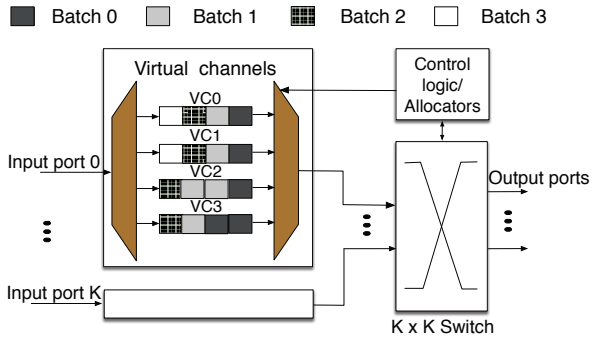


Fig. 12: The router microarchitecture with asynchronous batch scheduling. We use a generic router design with 4 virtual channels (VCs) per input port. VC 0 is reserved for CPU packets only, while other VCs are shared by CPU and GPU packets. 4 batches are shown with a batch size of 5 flits. Batch 0 will be delivered in entirety before processing the next batch.

Initially all VCs are empty and no batch contains any flits. When a source, say a CPU, pushes a flit into the VC 0, it marks it as belonging to batch 0. Further incoming GPU flits will be written into other VCs and fill up batch 0. After that, following flits will be written into batch 1, until batch 1 is filled up with the specified CPU and GPU share. This process is repeated until the entire input buffers are full. In our batch-based scheduling, all flits belonging to the same batch will be delivered together, and batches are processed according to their numbers. For example, in Figure 12, batch 0, 1, 2, and 3 will be delivered in order. Compared to time-division multiplexing or traditional round-robin scheduling, our batch-based scheduling enables a flexible allocation of network resources to different sources, without enforcing strict timing constraint. In the meantime, to avoid starvation, we allow asynchronous processing of flits inside each batch, so that a flit does not have to wait for other flits to fill up the entire batch before it can proceed.

In addition, since the number of CPU packets versus GPU packets is application-dependent, we keep the batch size constant but dynamically change the share of CPU and GPU packets in each batch in an epoch basis. We use a sample window size of $100\mu s$ and use two counters to track the number of CPU requests and GPU requests at each input port, respectively. Then the corresponding ratio will be adjusted in a batch and the new batch composition will be used at the beginning of the next epoch.

B. Weak Ordering through Network Prioritization

Our asynchronous batch processing strategy groups CPU and GPU packets into batches. However, inside a batch, there is still contention of CPU and GPU packets. Moreover, the batch scheduling process is agnostic about the asymmetric latency of read/write packets. Therefore, we further explore network prioritization to provide weak ordering of network packets inside each batch. Since these packets have different criticality to the system performance as illustrated before, an appropriate prioritization scheme is required to allocate sufficient NoC resources to those critical packets.

Figure 13a depicts a generic NoC router. Specifically, in the virtual channel allocation (VA) and switch allocation (SA) stages, the allocators determine which request will be granted, thus we can prioritize the network packets by integrating the priorities into the existing allocation policies.

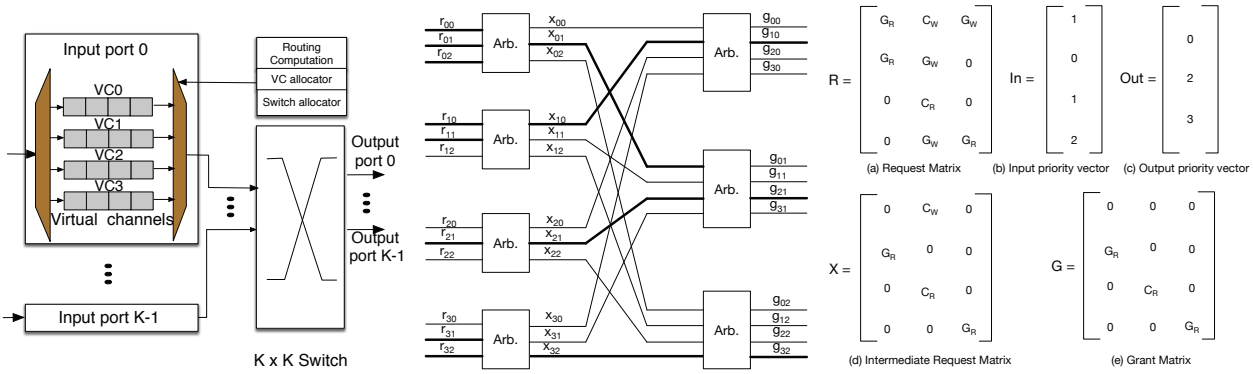
1) Prioritizing CPU over GPU

As shown in Figure 3, the number of GPU packets in the network can be several orders of magnitude higher than that of CPU packets. As a result, GPU packets will occupy most of the NoC resources (e.g., input buffers), and frequently win the allocation in the router which in turn block the transmission of CPU packets.

