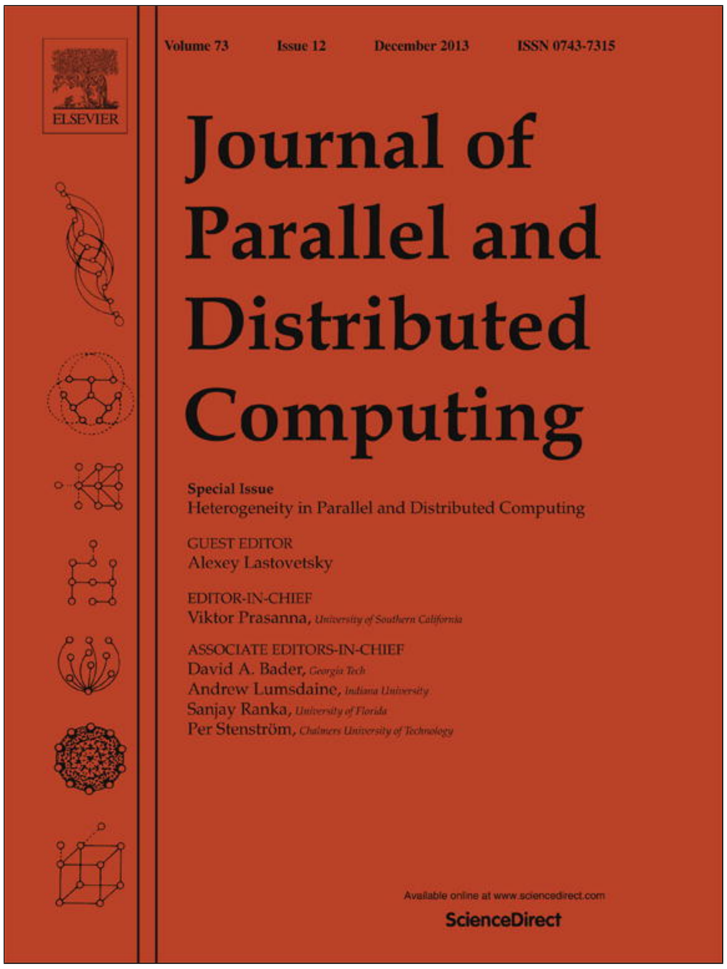


J. Parallel Distrib. Comput. 73 (2013) 1525–1538

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Design space exploration of on-chip ring interconnection for a CPU–GPU heterogeneous architecture

Jaekyu Lee [a,*], Si Li [b], Hyesoon Kim [a], Sudhakar Yalamanchili [b]

[a] School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, United States
[b] School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, United States

HIGHLIGHTS

- We study the possible problems and design space exploration of the on-chip network in CPU–GPU heterogeneous architecture.
- We examine different placements for the component (CPU, GPU, L3 cache, and memory controllers).
- We discover that the resource partitioning, in particular router virtual channels, shows effectiveness to prevent interference.
- We discover that heterogeneous configurations can improve the performance of the system while not incurring too much overhead.
- Based on our findings, we suggest an optimal ring network configuration.

ARTICLE INFO

Article history:
Received 15 October 2012
Received in revised form
2 May 2013
Accepted 26 July 2013
Available online 14 August 2013

Keywords:
Heterogeneous architecture
On-chip interconnection network
Design space exploration

ABSTRACT

Incorporating a GPU architecture into CMP, which is more efficient with certain types of applications, is a popular architecture trend in recent processors. This heterogeneous mix of architectures will use an on-chip interconnection to access shared resources such as last-level cache tiles and memory controllers. The configuration of this on-chip network will likely have a significant impact on resource distribution, fairness, and overall performance.

The heterogeneity of this architecture inevitably exerts different pressures on the interconnection due to the differing characteristics and requirements of applications running on CPU and GPU cores. CPU applications are sensitive to latency, while GPGPU applications require massive bandwidth. This is due to the difference in the thread-level parallelism of the two architectures. GPUs use more threads to hide the effect of memory latency but require massive bandwidth to supply those threads. On the other hand, CPU cores typically running only one or two threads concurrently are very sensitive to latency.

This study surveys the impact and behavior of the interconnection network when CPU and GPGPU applications run simultaneously. Among our findings, we observed that significant interference exists between CPU and GPU applications and resource partitioning, in particular virtual and physical channel partitioning, shows effectiveness to solve the interference problem. Also, heterogeneous link configurations show promising results by optimizing traffic hotspots in the network. Finally, we evaluated different placement policies and found that how to place different components in the network significantly affects the performance. Based on these findings, we suggest an optimal ring interconnect network. Our study will shed light on other architectural interconnection studies on CPU–GPU heterogeneous architectures.

1. Introduction

The demand for more computational power never ends. Traditionally, growth in computational power was carried out by ever increasing clock frequency until the power wall was hit. To circumvent this barrier, future chip multiprocessors (CMPs) will try to integrate heterogeneous mixtures of architectures on the same chip where one type of architecture is more power efficient at a subset of tasks while the effort on increasing frequency continues with new technologies. A general-purpose GPU (GPGPU) is one such example that is more power efficient for tasks involving massive data-level parallelism. Incorporating a GPU architecture into CMPs is the next logical step. Recent examples include Intel's Ivy Bridge [24], AMD's Fusion [3], and NVIDIA's Denver [36] project. In these architectures, various on-chip resources are shared by CPU and GPU cores, including the last-level cache, memory controllers, and DRAM. Access to these shared resources is controlled

* Corresponding author.
E-mail address: jq.lee17@gmail.com (J. Lee).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.07.014

by the on-chip interconnection, which has a significant impact on resource distribution, fairness, and overall performance.

To improve network and overall performance, many researchers have proposed a variety of mechanisms involving topologies, adaptive routing, scheduling, and arbitration policies. Many different topologies [7,29,11,16,40] have been proposed to improve performance. They tend to focus on either all CPU-based architectures or on specialized SoCs operating running a narrow spectrum of applications. Various adaptive routing algorithms are introduced to improve performance [31,21,22,33]. A significant amount of work is proposed in router arbitration policies [14,15]. Recently, proposals on heterogeneous interconnection configurations have been introduced [32,18].

On-chip CPU–GPU heterogeneous architectures as well as their interconnections, however, are not as well studied. While we anticipate that they will have characteristics similar to more conventional CMP networks, we also expect additional complexities involving resource sharing mechanisms caused by the opposing memory demands exerted by applications running on the two architectures. CPU and GPU architectures possess fundamentally diametric network demands. CPU cores rely on instruction-level parallelism (ILP), large caches, and speculative mechanisms to achieve high performance in serial execution. Since CPUs usually operate on a very small number of threads, when one is stalled on a memory access, it incurs a large penalty until that access is satisfied. At the opposite end of the spectrum, GPUs target data-parallel applications to achieve high throughput. They exchange large caches and other power-consuming mechanisms for more processing elements (PE) to execute on multiple data sets in parallel. Although a hardware-managed cache exists, GPUs mainly leverage the zero overhead, single-cycle context switch capability to remove latency introduced by long latency instructions and their dependencies. When a thread is blocked, the instruction scheduler context switches to the next available thread or group of threads. Due to the often massive number of potential threads waiting for execution, GPUs can execute many memory instructions concurrently and also in parallel, thereby achieving higher bandwidth requirements than CPUs.

In this paper, we evaluate the NoC behavior of this CPU–GPU heterogeneous architecture. However, we limit our study to the ring network. Although the ring network is not scalable with many cores, it is still relevant because most commercial processors currently employ a ring network with a reasonable number of CPU and GPU cores. Based on a network characterization of the Cell processor [2] and Intel's Xeon Phi coprocessor [26], we believe that the ring network will be used at least for the next few years until the number of cores breaches a threshold of 10 or 12 cores.

In this study, we seek answers to the following questions: (1) How does the ring interconnection behave in CPU–GPU heterogeneous workloads? and (2) What is the best ring router configuration in heterogeneous workloads. We first study the impact of a variety of network resources and mechanisms on the system performance of CPU and GPGPU applications running separately, including the number of virtual channels, link width, link latency, and different placements. Then, we evaluate the resource sharing in the interconnection when both applications are running concurrently. In particular, we study virtual and physical channel partitioning between CPU and GPU cores, heterogeneous link configuration for each router, arbitrations, routing algorithms, and placements.

Based on the findings from empirical studies, we suggest an enhanced ring interconnection network in CPU–GPU heterogeneous architectures that improves performance by 22%, 19%, and 16% for one-CPU/one-GPU, two-CPU/one-GPU, and four-CPU/one-GPU workloads, respectively. We believe our studies will lead to further architectural studies in this area of on-chip interconnection for a CPU–GPU heterogeneous architecture.[1]

2. Background

2.1. CPU–GPU heterogeneous architecture

Intel's Sandy Bridge [25], which was released in 2011, is the first commercial CPU–GPU heterogeneous architecture product. In the Sandy Bridge architecture, the CPU and GPU cores share the last-level (L3) cache and memory controllers connected by a 256-bit wide ring network. Note that GPU cores can execute graphics as well as data-parallel GPGPU applications. AMD also released a different heterogeneous design, the Fusion architecture [3], that integrates more powerful GPU cores. Although CPU and GPU cores share caches and memory controllers, all communications are performed through the north bridge, not the generic interconnection.

Although the current GPUs in on-chip heterogeneous architectures are not as powerful as today's high-performing GPUs, we project that more single-instruction multiple-data (SIMD) cores will be integrated in future generations. Therefore, we model our baseline heterogeneous architecture such that it has both high-performance CPU and GPU cores on-chip. Section 4.1 details the configuration of our baseline architecture.