Therefore, a straightforward solution is to prioritize CPU packets over GPU packets. In this way, the CPU packets will be delivered faster to the destination nodes. Usually, the problem with network prioritization [9] is starvation. In our case, always prioritizing the CPU packets over GPU packets may result in extra delay of some GPU packets. However, the good news is: 1) the number of CPU packets is much lower as shown in Figure 3 and thus the chance of starvation caused by CPU packets is rare, and 2) GPU applications are usually not latency-sensitive, as validated by many prior work [32] and also our results in Figure 8. Thus, the occasional network stalls have little influence on the GPU performance. The main reason that GPUs tolerate miss latency (including NoC, cache, and even memory access latency due to local L1/L2 cache misses) is that, GPU programs usually exhibit much higher TLP than CPU programs, thus the GPU performance would not be significantly influenced by cache misses as long as there are sufficient threads to be executed.

2) Prioritizing Reads over Writes

One important characteristics of STT-RAM is the asymmetric read/write latency. While the read latency is similar to the SRAM counterparts, its write latency is significantly longer. As shown in Figure 7 and Figure 8, the long STT-RAM write latency has a serious impact on the system performance, especially for CPUs. In the scenario where a write operation is followed by several read operations, the ongoing long write operation may block the upcoming read operations and cause performance degradation. Because there are a significant number of write operations (as shown in Figure 9) and that reads are typically on the critical path, an STT-RAM friendly



(a) A generic NoC router microarchitecture (b) A separatable allocator design (c) The request-grant process

Fig. 13: Our priority based allocator design in an NoC router. It uses a separable allocator based on iSLIP but with priority-based matrices during the request-grant process. The order of the priorities are: CPU read (C_R) > CPU write (C_W) > GPU read (G_R) > GPU write (G_W).

prioritization scheme should favor read packets over write packets, for both CPU traffic and GPU traffic.

3) Overall Prioritization Scheme

In our heterogeneous multicore system, the mixed-criticality read/write packets to STT-RAM add another layer of complexity to the NoC resource allocation. As a result, priority-based network allocation is a two-dimensional process: 1) CPU against GPU, and 2) read against write. Nevertheless, because the number of GPU packets are usually several order of magnitudes larger than that of CPU packets as shown in Figure 3, and we have demonstrated in Section III-B that the long STT-RAM write requests have significant performance impact on CPU applications as opposed to little impact in GPU applications, the priority assignment problem can be simplified: *always prioritize CPU packets over GPU packets, then we can prioritize read packets over write packets in each traffic class.* In summary, the priorities of the network packets are ordered as: CPU read (C_R), CPU write (C_W), GPU read (G_R), and GPU write (G_W).

To implement our prioritization policy in the NoC, we integrate two bits to the packet header flit to indicate the packet type (00: CPU read, 01: CPU write, 10: GPU read, 11: GPU write), and then modify the allocation policies in VC allocation and switch allocation of the router datapath. Specifically, our design is based on the popular iSLIP allocator [31]. Therefore, our priority-based allocator is performed as in iSLIP, but with preference towards higher priority packets instead of using existing round-robin arbitration schemes.

Figure 13b shows our 4×3 priority based allocator design which accepts four 3-bit vectors to the input arbiters and generates four 3-bit vectors from the output arbiters. The bold lines show the request-grant process. Specifically, as shown in Figure 13c, the input vectors form a request matrix R. The type of the requests (C_R, C_W, G_R, G_W) are also indicated in the matrix. Based on the request type and our prioritization policy, the input priority vector **In** is $\{1,0,1,2\}$, and the output priority vector **Out** is $\{0,2,3\}$.

Note that our allocator design is a separable allocator in which allocation is performed as two sets of arbitration: one across the inputs and one across the outputs. Therefore, based on the input priority vector **In**, the intermediate request matrix is shown in X. Finally, based on the output priority vector **Out**, the grant matrix G gives the final allocation results, which is consistent with Figure 13b.

The asynchronous batch scheduling ensures a strict ordering between batches, but not among flits in the same batch. Therefore, our priority based allocation scheme as shown in Figure 13 can be integrated with this scheduling algorithm. Specifically, within each batch, flits will be served based on the priority of: CPU read (C_R) > CPU write (C_W) > GPU read (G_R) > GPU write (G_W). We name our integrated approach as OSCAR, because it **O**rchestrates NoC traffic for **S**TT-RAM **C**aches in heterogeneous **AR**chitectures.

V. Evaluation

In this section, we first describe the experimental setup and the benchmarks for evaluation, then conduct performance and energy analysis on our proposed techniques.

A. System Setup

To evaluate our proposed schemes, we integrate GPGPU-sim v3.2.0 [4] with an in-house cycle-level trace-driven x86 CMP simulator. We fast-forward the CPU applications to the ROI for the CPU traces. Then the simulation warms up with CPU execution until the slowest CPU core has completed 500K instructions. After that the GPUs start execution together with CPUs. To measure CPU performance, we run the simulation until the slowest CPU core issues 5 million instructions or the entire GPU application completes execution, whichever comes first.[1] We simulate the heterogeneous multicore system as shown in Figure 1.

Table I shows the detailed CPU, GPU, cache, NoC, and memory configurations. There are a total of 28 GPU cores and 14 CPU cores. A GPU core contains 32-wide SIMD lanes and is equipped with an instruction cache, private L1 data cache, constant, and texture caches. Each CPU core is a 3-way issue x86 core with private write-back L1 instruction/data cache and

[1] A majority of GPU applications finish execution before the slowest CPU hitting 5 million instructions, as mentioned in Section II-C

L2 caches. The shared LLC is partitioned into 8 modules. For the baseline SRAM based LLC, each module is 1MB and thus the total size of LLC is 8MB. The baseline design also uses a shared 6x6 2D mesh network to connect all the core and cache components. The NoC employs a dimension-order routing algorithm and wormhole flow control. Each network packet contains five flits and each flit is set to be 32-byte long. As for the router microarchitecture, we adopt a classic router design with four-stage pipelines. Each input port of the router contains four virtual channels (VCs). Each VC has a buffer depth of 4 flits. As for the main memory, we use a detailed GDDR5 DRAM timing model. The main memory is also partitioned into 8 banks, with each DRAM bank connected to a memory controller.

TABLE I: Baseline heterogeneous architecture configuration

GPU core	28 shader cores, 1400 MHz, SIMT width = 16x2, Max. 48 warps/core, 32 threads/warp
GPU caches / core	16KB 4-way L1 data cache, 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B line size
CPU core	16 x86 cores, 2000 MHz, 128-entry instruction window, OoO fetch and execution
CPU L1 cache	32KB 4-way, 2cycle lookup, 128B line size
CPU L2 cache	256KB 8-way, 8cycle lookup, 128B line size
Share SRAM LLC	1x8 MB, 128B line, 16-way
NoC	6x6 shared 2D mesh, dimension-order routing, 32B phit width, 4 VCs per port, 4 buffers per VC, islip allocator
Main memory	8 shared GDDR5 MCs, 800 MHz, FR-FCFS, 8 DRAM-banks/MC

To compare SRAM with STT-RAM as the LLC, we list their capacity, latency, and energy numbers [13], [45] under the same area budget, as shown in Table II.

TABLE II: Comparison of SRAM LLC and STT-RAM LLC

Memory type	1MB SRAM	4MB STT-RAM
Read latency (ns)	1.748	2.730
Write latency (ns)	1.491	11.212
Read energy (nJ)	0.054	0.13
Write energy (nJ)	0.051	0.352
Leakage power (mW)	33.750	2.477
Area (mm^2)	0.87	0.791

B. Workloads for Evaluation

We run a wide range of CPU and GPU workloads for performance evaluation. As shown in Table III, we evaluate 12 GPU applications from CUDA SDK [4], Rodinia [7], Parboil [41], and LonestarGPU [5]. For CPUs, we run multi-programmed workloads with a mix of applications from different benchmark suites including scientific, commercial, and desktop applications drawn from the SPEC® CPU 2000/2006 INT and FP suites and commercial server workloads. Furthermore, we conduct workload analysis and select 14 CPU benchmarks that represent a wide range of MPKI values (miss-per-kilo-instruction). The selected CPU benchmarks are listed in Table IV. In our scenario, we are interested in multi-programmed workloads in which CPU and GPU inject traffic at the same time, as compared to letting CPU execute the sequential part and GPU execute the parallel part without overlapping. As for cooperative workloads in which CPU and GPU execute the same context in a coherent APU-like system, network traffic will be even higher due to coherence. For example GPU requests would create additional coherence network traffic from CPU cores as well. So we would see traffic from both CPUs and GPUs at the same time, and a single memory request might result in multiple network messages, leading to even more traffic in the NoC.