2.2. Characteristics of CPU and GPU cores

This section describes the characteristics of CPU and GPU cores. Modern high-performance CPU cores are typically based on N-wide superscalar out-of-order cores. To reduce the penalty of the branch instructions, novel and often power-intensive branch prediction mechanisms are used. Large private caches (L1 and L2) are often employed to avoid long-latency access to off-chip memory. These cores are ideal for the serial execution.

On the other hand, GPUs pack more processing elements in each core. Each GPU core is an in-order SIMD processor. Multiple threads execute the same instruction with different data sets per core. When branch directions within a batch of threads are diverged, the execution of each branch path is serialized. Currently, no branch prediction mechanism exists. The core context switches to other batches of threads until the branch is resolved. Similarly, to hide memory latencies, GPU cores utilize massive multi-threading. When a thread is stalled due to the memory instruction, the execution is switched to other available threads.

2.3. Network-on-chip (NoC) router microarchitecture

In this section, we provide a brief background of the structure of an NoC router microarchitecture. A router has *N input and output ports* [13]. For example, a 2D-mesh has five (local, north, south, east, and west) ports. Each input port has *M input buffers* or *virtual channels (VC)*. New packets from the local network interface are inserted into the VCs, and packets from other routers are inserted into their respective VCs. When a new packet (or flit) is inserted, the *routing computation* unit decides the output port to the next router. The *virtual channel allocator (arbiter)* assigns a virtual channel on that output port. *The switch arbitration* unit controls the *crossbar* to move a flit to the assigned output port. Each flit traverses a link from the output port of one router to the input port of the next.

[1] In this paper, we interchangeably use the term on-chip interconnection network and the network on chip (NoC).

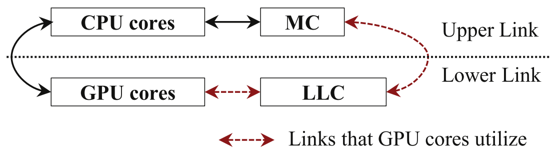


Fig. 1. Link utilization of GPU cores in the ring network of CPU–GPU heterogeneous architecture.

The flow of packets is pipelined in the NoC router with following stages: (1) Input buffering (IB): Flits received over a link or the source node are inserted into the virtual channel. (2) Route computation (RC): Using the information in the header flit, the output port is determined. (3) Virtual channel allocation (VCA): Using the output port information, a downstream virtual channel with available credits acquires a packet. (4) Switch allocation (SA): To traverse to the output port, a packet needs an exclusive grant to access the cross bar from its input virtual channel to the output port. (5) Switch traversal (ST): Once a switch is allocated to a packet, it can traverse to the output port over the crossbar. (6) Link traversal (LT): A flit is moved to the next router through the link.

In multiple pipeline stages of the router, packets from the same or different input virtual channels compete against each other for a grant to a virtual channel or a switch to the output port. Simple policies are used to arbitrate between these packets: (1) round-robin: the winning virtual channel is chosen in a sequential manner and (2) oldest-first: all packets in the router are searched and the oldest request is scheduled. More sophisticated proposals in the literature are described in Section 6.

3. Problems and design space exploration in NoCs of CPU–GPU heterogeneous architecture

This section describes the potential problems in designing the on-chip interconnection network in a CPU–GPU heterogeneous architecture.

3.1. Routing algorithm

NoC routers typically employ a simple static routing algorithm to minimize latency and complexity. For example, x–y or shortest-distance algorithms are widely used. However, this may result in link congestion in the heterogeneous architecture. For example, in Fig. 1, the GPGPU packets are not likely to use the upper link since the lower link offers the shortest distance from the GPU cores to the last-level cache (LLC) and memory controllers (MC). The lower link is also used between the L3 and the memory controllers. Therefore, only CPU packets use the upper link, which is possibly under-utilized. While studies on other routing algorithms show improved network performance, they are limited to traffic generated by specialized or CPU-only applications [31,21,22].

3.2. Resource contention and partitioning

CPU and GPU packets compete to acquire resources in various places, especially virtual and physical channels. When the resources are naively shared by both kinds of cores, higher-demanding cores will acquire the most resources, which are GPU cores. This is the same problem found in the LRU cache-replacement policy in the shared cache. To solve this problem, many researchers have proposed various static and dynamic cache partitioning mechanisms [41,39]. Similarly, partitioning mechanisms can be applied to on-chip virtual and physical channels. As explained in Section 2.3, each port has multiple virtual channels. We can partition these virtual channels to each application. Similarly, if multiple physical channels exist, we can dedicate some channels to CPU cores and the other channels to GPU cores. If the interference exhibited by other applications is significant, resource partitioning would prevent interference and improve performance. However, this can lead to resource under-utilization if partitioning is not balanced with demand. Therefore, partitioning should be carefully applied to on-chip network resources.

3.3. Arbitration policy

As described in Section 2.3, multiple arbiters exist in each router to coordinate packets from different ports. In a CPU–GPU heterogeneous architecture, due to the different network demands, arbitration between CPU and GPU packets is a non-trivial problem. At first glance, statically giving higher priority to CPU applications appears to be a reasonable solution since CPU applications are more latency sensitive. However, when CPU and GPGPU applications are both bandwidth-intensive, CPUs may be robbed of their fair share of the bandwidth. Therefore, the arbitration policy should also be carefully applied.

3.4. Homogeneous or heterogeneous link configuration

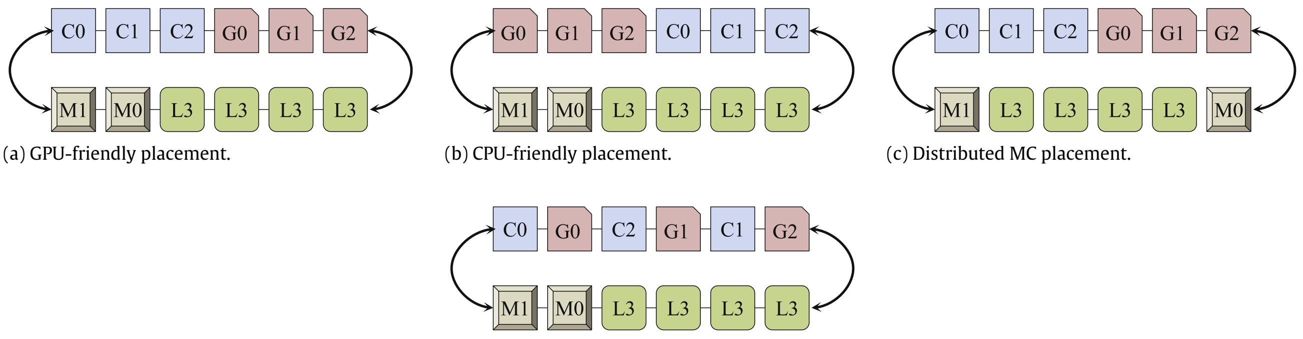
A homogeneous router configuration has the practical benefit of easier implementation. If all NoC routers are identical, each router module can be duplicated with little or no individual adjustment. Since the requirements of CPU and GPU cores are very different, routers may require higher bandwidth interconnection in terms of the link width or larger buffers to effectively handle traffic from both applications. However, this may result in under-utilization of resources in a certain core. For example, if a wider link width is used, GPGPU applications may directly benefit from more bandwidth capability, but CPU applications may not because they do not require such a high bandwidth. Therefore, the utilization of CPU links will be low. A heterogeneous link configuration may work better in this situation but requires more complex implementation and may not perform as well in some bandwidth-intense situations. However, a heterogeneous configuration will require more design and implementation efforts compared to the homogeneous network. We leave this discussion to future work since this is beyond the scope of our study.

3.5. Placement

As explained in Section 3.1, any placement of these components – CPU, GPU, L3, MC – results in unbalanced utilization of on-chip interconnection resources for some scenarios or under-utilization for all situations. Fig. 2 shows four possible examples of placement in the ring network. Among these examples, the placement of memory controllers (Fig. 2(d)) in many-core CMPs is studied by Abts et al. [1].

The first two examples are GPU- and CPU-friendly placements. Since all cache misses from a core need to reach L3 caches first, the distance between a core and a target L3 node may have a major impact on performance. Fig. 2(a) shows that the distance between the GPU cores and the L3 caches is shorter than the distance from the CPU cores. If there are more frequent accesses from the GPU cores to the L3 caches, this placement results in better system performance. For the same reason, Fig. 2(b) is more beneficial for CPU applications.

In another configuration, each memory controller is placed at the end of the die in Fig. 2(c). If we can map the disjoint address range of the physical memory for the two types of cores (by the operating system), we can balance the link usage and the latency between each core to the L3 cache and traffic to the memory controllers will be reduced. This setup could effectively divide the chip into two halves, which would be the most beneficial when



(a) GPU-friendly placement.

(b) CPU-friendly placement.

(c) Distributed MC placement.

(d) Interleaved placement.

Fig. 2. Placement examples in the ring network.

Table 1
Processor configuration.

CPU	1–4 cores, 3.5 GHz, 4-wide, out-or-order (OOO) gshare branch predictor 8-way, 32 KB L1 D/I cache, 2-cycle 8-way 256 KB L2 cache, 8-cycle
GPU	4 cores, 1.5 GHz, in-order, 2-way 16-wide SIMD 8-way, 32 KB L1 D (2c), 4-way 4KB L1 I (1c) 16 KB s/w managed cache
L3 Cache	4 tiles (each tile: 32-way, 2 MB), 64 B line
Memory controller	DDR3-1333, 2 MCs (each 8 banks, 2 channels) 41.6 GB/s BW, 2 KB row buffer, FR-FCFS scheduler

Table 2
NoC configuration.