TABLE III: GPU benchmarks

#	Suite	Application	Abbr.
1	CUDA SDK	BlackScholes	BLK
2	CUDA SDK	MUMerGPU	MUM
3	CUDA SDK	RAY Tracing	RAY
4	Rodinia	Backpropagation	BP
5	Rodinia	Hotspot	HOT
6	Rodinia	Pathfinder	PATH
7	Parboil	Lattice-Boltzmann Method	LBM
8	Parboil	Sum of Abs. Differences	SAD
9	Parboil	Sparse-Matrix-Mul.	SPMV
10	LonestarGPU	Bread First Search	BFS
11	LonestarGPU	Min. Spanning Tree	MST
12	LonestarGPU	Single-Source Shortest Path	MST

TABLE IV: CPU benchmarks

CPU app. category	Applications	L2 MPKI range
Low	povray, namd, dealII, gobmk	[0.2, 2.3]
Medium	sjas, astar, sjbb, ocean, libquantum, xalan	[4.8, 22]
High	milc, soplex, omnetpp, mcf	[25, 112.4]

C. Evaluation of Network Prioritization

We first study the effect of network prioritization standalone, before exploring our integrated approach OSCAR. Figure 14 compares different prioritization schemes. There are several important observations to be highlighted here: 1) With CPU prioritization, there is a clearly performance improvement for all the benchmarks, the average weighted-speedup improvement is 5.2% and can be as high as 10.2%, compared to the baseline; 2) With read prioritization only, the average performance improvement is small (about 1.6%), because even though read requests are always prioritized, the enormous GPU read requests win most of the shared resources and cause delay to CPU read requests; and 3) When incorporating both prioritization schemes as demonstrated in Figure 13, we can achieve an additional 6.9% CPU performance improvement over the STT-RAM LLC based baseline, with the highest performance increase being 13.5%.

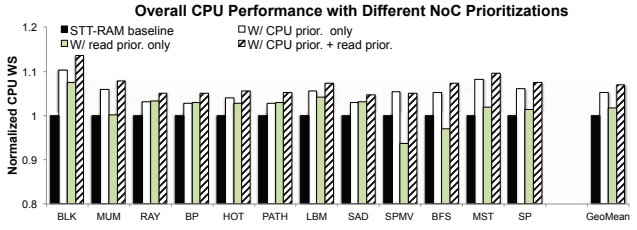


Fig. 14: The overall CPU performance with different NoC prioritization schemes, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

As illustrated before, GPU applications have better latency tolerance. Figure 15 shows that the impact of different prioritization policies on GPU performance is negligible. The average IPC loss is only 1.0% with an integrated CPU and read prioritization, compared to the baseline.

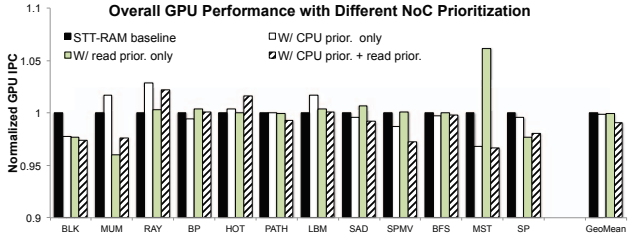


Fig. 15: The overall GPU performance with different NoC prioritization schemes, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

D. Evaluation of Multi-Network Setup

Here we evaluate how network isolation with separate CPU and GPU network would affect the CPU and GPU performance. As mentioned in Section IV-A, we assume the total network channel width is fixed. Furthermore, we divide the total channel width into four shares and assign different shares to the CPU/GPU network. For example, for a total channel width of 256 bits, "3G1C" means 192 bits for GPU network and 64 bits for CPU network. Figure 16 shows the performance results. On average, 3G1C, 2G2C, and 1G3C increase the overall CPU performance by 6.3%, 9.9%, and 4.5%, while decreasing the overall GPU performance by 2.5%, 14.3%, and 48.0%, respectively. A counter-intuitive observation here is that, when the majority of NoC resources is allocated to CPUs (e.g., 1G3C), the CPU performance starts to decrease. This is because most CPU packets stall at the MCs, blocked by GPU packets (due to the reduction of GPU network resources) that are waiting in MCs to be injected into the reply network. In this scenario, increasing CPU network resources causes bandwidth reduction of GPU reply network that in turn blocks CPU packets at the MCs. This phenomenon is also observed by others [3], [14].

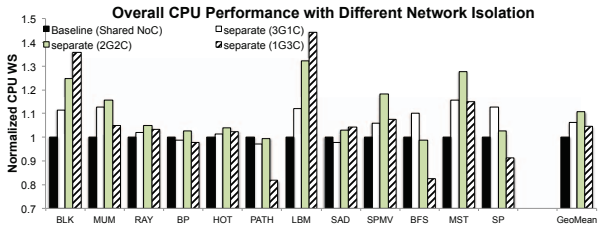


Fig. 16: The impact of network isolation to CPU performance, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

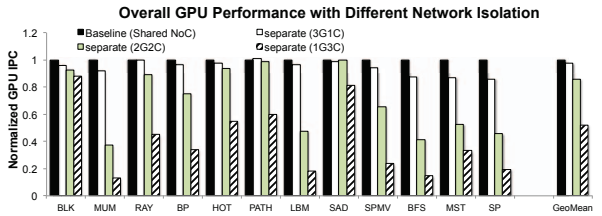


Fig. 17: The impact of network isolation to GPU performance, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

Generally, network isolation benefits CPUs. However, the performance degradation to GPUs will be severe if the majority of channel width is allocated to CPUs. Therefore, such static bandwidth partitioning may result in detrimental performance due to the unpredictable traffic pattern.