Topology	Bi-directional ring network
Pipeline	5-stage (IB, RC, VCA, SA/ST, LT)
# VCs	4 per port (4-flit buffer)
# ports	3 (Local, Left, Right) per router
Link width	128 bits (16 B)
Link latency	2 cycles
Routing	Shortest distance
Flow control	credit-based, bubble routing [38]

each half requires the same amount of bandwidth, but would otherwise result in a major imbalance in resource distribution.

Fig. 2(d) shows an interleaved placement, where CPU and GPU cores are interleaved. The possible benefit of this design is that it can balance the traffic in each direction from each application. In other designs, the traffic from each type of application tends to head in the same direction due to the shortest-distance routing algorithm. When too much traffic is headed in one direction, the application will slow down.

Although some placements are not practical in an actual implementation, this is beyond the scope of our study. We leave this discussion to future work.

4. Evaluation methodology

4.1. Simulator

We use MacSim [20] for our simulations. We faithfully considered all CPU and GPU architectural components and performed detailed parameter space explorations and validations by comparing with executions on real processors. We did our best to correlate performance metrics against the measured data on real hardware. For all simulations, we repeat early terminated applications until all applications have finished at least once. This is to model the resource contention uniformly across the duration of simulation, which is similar to the work in [39,43,27,30].

Table 1 shows the processor configuration. We model our baseline CPU similarly to Intel's Sandy Bridge [25] with GPU cores similar to NVIDIA Fermi [35]. Table 2 shows the configuration of the NoC router. To avoid the deadlock configuration, we use bubble routing [38]. We set the GPU-friendly placement in Fig. 2(a) as the baseline configuration, which is a configuration similar to Intel's Sandy Bridge.

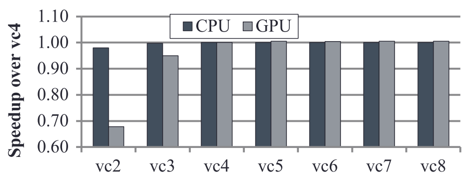
Table 3
CPU benchmark characteristics.

Benchmark	Suite	MPKI/Core	IPKC/Core
bzip2	Int	0.4	10.2
gcc	Int	1.2	10.9
mcf	Int	43.6	29.9
libquantum	Int	26.8	20.8
omnetpp	Int	10.2	21.5
astar	Int	7.6	17.8
bwaves	FP	22.3	61.6
milc	FP	31.1	49.4
zeusmp	FP	5.8	23.7
cactusADM	FP	6.2	31.6
leslie3d	FP	24.7	71.6
soplex	FP	13.9	34.4
GemsFDTD	FP	18.9	51.2
lbm	FP	18.0	71.1
wrf	FP	15.2	55.2
sphinx3	FP	0.6	31.2

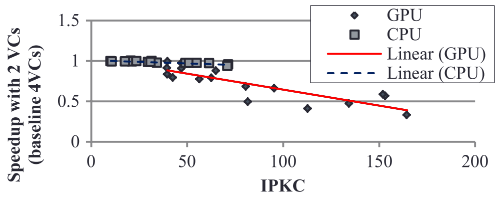
4.2. Benchmarks

We use SPEC 2006 CPU benchmarks and CUDA GPGPU benchmarks from publicly available suites, including Nvidia CUDA SDK, Rodinia [10], Parboil [23], and ERCBench [9]. For the CPU workloads, Pinpoint [37] was used to select a representative simulation region with the reference input set. Most GPGPU applications run until completion.

Among all available benchmarks, we only evaluate network-intensive CPU and GPU benchmarks based on IPKC ((Packet) Injection Per Kilo Cycles) metric. IPKC is very similar to MPKI (Miss Per Kilo Instruction) metric. These two metrics are strongly correlated because cache misses will introduce more traffic into the network. We classify benchmarks as network-intensive when IPKC is greater than 10 for CPU and 37.5 for GPU. Tables 3 and 4 show the characteristic of the evaluated network-intensive CPU and GPGPU benchmarks, respectively. We demonstrate that GPGPU benchmarks in general generate higher network traffic than CPU benchmarks. This is mainly due to the high number of concurrent threads running in a GPU core, which serves to hide memory latency by overlapping memory accesses. Subsequently, GPU cores generate a higher intensity of network traffic.



(a) Average speedup over vc4.



(b) X–Y chart of speedup and IPKC with linear trend lines.

Fig. 3. Evaluation of different # of virtual channels.

Table 4
GPGPU benchmark characteristics (MPKI and IPKC are the averages of each core).

Benchmark	Suite	MPKI/Core	IPKC/Core
BlackS	SDK	25.6	153.2
ConvS	SDK	0.0	39.6
Dct8x8	SDK	0.1	42.7
Histog	SDK	5.4	39.8
ImageD	SDK	0.1	62.7
MonteC	SDK	0.0	47.1
Reduct	SDK	123.5	164.5
SobolQ	SDK	22.7	152.1
Scalar	SDK	0.4	80.7
backPr	Rodinia	3.4	39.7
cfd	Rodinia	323.9	112.9
neares	Rodinia	0.1	81.7
bfs	Rodinia	10.6	95.4
needle	Rodinia	10.2	65.0
SHA1	ERCBench	4.5	47.1
fft	parboil	0.2	56.4
stencil	parboil	16.7	134.3

Table 5
Heterogeneous workload description.

Workload	# CPU	# GPU	# combinations
W-1CPU	1	1	13
W-2CPU	2	1	10
W-4CPU	4	1	10

We also generate workloads for the heterogeneous configuration experiments. We randomly choose combinations of network-intensive CPU and GPGPU applications. Table 5 describes the workload we evaluated.

4.3. Evaluation metric

We use the geometric mean (Eq. (1)) of the speedup of each application as the main evaluation metric, where n is the number of applications that are running concurrently in a workload. The speedup (Eq. (2)) of each application is defined as the IPC (Instruction per cycle) improvements over the baseline.[2]

$$speedup = geomean(speedup_{(0\ to\ n-1)}) \tag{1}$$

$$speedup_i = \frac{IPC_i}{IPC_i^{baseline}}. \tag{2}$$

We occasionally use the weighted speedup metric defined in Eq. (3), where n is the number of applications that are running concurrently in a workload. We specified the weighted speedup whenever it is used. Otherwise, the speedup metric in Eq. (1) is used.

$$weighted\_speedup = \sum_{i=0}^{n-1} \frac{IPC_i^{shared}}{IPC_i^{alone}}. \tag{3}$$

[2] The baseline uses the same configuration as in Table 1.

5. Results

In order to thoroughly cover possible problems and design space explorations in the on-chip network of CPU–GPU heterogeneous architectures described in Section 3, we present our findings for different combinations of workloads as following: Section 5.1 shows results of single-application workloads, where each application (CPU or GPGPU) runs in isolation. One-CPU/one-GPU workload (W-1CPU in Table 5) evaluations are presented in Section 5.2. Section 5.3 analyzes the result of W-2CPU and W-4CPU workloads. We conduct the scalability study of the ring network in Section 5.4 and we summarize findings and suggest an optimal ring network configuration in Section 5.5.

In each evaluation, we measure the impact of the number of virtual channels, link width, virtual/physical channel partition, link latency, arbitration policy, and network placement configuration. Note that we do not evaluate different routing algorithms since the ring network has only two possibilities of the route decision (left or right).

5.1. Single application analysis

We first evaluate each application in isolation (CPU or GPGPU application only) to analyze its characteristics without interference by other applications. Although previous studies exist [6,5] on the effect of the on-chip network for GPGPU applications, we again present the data to correlate with our other experiments. Each CPU application is tested by running it on one CPU core while all other cores remain idle. Similarly, GPGPU applications run all GPU cores while CPU cores remain idle.

5.1.1. Different number of virtual channels

Fig. 3(a) shows the result when we vary the number of virtual channels from two to eight. All results are normalized to the baseline configuration (4 VCs). As reported in [12], more number of virtual channels generally improves network bandwidth utilization and reduces network latency, which leads to better performance when considerable amount of the network loads exists. However, more virtual channels give only marginal benefits when an application does not show network intensity. This is not unexpected since even only one or a few virtual channels is not fully utilized. We can confirm this from the experiment. CPU applications suffered less performance loss than GPU applications with a small number of VCs. With two VCs, we observe a 2% performance degradation on average and the maximum performance loss is 6.6% for the lbm benchmark, which is the most network-intensive benchmark in Table 3. On the other hand, six GPGPU benchmarks show more than a 40% degradation.

When these applications are sharing the on-chip interconnection, we expect the inter-application interference to be a serious problem. We expect GPGPU applications to have a considerably greater impact on other applications when running concurrently on the network. However, from this experiment, guaranteeing a small number of VCs (one or two) to CPU applications is sufficient to maintain CPU application performance. Section 5.2.2 evaluates

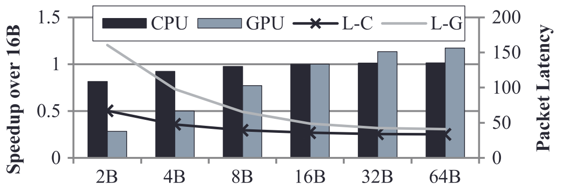


Fig. 4. Evaluation of different link widths (L-C: CPU network latency, L-G: GPU network latency).

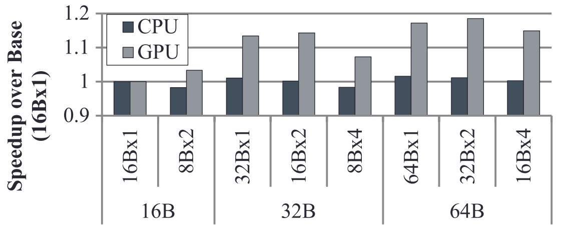


Fig. 5. Evaluation of different physical channels.

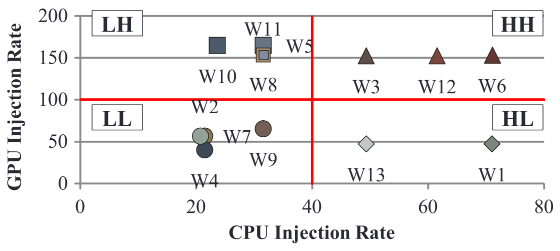


Fig. 6. Evaluation of different link latencies.

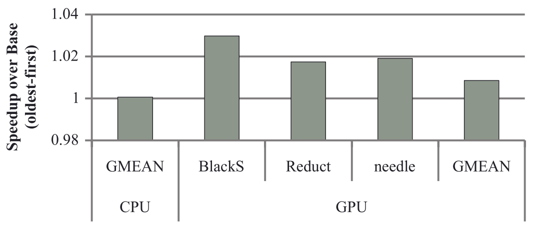


Fig. 7. Round-robin arbitration (baseline: oldest-first).

the effect of virtual channel partitioning in a multi-application environment.

To seek the correlation between the slowdown of having smaller VCs and IPKC, we show an *x*–*y* chart in Fig. 3(b). Although both applications show linear regression lines, CPU applications do not show much variance. Applications with higher IPKC show greater degradation in general.

5.1.2. *Different link width*

Fig. 4 shows the impact of link widths for each type of application compared to the baseline of the 16B link width. Compared to the impact of varying the number of virtual channels, link width has a greater effect on the performance of both applications. GPGPU benchmarks generally show more sensitivity to link widths than CPU benchmarks, especially in Reduct, Scalar, cfd, and bfs benchmarks. This is potentially due to the large number of memory requests made in a batch by a huge number of concurrently running threads. GPGPU applications can take advantage of the wider links and prevent network congestion. On the other hand, CPU benchmarks do not show significant improvement with wider links (32B: 1.1%, 64B: 1.6% on average) compared to the baseline. However, lower network bandwidth in GPGPU applications induces a significant latency increase, thereby hurting performance excessively. The L-G line in Fig. 4 shows that the average network latency of GPGPU applications is increased by 3.3 times, while CPU applications (L-C line) show a relatively small increase (1.87 times) with the 2B link. As a result, GPGPU applications show a 72% slowdown over the 16B link, but CPU applications only show a 19% slowdown.

From this observation, we conclude that having a wider link is very helpful to GPGPU applications, but not to CPU applications. This confirms that GPGPU benchmarks are more bandwidth-limited, while CPU benchmarks are not. As a result, this observation leads us to study heterogeneous link configurations in Section 5.2.6.

5.1.3. *Different channel configuration*

In this section, we evaluate the impact of different widths and number of physical channels. The purpose of this evaluation is based on the intuition that a wider link can be beneficial to reduce the latency in the cases of larger packet requests, but multiple channels can be better utilized for more general traffic when smaller packets become a contributing portion of traffic. Fig. 5 shows the results.

For CPU applications, wider links improve performance by only a small percentage. However, having smaller, multiple links degrades performance. On the other hand, GPGPU applications show improvement with two physical channels (16Bx2 and 32Bx2). However, having even more channels increases the packet latency significantly (2× longer latency for data packets) while not fully utilizing all channels. As a result, the benefit of wider links decreases. This finding indicates that the width and number of physical channels should be well balanced to support various applications that have different latency/bandwidth requirements.

5.1.4. *Different link latency*

In order to see the effect of different latencies of the link, we perform experiments with different link latencies. Fig. 6 shows the result with two-cycle as the baseline latency. We double the link latency from one to 64 cycles.

Although both types of applications are sensitive to link latency, we observe that the degree of sensitivity is much higher in CPU benchmarks. Even with small changes (from two to eight cycles), most CPU benchmarks suffer from the increase in latency. We observe 7%, 19%, and 36% degradations on average for four-cycle, eight-cycle, and 16-cycle configurations, respectively. However, the performance of many GPGPU benchmarks did not degrade significantly until the 16-cycle latency configuration. On average the performance degradations are 5% and 18% for GPGPU benchmarks with eight-cycle and 16-cycle configurations, respectively. As we explained in Section 2.1, CPU benchmarks are known to be latency-sensitive and GPGPU benchmarks are bandwidth-limited. This simulation result confirms this tendency.

5.1.5. *Different arbitration policy*

In this section, we evaluate two simple arbitration policies: round-robin arbitration and oldest-first policy while fixing other configurations the same.[3] Fig. 7 shows the results.

Not surprisingly, neither type of application shows significant changes. Since there are three input ports with few virtual channels in the ring network, not enough bottleneck is introduced by network complexity for an arbitration to resolve. CPU applications never show more than a 0.5% variance. Only a few GPGPU benchmarks, BlackS, Reduct, and needle, show more than a 1.5% variance.

[3] We evaluate other static arbitration policies with heterogeneous workloads in Sections 5.2.3 and 5.3.4.

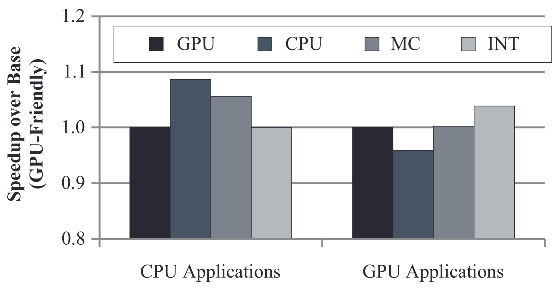


Fig. 8. Different placement results (MC: distributed memory controller, INT: interleaved).

5.1.6. *The effect of network placement configuration*

Fig. 8 shows the effect of different placement policies. We evaluate four different placements, as shown in Fig. 2. For each type of application, placing the cores closer to the L3 caches improves performance by reducing the round trip latency between the cores and the memory system (the L3 caches and the memory controllers). The CPU-friendly placement improves the performance of CPU applications by more than 8% on average. On the other hand, the CPU-friendly placement degrades the performance of GPGPU applications by 4.1% compared to the GPU-friendly placement.

The MC placement improves the performance of CPU applications by 5.6%. The performance gain is partially from reducing the through-traffic of the memory controllers. In other configurations, the shortest distance from Core 0 to the L3 caches is through the memory controllers. Therefore, even though the destination of a packet is not a memory controller but an L3 cache node, all packets from and to the core must travel through these memory controllers. As a result, this path is always busy and tends to congest traffic. However, by placing the memory controllers on both sides of the cache nodes, through traffic is reduced.

On the other hand, there are cases in GPGPU applications where the MC placement either hurts or improves performance (from −7% to 6%). For negative cases, this placement introduces an extra node in the critical path between GPU cores and L3 cache nodes (M0 in Fig. 2(c)), resulting in extra latency and higher congestion (near M0) as traffic injected by the memory controller is also in the critical path. Hence, in some applications these negative effects offset the benefits of reducing congestion to/from the memory controllers explained earlier. Overall, for GPGPU applications, the benefit of the MC placement is washed out.

For the INT placement, note that CPU-friendly and INT placements are identical for CPU applications since we run only one application on Core 0 (C0), and this placement configuration does not impact the latency from this core to memory nodes. For GPGPU applications, we observe improvements of 3.9% on average due to an increase in path diversity, as the cores will take both sides of the ring instead of favoring only one side due to the shortest-distance routing algorithm described in Section 3.1.

5.2. *Multiple-application experiments (W-1CPU workload)*

In this section, we look at multi-application experiments to analyze the impact of inter-application interference. Each test randomly chooses one network-intensive CPU and GPGPU application to run them concurrently (the W-1CPU workloads in Table 5). Fig. 9 shows an x–y chart of all W-1CPU workloads in terms of the IPKC characteristic. We split this into four regions (CPU Injection/GPU Injection, HH—High CPU and High GPU, HL, LH, and LL) based on the CPU and GPU injection rates.

The following factors are measured for their impact: router buffer partitioning, arbitration policy, network placement, physical channel partitioning, and heterogeneous link configuration.

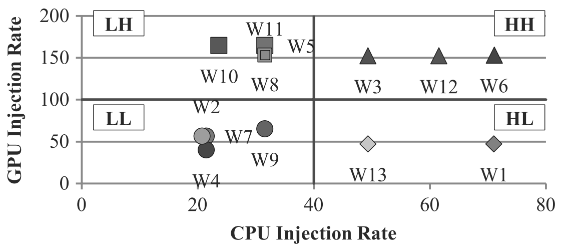


Fig. 9. W-1CPU workload characterization (IPKC: injection per kilo cycles).

5.2.1. *Interference with GPGPU applications*

We first explain how applications interfere with each other. Fig. 10(a) shows an x–y chart of the slowdown of each application compared to when they are running alone and Fig. 10(b) shows the weighted speedup[4] of each workload (sorted in ascending order).

We can observe that significant interference is caused by GPGPU applications. In Fig. 10(a), most GPGPU applications do not show more than a 5% slowdown, while only three CPU applications show less than a 20% slowdown. In Fig. 10(b), GPGPU applications in poorly performing combinations (from the leftmost, W3, W12, W10, W5, and W6) all belong to the HH or LH region, while the best performing ones (from the rightmost, W1 and W13) belong to the HL region in Fig. 9. We expect that although GPGPU applications will experience more interference with CPU applications as the number of concurrently running CPU applications increases, GPGPU applications are more likely to interfere with CPU applications.