E. Evaluation of OSCAR

As illustrated in Section IV-A2, our proposed OSCAR is the integration of asynchronous batch scheduling and priority based allocation techniques, with STT-RAM based LLC. In addition to the conventional SRAM LLC and the direct STT-RAM replacement design, we also compare OSCAR against a read-preemptive [42] scheme for STT-RAM cache and the GPU concurrency management [23] policy. Results are shown in Figure 18 and Figure 19.

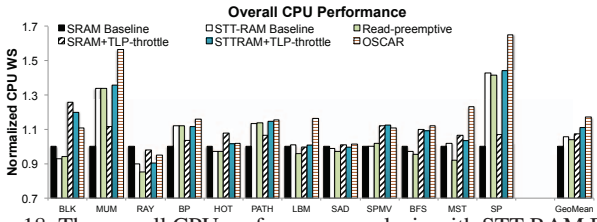


Fig. 18: The overall CPU performance analysis with STT-RAM Based LLC and NoC support, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

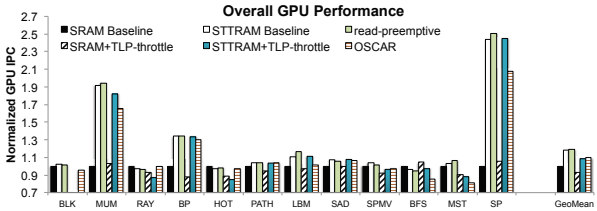


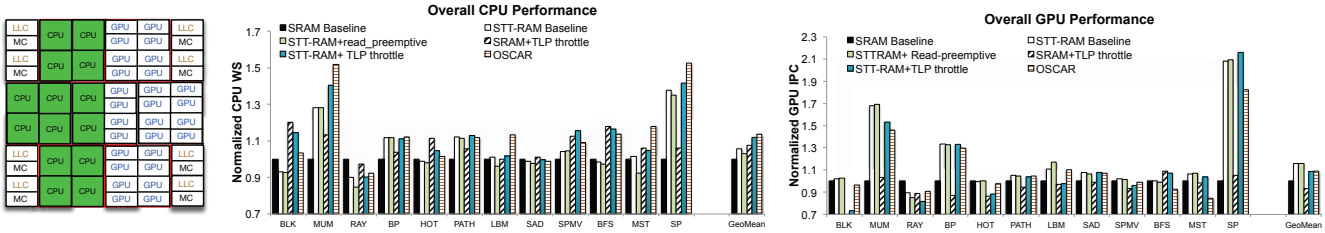
Fig. 19: The overall GPU performance analysis with STT-RAM Based LLC and NoC support, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

For CPUs, OSCAR improves the overall performance by 11.2% over the LLC design with simple STT-RAM replacement. When compared to the traditional SRAM based LLC for heterogeneous multicores, *OSCAR can improve the overall CPU performance by 17.4% on average and up to 67.4%*. In contrast, read-preemptive scheme improves the CPU performance by 3.9%. When GPU concurrency control is coupled with SRAM or STT-RAM, the average performance increase is 7.3% and 11.2%, respectively.

For GPUs, OSCAR only causes 5.6% performance degradation compared to the STT-RAM baseline. *Overall, OSCAR improves the GPU performance by 10.8% compared to the SRAM baseline.* The read-preemptive scheme can increase the STT-RAM performance by 1.4%. The GPU concurrency control incurs nearly 6.8% performance drop for GPUs with a small SRAM based LLC. With a STT-RAM replacement, it can increase the GPU performance by 8.5%.

F. Energy Analysis

Here we evaluate how our design influences the total LLC energy consumption. As shown in Table II, STT-RAM achieves significant lower leakage power compared to the



(a) Layout with clustered CPUs/GPUs on sides

(b) Overall CPU performance with the new layout

(c) Overall GPU performance with the new layout

Fig. 20: Alternative layout with CPUs and GPUs clustered on different sides and the corresponding performance results under different optimizations, when running the same multi-programmed CPU workloads (see Table IV) with different GPU workloads (X axis).