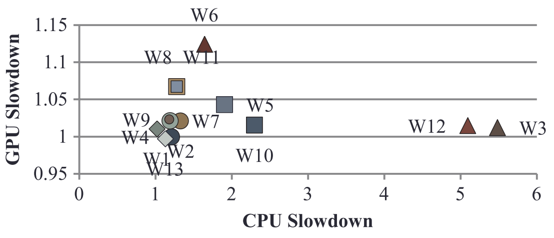
5.2.2. *Router buffer partitioning*

As Tables 3 and 4 show, GPGPU benchmarks exhibit more frequent network injections. If we do not partition the router buffer space, the GPGPU packets will occupy more space and unnecessarily degrade the performance of CPU applications. This is a similar problem to the traditional LRU replacement policy in CMPs. Since the naive LRU policy does not partition cache space, higher-demanding applications will occupy more cache space. To solve this problem, many researchers have proposed static and dynamic cache partitioning mechanisms [41,39]. Similarly, NoC virtual channels can be statically or dynamically partitioned; in other words, a few virtual channels can be dedicated to CPU packets and other channels to GPU packets. In this section, we evaluate the effect of virtual channel partitioning.

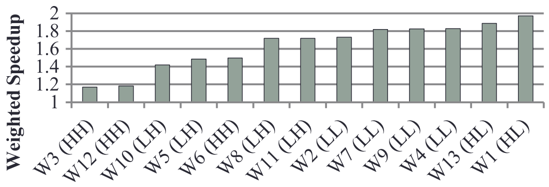
We compare the baseline unpartitioned virtual channels with various static partitioning configurations, as shown in Fig. 11. We observe that increasing the number of virtual channels is not helpful. GPGPU packets occupy most of the available buffer space. Increasing to six and eight virtual channels results in minor improvements in GPGPU applications, while the performance of CPU applications stays the same. Compared to the unmanaged baseline, allocating at least some fixed number of virtual channels to CPU applications significantly improves performance by more than 35%, while GPGPU applications show 8% and 1% degradations with three and four dedicated virtual channels, respectively.

For a deeper analysis, we show an x–y chart of CPU and GPU speedups in the 2:2 (VC4) virtual channel partitioning configuration for all W-1CPU workloads in Fig. 12. We pick this configuration because both CPU and GPGPU applications with this configuration show representative behavior from virtual channel partitioning. We observe that three groups exist in Fig. 12: group 1 (top left region, W1, W4, W9, W13: moderate CPU and GPU

[4] Eq. (3). Higher is better. The ideal weighted speedup is 2 if no inter-application interference is exhibited.



(a) Slowdown *X*–*Y* chart (△: HH, ⋄: HL, □: LH, ∘:LL).



(b) Weighted speedup (Workload and Group Id in Fig. 9).

Fig. 10. Interference with GPGPU applications.

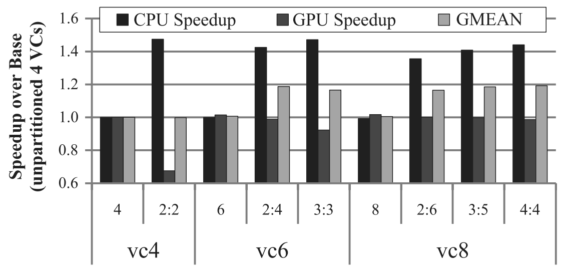


Fig. 11. Router buffer partitioning results (ex. 6 is 6 unpartitioned VCs; 2:4 is 2 CPU VCs, 4 GPU VCs).

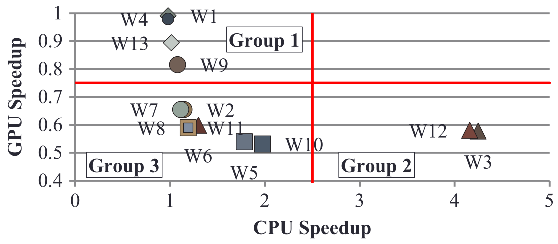


Fig. 12. VC partitioning *x*–*y* chart (baseline: shared VC, △: HH, ⋄: HL, □: LH, ∘:LL in Fig. 9).

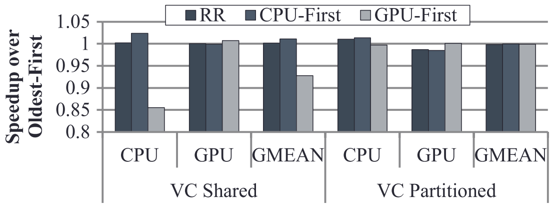


Fig. 13. Different arbitration policy results (CPU and GPU indicate the speedup of each application).

speedup), group 2 (bottom right region, W3, W12: excessive CPU and GPU), and group 3 (others: moderate CPU and excessive GPU). In group 1, GPGPU applications have low injection rate (HL or LL group in Fig. 9), so the interference by GPGPU applications is limited as well. Therefore, CPU and GPGPU applications effectively share VCs. As a result, both CPU and GPGPU applications do not show significant variances with VC partitioning. CPU applications in group 2 have higher injection rate (greater than 50 IPKC), so the performance of CPU applications is significantly improved with VC partitioning. Almost all CPU applications in group 3 have low injection rate (LH or LL group), so VC partitioning hurts the performance of GPGPU applications. In other configurations, we can observe that both CPU and GPGPU applications show similar trends, but we do not see as much degradation as in the 2:2 (VC4) configuration for GPGPU applications.

In addition, we examine the virtual channel occupancy of CPU and GPU types for each group with 4 VCs. With the 2:2 configuration, in group 1, the occupancy of CPU and GPU VCs are much less than 100%. Since group 1 mostly consists of low-intensive (L type) CPU and GPGPU applications, the number of in-flight packets is not many. In groups 2 and 3, the occupancy of GPU VC is always close to 100% while the occupancy of CPU VCs are near 100% only with workloads that consist of H-type CPU applications.

Across different VC partitioning configurations (1:3, 2:2, and 3:1), although the utilization of GPU VCs is generally very high, the utilization of CPU VCs is decreasing with more number of VCs. This result well correlates with the experiment performed in Section 5.1.1 (different number of VCs).

From these observations, we see that when CPU and GPGPU applications run concurrently in the shared on-chip network, guaranteeing the minimum buffer space to CPU applications would improve the overall system performance. However, providing additional buffer space does not result in additional performance increases.

5.2.3. *Arbitration policy*

In heterogeneous architectures, CPU and GPU packets compete for buffer space and switch arbitration. To see the effect of arbitration policies, we try two static policies: CPU-first and GPU-first policies. To prevent starvation, we implement a form of batching similar to that in [34,15]. We compare these two static policies and the round-robin policy with the oldest-first arbitration. Please note that the complexity of arbiters in case of shared and partitioned virtual channels might be different. In all evaluated arbitration policies, we need to select a packet in a random virtual channel position. When more virtual channels exist per arbiter (shared VC case), the hardware selection logic should be more complex. This may require more time for an operation and lead to performance degradations. On the other hand, in order to identify different types of packets and virtual channel types (partitioned VC case), we need to keep track of additional information.

Fig. 13 shows the results using the shared and partitioned virtual channel configurations. First, as seen in Section 5.1.5, there is no significant difference between oldest-first and round-robin policies, regardless of virtual channel configuration. However, for the two static policies, the VC configuration affects the result significantly. Since VC partitioning guarantees the minimal service for each type, the effect of different policies decreases. As a result, the three policies behave similarly (less than 1% delta) with VC partitioning.

However, with the shared VCs, the CPU-first policy slightly improves the performance of CPU applications without degrading that of GPGPU applications. On the other hand, the GPU-first policy degrades the performance of CPU applications by more than 15% on average while improving GPGPU performance by only 0.5%. Looking at the geometric mean, CPU-first improves performance by 1.5%, but GPU-first degrades it by 7.3% on average. This experiment again confirms that CPU packets are latency-sensitive, so we need to prioritize CPU packets.

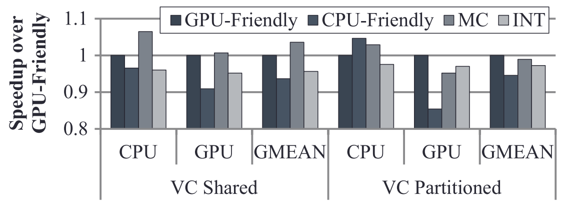


Fig. 14. Different placement results for heterogeneous workloads (MC: distributed memory controller, INT: interleaved).

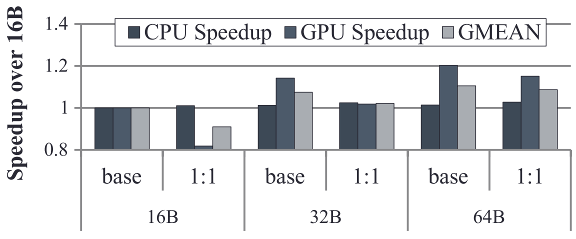


Fig. 15. Physical channel partitioning results (16B-base: unpartitioned 1 channel, 16B-1:1: 2×8B channels and one channel is dedicated for CPU and the other is for GPGPU application).

5.2.4. Network placement configuration

We evaluate different placement policies on heterogeneous workloads, as shown in Fig. 14. We again perform experiments with different VC configurations. With the shared VC, even CPU-friendly placement degrades CPU applications. This is because the lengthened distance from the GPU cores to both L3 caches and memory controllers increases system-level traffic congestion. As a result, each CPU packet is penalized by this congestion. The MC placement slightly improves the performance of both applications (3.5% on average), while the INT placement degrades both (−4.4% on average).

However, with the partitioned VC, we observe the expected behavior. The CPU-friendly placement shows better performance for the CPU applications (4.6%), but it degrades performance of the GPGPU applications even more (−14.6%). This degrades overall performance. The MC placement improves CPU application slightly by partially reducing the distance to the memory controllers, but it worsens GPGPU applications. The INT placement degrades the performance of both applications.

From this experiment, we observe that GPGPU applications have more impact on the network. The overall performance gain can be acquired by not penalizing GPGPU applications.

5.2.5. Physical channel partitioning

In this section, we evaluate physical channel partitioning. Similar to router buffer partitioning, if multiple physical channels exist in the router, we can partition channels to each type of application.

Fig. 15 shows the results. Similar to VC partitioning, the CPU applications benefit from a dedicated channel. However, a significant performance loss in GPGPU applications results. This indicates that GPGPU applications require a wider physical channel than CPU applications. When the channel is partitioned, we observe that the utilization of the GPU channel is slightly increased while that of the CPU shows very low utilization (around 5%). Therefore, to obtain better channel utilization for GPGPU applications, a wider channel instead of multiple channels is more effective.

5.2.6. Heterogeneous link configuration

In this section, we evaluate heterogeneous link configurations. We first categorize routers into three groups: CPU-router (4 routers), GPU-router (4 routers), and memory-router (6 routers: 4 L3 cache and 2 memory controller routers). Our baseline uses

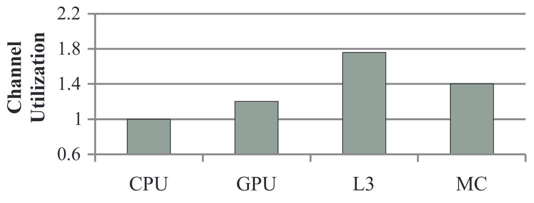


Fig. 16. Physical channel utilization (relative to CPU routers).

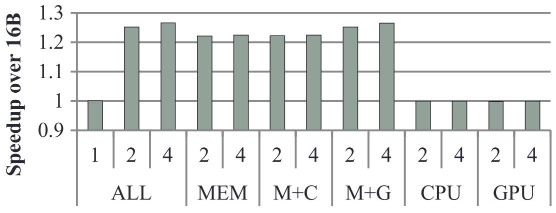


Fig. 17. Heterogeneous link configuration results (ALL: all routers, MEM: vary memory router link only, and M+C: memory and CPU routers. 1, 2, or 4 in the x-axis indicates the number of physical channels).

one 128-bit (16B) link between each node. We vary the number of physical channels in each group. First, we show the physical channel utilization of each router normalized to the CPU routers in Fig. 16. L3 and MC routers obviously have much more traffic, thereby utilizing channels more compared to processor routers. Even if the number of concurrently running CPU applications increases, we expect that a similar trend will be observed.

Fig. 17 shows the results of various link configurations. We first evaluate the homogeneous link configuration (ALL). Having more physical channels is always beneficial, but there is a diminishing return in performance after two channels. We observe performance improvements of 25% and 27% on average with two and four physical channels, respectively. Then, we evaluate heterogeneous configurations by varying each type of router (MEM, CPU, GPU, M+C, and M+G). While increasing the number of channels for CPU or GPU routers does not help improve performance, increasing memory-router channels has a direct effect on performance (22.1% and 22.4% with two and four physical channels, respectively). As in Fig. 16, memory routers are mostly busy during the entire execution, but CPU and GPU routers are not. Because there are five different data flows in the ring network – (1) CPU to L3, (2) GPU to L3, (3) L3 to Memory controller (MC), (4) MC to L3, and (5) L3 to CPU (or GPU) – most traffic goes through the memory routers. By allocating wider links to only the memory routers rather than all routers, we can fully utilize these links without powering underutilized links for the CPU and GPU routers. In addition, giving GPU routers additional channels, on top of wider memory router links, shows an additional boost in performance. Since GPUs are major sources of network traffic, without more channels between them, GPU-routers will become the new bottleneck.

5.3. Multiple CPU application experiments (W-2CPU and W-4CPU workloads)

In this section, we evaluate multiple CPU applications and one GPGPU application running simultaneously. We repeat the same set of experiments as in Section 5.2.

5.3.1. Router buffer partitioning

Fig. 18 shows the results of VC partitioning in the W-4CPU workloads (4 CPUs + 1 GPGPU). Note that the W-2CPU (2 CPUs + 1 GPGPU) workloads are omitted, as they show roughly identical results as W-4CPU results. W-2CPU and W-4CPU workload data show very similar trends as W-1CPU (1 CPU + 1 GPGPU) experiments. Dedicated (at least a few) virtual channels to CPU applications are

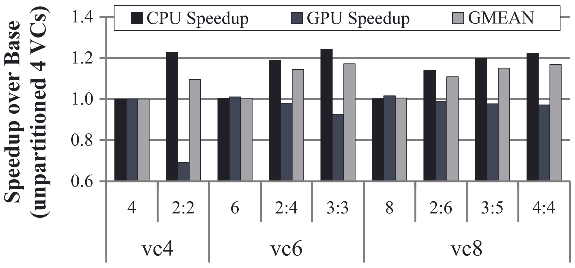


Fig. 18. Router buffer partitioning results (4 CPUs + 1 GPGPU workloads).

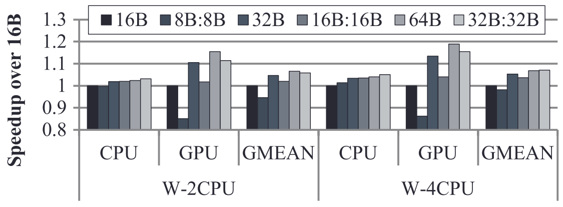


Fig. 19. Physical channel partitioning results (16B-base: unpartitioned 1 channel, 16B-1:1: 2×8B channels and one channel is dedicated for CPU and the other is for GPGPU application).

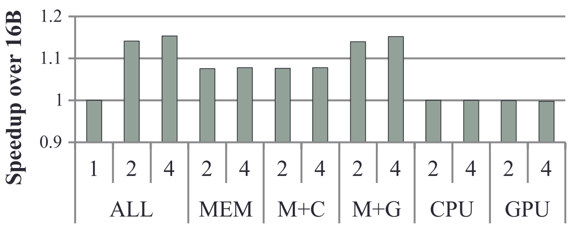


Fig. 20. Heterogeneous link configuration results (ALL: all routers, MEM: vary memory router link only, and M+C: memory and CPU routers. 1, 2, or 4 in the *x*-axis indicates the number of physical channels).

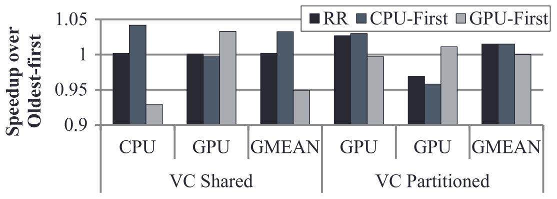


Fig. 21. Different arbitration policy results.

shown to be helpful, but the performance of GPGPU applications degrades with less than three VCs. This again indicates significant traffic injected by GPGPU applications and interference by GPGPU applications. VC partitioning will reduce this interference.

From the various VC-related experiments, our conclusions of the ideal VC configuration are that (1) VCs should be partitioned, especially for CPU applications; (2) However two to three VCs are sufficient; and (3) a GPGPU application requires at least three VCs, but having more does not help. Therefore, the ideal VC configuration will be five virtual channels: two are dedicated for CPU and the other three are for GPGPU applications.

5.3.2. Physical channel partitioning

Fig. 19 shows the results for both W-2CPU and W-4CPU workloads. As the number of concurrently running CPU applications increases, the benefit of having a separate physical channel for CPU applications decreases since the channel itself is shared by more applications. However, GPGPU applications still suffer from narrower links, as they depend on bandwidth. As a result, we always observe a system performance degradation with a partitioned physical channel configuration.

5.3.3. Heterogeneous link configuration

Fig. 20 shows the results for the W-4CPU workloads. Again, the W-2CPU workloads are omitted due to their similarity to the W-4CPU workloads. Compared to the W-1CPU workloads, there is no significant difference. Multiple physical channels for the memory and the GPU routers proved to be beneficial. However, increasing the number of physical channels for only CPU or GPU routers is not beneficial because half of the traffic is cache-miss requests. The size of request traffic is small (1 flit) and the request traffic would not occupy multiple physical channels. Therefore, having multiple physical channels does not improve performance. However, for GPU routers, the additional channels for the memory routers adds additional improvements.

5.3.4. Arbitration policy

Fig. 21 shows the results from different arbitration policies on both shared (CPU and GPU packets share all VCs) and partitioned VCs (2 VCs are dedicated for each type) for the W-4CPU workloads. Round-robin is comparable to the Oldest-First policy. Although the CPU-First policy consistently shows better performance, the benefit decreases with the partitioned VC. On average, CPU-First shows 3.2% and 1.5% improvements with the shared and partitioned VC, respectively. On the other hand, GPU-First degrades CPU applications in the shared VC configuration.

Throughout multiple-workload evaluations (W-1CPU, W-2CPU, and W-4CPU), the effect of different arbitration policies is not so important, especially if virtual channels are partitioned to each type of application. This is not entirely unexpected since there is a smaller number of input and output ports in the ring network. We expect that the role of intelligent arbitration becomes significant with 2D topologies.

5.3.5. Placement

Fig. 22 shows different placement evaluations for W-2CPU and W-4CPU workloads with different virtual channel configurations (shared and partitioned). With the shared VC, the MC placement shows the best results. This is mainly because traffic from/to the memory controllers is distributed to both sides of the chip, thereby reducing the congestion in those routers. With the partitioned VC, the effect of different placement policies is a bit reduced. GPU-friendly and MC placements perform the best.

5.4. Scalability

The last evaluation of our study is the scalability of the ring network. The ring network is known to handle a small number of nodes. In this section, we stress the ring network to show how many cores can be used. We run GPGPU applications and scale the number of cores from four to 20.

Fig. 23 shows four different types of applications: linear-scale, log-scale, saturated-scale (performance saturated after *N* cores), and unscalable. Linear and log-scale applications are less network intensive and show performance improvement because increasing the number of cores does not saturate the interconnection network.

The other two types show that the performance flat lines at some point due to the congestion in the network. Especially, the performance of unscalable benchmarks degrades for even a small number of cores (Fig. 23(d)). These are the most network-intensive benchmarks in Table 4 (BlackS, Reduct, SobolQ, cfd, bfs).