SRAM counterparts. However, the dynamic energy associated with STT-RAM write operations are also much higher than SRAM. Figure 21 shows the *normalized* total LLC energy consumption for the traditional SRAM based LLC and our STT-RAM based design.

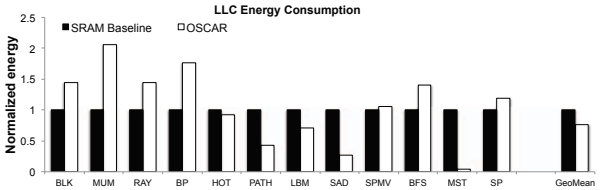


Fig. 21: Comparison of LLC energy consumption

The energy impact of our design is application-dependent. For CPU mixes with GPU applications like `BLK`, `MUM`, `RAY` and `BP`, the overhead of slow writes outweighs the leakage savings, indicating high write density (number of writes per time unit) during execution. On the other hand, CPU mixes with GPU applications like `PATH`, `SAD`, and `MST` achieve significant energy savings with less intensive writes. On average, OSCAR achieves 28.9% LLC energy savings. Although it is out of the scope of this paper, there are existing optimizations to achieve more energy savings by relaxing the non-volatility of STT-RAM [39], [22]. Furthermore, hybrid SRAM/STT-RAM cache designs were also proposed to direct intensive write operations to a small portion of SRAM cache banks to reduce the impact of the energy/latency overhead associated with the write operation of STT-RAM [42], [45]. Nevertheless, our NoC management policy can also be applied to such hybrid LLC cache designs to achieve further performance/energy improvement.

We have implemented the modified allocator in Verilog for the asynchronous batch scheduling and priority based allocation. The area overhead is obtained by synthesizing our Verilog using the Synopsys Design Compiler, with 45nm technology node. In total, our design only adds 0.0052 mm^2 area per router, which is a negligible 1.8% overhead.

G. Sensitivity Study with Alternative Layout

We also evaluate our design on a different layout as shown in Figure 20a, which clusters CPUs and GPUs and places them on opposite sides of the chip. That also creates an asymmetric set of traffic patterns as the high-bandwidth GPU traffic is coming from one side of the chip, and the lower-bandwidth but latency-sensitive traffic comes from the other side.

We conduct similar comparisons over different optimization techniques. Overall, OSCAR performs the best for CPUs, with performance increase of 16.1% across all the workloads. In contrast, the read-preemptive design achieves only 3.1% performance improvement. When GPU concurrency control is coupled with SRAM or STT-RAM, the average performance increase is 7.6% and 11.5%, respectively.

For GPU applications, even though performance degradation happens compared to the STT-RAM baseline as we prioritize CPUs over GPUs, the degradation is insignificant and we can still achieve a 9.8% IPC increase compared to the conventional SRAM based LLC design. The read-preemptive scheme does not give further performance benefit to the STT-RAM in this case. Similarly, the GPU concurrency control incurs about 6.4% performance drop for GPUs with a small SRAM based LLC. With a STT-RAM replacement, it increases the GPU performance by 8.7%.

VI. Related Work

Managing Interference in Heterogeneous Multicores. In heterogeneous multicores, CPU and GPU applications interfere with each other in shared resources. Lee and Kim [27] reduces the performance impact of limited LLC capacity by cache partitioning to isolate CPU and GPU traffic in LLC. Here we address the same problem directly at its source, by employing STT-RAM to increase the cache capacity. Lee *et al.* [29], [28] address the NoC contention by proposing an adaptive VC partitioning mechanism. Yin *et al.* [46] propose to route GPU packets through non-minimal paths because of their latency-tolerance. Our NoC optimizations take STT-RAM properties into account to prioritize reads over writes. Kayiran *et al.* [23] manage the interference by throttling the number of concurrently executing GPU warps. To reduce the interference of CPU and GPU traffic in DRAM, Ausavarungnirun *et al.* [2] develop a memory scheduling technique to group memory requests based on row-buffer locality. There are other works [21], [51], [50] that provide NoC optimizations for GPU systems only.

STT-RAM LLC Management. To address the asymmetric read and write problem in STT-RAMs, Sun *et al.* [42] use a read-preemptive scheme with write buffers which prioritizes

cache read operations to slow write operations. Zhou *et al.* [49] propose early write termination to reduce the STT-RAM write overhead. In contrast, we propose light-weight NoC optimizations to address these problems without incurring extra hardware overhead in the cache. Mishra *et al.* [33] address the slow write problem by scheduling critical read requests in the network to idle STT-RAM banks. However, we deal with not only the interference of read/write requests but also CPU/GPU requests in the network. Samavatian *et al.* [38] explore STT-RAM based L2 cache for GPUs and partition the STT-RAM into high/low retention regions through non-volatility relaxation. While there are many STT-RAM LLC optimization techniques [44], [30], [45], we observe the performance bottleneck has shifted to the network backbone with large enough LLC capacity, thus providing NoC support to maximize the potential of STT-RAM LLC. Meanwhile, there are studies [47], [20] on STT-RAM based buffer designs for NoC.

Application-Aware NoC Optimization. Das *et al.* [9], [10] explore the criticality of different packets and propose prioritization schemes in the NoC routers based on their "stall-time criticality" or "latency slack". In contrast, our NoC management policy is not only application-dependent due to heterogeneous CPU and GPU workloads, but also cache-dependent due to the asymmetric read/write performance of STT-RAM. Other prior works [43], [25], [12], [48] use application-aware NoC optimization techniques to improve reliability or timing-correctness.

There are related work [26], [15], [37] on frame-based scheduling. Regardless of dealing with a new context with heterogeneous workloads, our asynchronous batch scheduling employs different methods. The global-synchronized frame [26], [15] treats the entire NoC as a "MUX" with global synchronization and suffers from large frame sizes, under-utilization of excess bandwidth, etc. The local-synchronized frame [37] schedules packets based on their ages. It is a flow-based scheduling algorithm for IP flows in real-time embedded systems, and does not deal with packets with different priorities/criticalities. Moreover, all the flits in a frame are treated equally in work [37] whereas OSCAR differentiates the criticality of CPU/GPU and read/write flits.

VII. Conclusions

Heterogeneous multicores raise challenges to the shared on-chip resources such as LLC and NoC. We study the replacement of the conventional SRAM based LLC by STT-RAM, which can provide much larger capacity to mitigate the contention of CPU and GPU accesses in LLC while saving leakage power. In addition, we propose asynchronous batch scheduling and priority based allocation schemes in the NoC to address the interference of CPU/GPU traffic and asymmetric read/write operations to STT-RAM. Simulation results demonstrate averaged 17.4% and 10.8% performance improvement for CPUs and GPUs, respectively, and 28.9% energy saving for the entire LLC.

Acknowledgment

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award number DE-SC0013553 with disclaimer at http://seal.ece.ucsb.edu/doe/. It was also supported in part by NSF 1533933, 1461698, 213052, 1500848, 1213052, and 1409095. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. SPEC is a registered trademark of Standard Performance Evaluation Corporation (SPEC). Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] AMD. (2014) Compute Cores. http://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf.

[2] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[3] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective On-chip Networks for Manycore Accelerators," in *MICRO*, 2010.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.

[6] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology Comparison for Large Last-level Caches (L 3 Cs): Low-leakage SRAM, Low Write-energy STT-RAM, and Refresh-optimized eDRAM," in *HPCA*, 2013.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[8] B. Dally, "Project Denver," *Processor to usher in new era of computing.[Online] Available from http://blogs. NVIDIA.com/2011/01/project-denver-processor-tousher-in-new-era-of-computing*, 2011.

[9] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware Prioritization Mechanisms for On-chip Networks," in *MIRCO*, 2009.

[10] ——, "Aérgia: Exploiting Packet Latency Slack in On-chip Networks," in *ISCA*, 2010.

[11] R. Das, S. Narayanasamy, S. Satpathy, and R. Dreslinski, "Catnap: Energy Proportional Multiple Network-on-Chip," in *ISCA*, 2013.

[12] D. DiTomaso, A. Kodi, and A. Louri, "QORE: A Fault Tolerant Network-on-chip Architecture with Power-efficient Quad-function Channel (QFC) Buffers," in *HPCA*, 2014.

[13] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement," in *DAC*, 2008.

[14] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.

[15] B. Grot, S. W. Keckler, and O. Mutlu, "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-chip," in *MICRO*, 2009.

[16] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The Fourth-generation Intel Core Processor," *IEEE Micro*, no. 2, pp. 6–20, 2014.

[17] J. Hestness, S. W. Keckler, and D. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *IISWC*, 2014.

[18] Intel. (2015) The Compute Architecture of Intel Processor Graphics Gen9. https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf.

[19] ITRS. (2013) Process Integration, Devices, and Structures (PIDS). http://www.itrs.net/Links/2013ITRS/2013Tables/PIDS_2013\Tables.xlsx.