Fig. 24 shows the correlation between the scalability of a ring network and the IPKC. As we can expect, in general, higher IPKC applications (especially above 100 IPKC) do not show good scalability.

In sum, the ring network is not scalable when it comes to significantly memory/network-intensive applications. However, we observe that the ring network is still a good candidate to handle moderate memory/network-intensive applications with a moderate number of cores until the on-chip interconnection bandwidth can handle them.

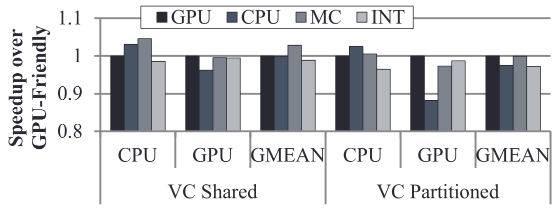
5.5. Summary of findings

In this section, we summarize the findings on the ring network for CPU–GPU heterogeneous architecture from our evaluations in previous sections.

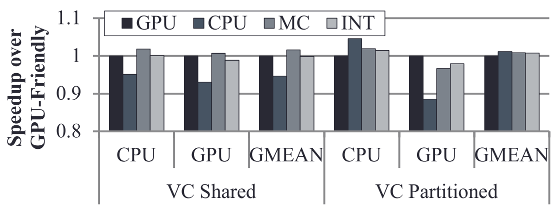
1. As is widely known, CPU benchmarks are latency-sensitive and GPGPU benchmarks are bandwidth-limited. From the empirical data, we confirm that this applies to the on-chip interconnection under various circumstances.
2. When CPU and GPGPU applications share the same NoC, a significant interference by GPGPU applications exists. To prevent this interference, we evaluate two resource partitioning schemes: virtual and physical channels. Having enough dedicated virtual channels for each type improves performance. On the other hand, multiple narrower physical channels rather than a single wider channel significantly degrades the performance of GPGPU applications. However, they do not improve CPU application performance due to lower link utilization.
3. A heterogeneous link configuration shows a promising result. Since the traffic is the highest in the last-level cache and the memory controllers (to/from CPU and GPU cores), those routers become traffic hotspots. By adding more physical channels, we can achieve similar performance compared to having multiple channels for all routers.
4. Prioritizing CPU request packets yields the best performance among other policies, but the benefit is very small. Generally, router arbitration has minimal effect on performance in a ring network.
5. GPU cores should be located close to the L3 caches and the memory controllers to avoid congestion, which eventually affects other applications as well. Moreover, separating memory controllers to both sides of the chip would reduce the traffic and improve performance in some cases.
6. The ring network is not scalable and even saturates with only a small number of cores (4–6) for some benchmarks. However, we observe that the ring network is still a good medium to handle a moderate number of cores until on-chip interconnection bandwidth can handle them.

From our findings, we suggest an optimal router option that is a combination of the best performance with minimal hardware resources for each individual experiment in Table 6. Please note that the suggested configuration may not be applicable to other network configurations and is specific for the ring on-chip network with our configurations. Also, some of suggested options may require more hardware resources (one more virtual channel per port in a router and more physical channels for memory and GPU routers) than the baseline. The detailed tradeoff between more hardware resources and performance/power improvements remains in future work.

Fig. 25 shows the evaluation results. We incrementally add one type of optimization on top of other optimizations. As can be seen in the figure, we can observe that greater improvements mostly come from virtual channel partitioning and heterogeneous link configurations, while other optimizations are less critical. This result is expected and well correlates with previous observations. Overall, our suggested router configuration improves performance by 22%, 19%, and 26% for W-1CPU, W-2CPU, and W-4CPU workloads, respectively.

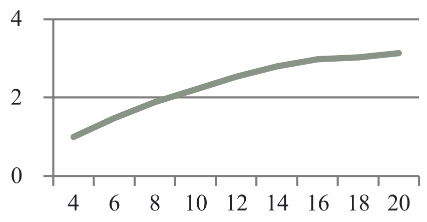


(a) W-2CPU workloads (2 CPUs + 1 GPGPU).

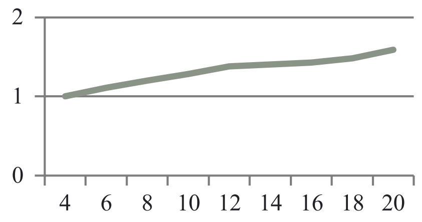


(b) W-4CPU workloads (4 CPUs + 1 GPGPU).

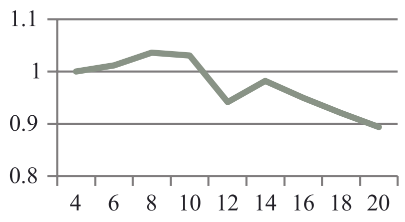
Fig. 22. Placement (MC: distributed memory controller, INT: interleaved).



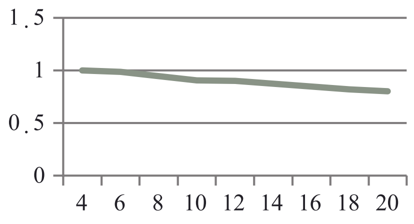
(a) Linear-scale.



(b) Log-scale.



(c) Saturated-scale.



(d) Unscalable.

Fig. 23. Scalability test (*x*-axis: # cores, *y*-axis: speedup over 4-core).

Table 6
Putting it all together—baseline and a suggested ring configurations.

	Base	Suggested option
Virtual Channel	shared 4 VCs	partitioned 5 VCs (2: CPU, 3: GPU)
Physical Channel	1 × 16 B	1 × 16 B for CPU 2 × 16 B for LLC, MC, and GPU
Arbitration	Oldest-first	CPU-first
Placement	GPU-Friendly	MC or GPU-Friendly

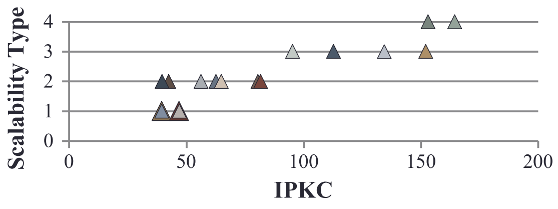


Fig. 24. Scalability (*y*-axis: scalability type, 1: linear, 2: log, 3: saturated, 4: unscalable).

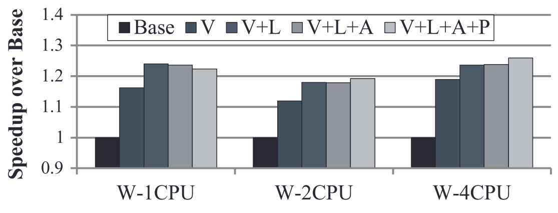


Fig. 25. Suggested on-chip interconnection configuration results (V: VC partitioning, L: Heterogeneous link configuration, A: Arbitration, P: Placement).

6. Related work

6.1. On-chip ring network

Although the ring network has been extensively studied in the past, we limit our discussion to the on-chip ring network in this section. Bononi and Concer [7] studied and compared various on-chip network topologies, including the ring, in the SoC (System on Chip) domain. Bourduas and Zilic [8] proposed a hybrid ring/mesh on-chip network. A conventional 2D-mesh network has a large communication radius. To reduce the communication cost, the network is partitioned into several sub meshes and the ring connects these partitions. Ainsworth and Pinkston [2] performed a case study of the Cell Broadband Engine's Element Interconnect Bus (EIB), which consists of four ring networks for data and a shared command bus that connects 12 elements.

6.2. CPU–GPU heterogeneous architecture research

Since the CPU–GPU heterogeneous architecture was introduced recently, not many studies are available in the literature. In particular, the resource-sharing problem is not well discussed. Lee and Kim [30] recently studied the cache-sharing behaviors in a CPU–GPU heterogeneous architecture and proposed TLP-aware cache management schemes, which sample GPU cores with different cache policies to see the performance effects by caches.

Yang et al. [44] proposed a pre-execution mechanism of GPGPU applications on CPU cores. By automatically extracting memory operations of the GPGPU kernel and dispatching these operations on the CPU when the kernel is launched, data blocks of GPGPU kernels are brought in the shared cache by CPU cores. As a result, most off-chip accesses from GPGPU applications are hit in the cache.

Jeong et al. [28] considered quality-of-service (QoS) in a multiprocessor system-on-chip when off-chip bandwidth is shared between CPU and real-time constrained graphics applications. Depending on the progress made by graphics applications, the priority of CPU and GPU requests is dynamically adjusted. Ausavarungnirun et al. [4] proposed the staged memory scheduler (SMS). Due to massive memory accesses by GPU cores, the visibility of the memory requests by the memory scheduler is very limited. SMS tackles this problem with a multiple-stage memory scheduler.

In addition, some work on utilizing idle cores to boost performance has been done. Woo and Lee proposed Compass [42], which utilizes idle GPU cores for various prefetching algorithms in heterogeneous architectures.

6.3. Heterogeneous NoC configuration

Much research has been done on heterogeneous NoCs involving many-core CPUs. Heterogeneous network configurations (HeteroNoC), by Mishra et al. [32], proposed asymmetric resource allocations (buffers and link bandwidth) to exploit non-uniform demand on a mesh topology. HeteroNoC showed that routers along the diagonals provided performance improvement over homogeneous resource distribution. Grot et al. [18] proposed Kilo-NOC, with shared resources isolated into QoS-enabled regions to minimize the network complexity. Kilo-NOC also reduces area and energy, in non-QoS regions by using a MECS—(Multidrop Express Channels) [19] based network with elastic buffer and novel virtual channel allocation that reduces VC buffer requirements by eight times over MECS with minimal latency impact.