[20] H. Jang, B. S. An, N. Kulkarni, K. H. Yum, and E. J. Kim, "A hybrid buffer design with STT-MRAM for on-chip interconnects," in *Networks*

on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on. IEEE, 2012, pp. 193–200.
[21] H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-Efficient On-Chip Interconnect Designs for GPGPUs," in *DAC*, 2015.
[22] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs," in *DAC*, 2012.
[23] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
[24] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," *AMD Fusion Developer Summit*, 2012.
[25] D. Lee, R. Parikh, and V. Bertacco, "Highly Fault-tolerant NoC Routing with Application-aware Congestion Management," in *NOCS*, 2015.
[26] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-synchronized Frames for Guaranteed Quality-of-service in On-chip Networks," in *ISCA*, 2008.
[27] J. Lee and H. Kim, "TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture," in *HPCA*, 2012.
[28] J. Lee, S. Li, H. Kim, and S. Yalamanchili, "Adaptive Virtual Channel Partitioning for Network-on-chip in Heterogeneous Architectures," *TODAES*, vol. 18, no. 4, p. 48, 2013.
[29] ——, "Design Space Exploration of On-chip Ring Interconnection for a CPU–GPU Heterogeneous Architecture," *JPDC*, vol. 73, no. 12, pp. 1525–1538, 2013.
[30] M. Mao, G. Sun, Y. Li, A. K. Jones, and Y. Chen, "Prefetching Techniques for STT-RAM Based Last-level Cache in CMP systems," in *ASP-DAC*, 2014.
[31] N. McKeown, "The iSLIP Scheduling Algorithm for Input-queued Switches," *Networking, IEEE/ACM Transactions on*, vol. 7, no. 2, pp. 188–201, 1999.
[32] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, "Managing Shared Last-level Cache in a Heterogeneous Multicore Processor," in *PACT*, 2013.
[33] A. K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. R. Das, "Architecting On-chip Interconnects for Stacked 3D STT-RAM Caches in CMPs," in *ISCA*, 2011.
[34] A. K. Mishra, O. Mutlu, and C. R. Das, "A Heterogeneous Multiple Network-on-chip Design: An Application-aware Approach," in *DAC*, 2013.
[35] H. Noguchi, K. Ikegami, N. Shimomura, T. Tetsufumi, J. Ito, and S. Fujita, "Highly Reliable and Low-power Nonvolatile Cache Memory with Advanced Perpendicular STT-MRAM for High-performance CPU," in *VLSI*, 2014.
[36] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita, "7.5 A 3.3ns-access-time 71.2 uW/MHz 1Mb Embedded STT-MRAM Using Physically Eliminated Read-disturb Scheme and Normally-off Memory Architecture," in *ISSCC*, 2015.
[37] J. Ouyang and Y. Xie, "Loft: A High Performance Network-on-chip Providing Quality-of-service Support," in *MICRO*, 2010.
[38] M. H. Samavatian, H. Abbasitabar, M. Arjomand, and H. Sarbazi-Azad, "An Efficient STT-RAM Last Level Cache Architecture for GPUs," in *DAC*, 2014.
[39] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing Non-volatility for Fast and Energy-efficient STT-RAM Caches," in *HPCA*, 2011.
[40] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," in *ASPLOS*, 2000.
[41] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, 2012.
[42] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs," in *HPCA*, 2009.
[43] J. Sun, R. Lysecky, K. Shankar, A. Kodi, A. Louri, and J. Roveda, "Workload Assignment Considering NBTI Degradation in Multicore Systems," *JETC*, vol. 10, no. 1, p. 4, 2014.
[44] J. Wang, X. Dong, and Y. Xie, "OAP: An Obstruction-aware Cache Management Policy for STT-RAM Last-level Caches," in *DATE*, 2013.
[45] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive Placement and Migration Policy for an STT-RAM-based Hybrid Cache," in *HPCA*, 2014.
[46] J. Yin, P. Zhou, A. Holey, S. S. Sapatnekar, and A. Zhai, "Energy-efficient Non-minimal Path On-chip Interconnection Network for Heterogeneous Systems," in *ISLPED*, 2012.
[47] J. Zhan, J. Ouyang, F. Ge, J. Zhao, and Y. Xie, "DimNoC: A dim silicon approach towards power-efficient on-chip network," in *DAC*. IEEE, 2015, pp. 1–6.
[48] J. Zhan, N. Stoimenov, J. Ouyang, L. Thiele, V. Narayanan, and Y. Xie, "Designing energy-efficient NoC for real-time embedded systems through slack optimization," in *DAC*, 2013, p. 37.
[49] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy Reduction for STT-RAM Using Early Write Termination," in *ICCAD*, 2009.
[50] A. K. Ziabari, J. L. Abellán, Y. Ma, A. Joshi, and D. Kaeli, "Asymmetric NoC Architectures for GPU Systems," in *NOCS*, 2015.
[51] A. K. K. Ziabari, J. L. Abellán, R. Ubal, C. Chen, A. Joshi, and D. Kaeli, "Leveraging Silicon-photonic NoC for Designing Scalable GPUs," in *ICS*, 2015.