6.4. NoC prioritization

Application-aware prioritization [14] computes the network demand of applications at intervals by looking at a number of metrics such as private cache misses per instruction, average outstanding L1 misses in MSHRs, and average stall cycles per packet. This produces a ranking of an application, and all packets of one application are prioritized over another, resulting in a coarse granularity of control. To prevent application starvation, a batching framework is implemented that prioritizes all packets of one time quantum over another, regardless of source application. Aérgia [15] predicts the available latency (slack) of any packet by the number of outstanding L1 misses and prioritizes low-slack (critical) packets over packets with higher slack when they are within the same batching interval.

6.5. NoC routing

Ma et al. [31] proposed destination-based adaptive routing (DBAR), a network with a novel congestion information network with a low wiring overhead. Like RCA [17], DBAR uses the virtual channel buffer status of nodes on the same dimension to route around congested paths. In addition, DBAR ignores nodes outside the potential path to eliminate interference and provides dynamic isolation from outside regions of the network.

Bufferless routing [33] showed substantial energy savings from removing input buffers by deflecting incoming packets from congested output ports. This routing algorithm managed performance similar to other buffered routing algorithms but only at low traffic.

7. Conclusion and future work

In this paper, we explore a broad design space in the on-chip ring interconnection network for a CPU–GPU heterogeneous architecture. We observe that this type of heterogeneous architecture has been adopted by major players in the industry and will become the mainstream processor type in subsequent generations. We observe that the interference exhibited by other applications, mostly by GPGPU applications, is significant and can be detrimental to system performance if not properly managed. We examine resource partitioning schemes for virtual and physical channels. Virtual channel partitioning improves performance, but physical channel partitioning degrades it because of link under-utilization. Heterogeneous link configurations, different arbitration policies, and placement configurations have been considered in this paper as well. The heterogeneous link configuration shows effectiveness, but other configurations have less benefit. From numerous experimental results, we suggest an optimal router configuration that combines the best of individual experiments for this architecture, which improves performance by 22%, 19%, and 16% for W-1CPU, W-2CPU, and W-4CPU workloads, respectively.

In future work, we will evaluate other topologies, including two-dimensional mesh and torus. Also, we will study various resource partitioning mechanisms for the on-chip network.

Acknowledgments

We gratefully acknowledge the support of the National Science Foundation (NSF) CAREER CCF 1054830, CNS 0855110, and Sandia National Laboratories. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or Sandia Lab.

References

[1] D. Abts, N.D.E. Jerger, J. Kim, D. Gibson, M.H. Lipasti, Achieving predictable performance through better memory controller placement in many-core cmps, in: S.W. Keckler, L.A. Barroso (Eds.), Proc. of the 31st Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, 2009, pp. 451–461.
[2] T.W. Ainsworth, T.M. Pinkston, On characterizing performance of the cell broadband engine element interconnect bus, in: Proc. of the 1st ACM/IEEE Int'l Symp. on Network-on-Chip, NOCS, IEEE Computer Society, Washington, DC, USA, 2007, pp. 18–29.
[3] AMD, Fusion. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx, 2011.
[4] R. Ausavarungnirun, G. Loh, K. Chang, L. Subramanian, O. Mutlu, Staged memory scheduling: achieving high performance and scalability in heterogeneous systems, in: Proc. of the 34th Annual Int'l. Symp. on Computer Architecture, ISCA, IEEE Press, Piscataway, NJ, USA, 2012, pp. 416–427.
[5] A. Bakhoda, J. Kim, T.M. Aamodt, Throughput-effective on-chip networks for manycore accelerators, in: Proc. of the 43rd Int'l. Symp. on Microarchitecture, MICRO, IEEE, 2010, pp. 421–432.
[6] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, T.M. Aamodt, Analyzing cuda workloads using a detailed GPU simulator, in: Proc. of the 2009 IEEE Int'l. Symp. on Performance Analysis of Systems and Software, ISPASS, IEEE, 2009, pp. 163–174.
[7] L. Bononi, N. Concer, M.D. Grammatikakis, M. Coppola, R. Locatelli, Noc topologies exploration based on mapping and simulation models, in: Proc. of the 10th Euromicro Conf. on Digital System Design Architectures, Methods, and Tools, DSD, IEEE, 2007, pp. 543–546.
[8] S. Bourduas, Z. Zilic, A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing, in: Proc. of the 1st ACM/IEEE Int'l Symp. on Network-on-Chip, NOCS, IEEE Computer Society, Washington, DC, USA, 2007, pp. 195–204.
[9] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte, K. Compton, ERCBench: an open-source benchmark suite for embedded and reconfigurable computing, in: Proc. of 20th Intl. Conf. on Field Programmable Logic and Applications, FPL, 2010, pp. 408–413.
[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: a benchmark suite for heterogeneous computing, in: Proc. of the 2009 IEEE Int'l Symp. on Workload Characterization, IISWC, IEEE, 2009, pp. 44–54.
[11] N. Concer, S. Iamundo, L. Bononi, aEqualized: a novel routing algorithm for the spidergon network on chip, in: Proc. of Design, Automation, and Test in Europe, DATE, European Design and Automation Association, Leuven, Belgium, 2009, pp. 749–754.
[12] W.J. Dally, Virtual-channel flow control, in: Proc. of the 12th Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 1990, pp. 60–68.
[13] W. Dally, B. Towles, Principles and Practices of Interconnection Network, Morgan Kaufmann, 2004.
[14] R. Das, O. Mutlu, T. Moscibroda, C.R. Das, Application-aware prioritization mechanisms for on-chip networks, in: Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO, ACM, New York, NY, USA, 2009, pp. 280–291.
[15] R. Das, O. Mutlu, T. Moscibroda, C.R. Das, Aérgia: exploiting packet latency slack in on-chip networks, in: Proc. of the 32nd Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2010, pp. 106–116.
[16] V. Dumitriu, G. Khan, Throughput-oriented noc topology generation and analysis for high performance socs, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 17 (10) (2009) 1433–1446.
[17] P. Gratz, B. Grot, S.W. Keckler, Regional congestion awareness for load balance in networks-on-chip, in: Proc. of the 14th Int'l. Symp. on High Performance Computer Architecture, HPCA, IEEE Computer Society, Washington, DC, USA, 2008, pp. 203–214.
[18] B. Grot, J. Hestness, S.W. Keckler, O. Mutlu, Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees, in: Proc. of the 33rd Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2011, pp. 401–412.
[19] B. Grot, J. Hestness, S.W. Keckler, O. Mutlu, Express cube topologies for on-chip interconnects, in: Proc. of the 15th Int'l. Symp. on High Performance Computer Architecture, HPCA, IEEE Computer Society, Washington, DC, USA, 2009, pp. 163–174.
[20] HPArch, MacSim simulator. http://code.google.com/p/macsim/, 2012.
[21] J. Hu, R. Marculescu, Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures, in: Proc. of Design, Automation, and Test in Europe, DATE, IEEE Computer Society, Washington, DC, USA, 2003, pp. 688–693.
[22] J. Hu, R. Marculescu, DyAD: smart routing for networks-on-chip, in: S. Malik, L. Fix, A.B. Kahng (Eds.), Proc. of the 41st Annual Design Automation Conference, DAC, ACM, New York, NY, USA, 2004, pp. 260–263.
[23] IMPACT, Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.
[24] Intel, Ivy Bridge. http://www.intel.com/content/www/us/en/silicon-innovations/intel-22nm-technology.html.
[25] Intel, Sandy Bridge. http://software.intel.com/en-us/articles/sandy-bridge/, 2011.
[26] Intel, Xeon Phi Coprocessor. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.
[27] A. Jaleel, K.B. Theobald, S.C. Steely Jr., J. Emer, High performance cache replacement using re-reference interval prediction (RRIP), in: Proc. of the 32nd Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2010, pp. 60–71.
[28] M.K. Jeong, M. Erez, C. Sudanthi, N. Paver, A qos-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an mpsoc, in: Proc. of the 48th Annual Design Automation Conference, DAC, ACM, New York, NY, USA, 2012, pp. 850–855.
[29] J. Kim, W.J. Dally, D. Abts, Flattened butterfly: a cost-efficient topology for high-radix networks, in: Proc. of the 29th Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2007, pp. 126–137.
[30] J. Lee, H. Kim, TAP: a TLP-aware cache management policy for a CPU–GPU heterogeneous architecture, in: Proc. of the 18th Int'l. Symp. on High Performance Computer Architecture, HPCA, IEEE, Washington, DC, USA, 2012, pp. 91–102.
[31] S. Ma, N.D.E. Jerger, Z. Wang, Dbar: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip, in: Proc. of the 33rd Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2011, pp. 413–424.
[32] A.K. Mishra, N. Vijaykrishnan, C.R. Das, A case for heterogeneous on-chip interconnects for cmps, in: Proc. of the 33rd Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2011, pp. 389–400.
[33] T. Moscibroda, O. Mutlu, A case for bufferless routing in on-chip networks, in: Proc. of the 31st Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2009, pp. 196–207.
[34] O. Mutlu, T. Moscibroda, Parallelism-aware batch scheduling: enhancing both performance and fairness of shared dram systems, in: Proc. of the 30th Annual Int'l. Symp. on Computer Architecture, ISCA, IEEE Computer Society, Washington, DC, USA, 2008, pp. 63–74.
[35] NVIDIA, Fermi: Nvidia's next generation cuda compute architecture, http://www.nvidia.com/fermi.
[36] NVIDIA, Project denver. http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/.
[37] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, A. Karunanidhi, Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation, in: Proc. of the 37th Int'l. Symp. on Microarchitecture, MICRO, IEEE Computer Society, Washington, DC, USA, 2004, pp. 81–92.
[38] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo, J.M. Prellezo, The adaptive bubble router, J. Parallel Distrib. Comput. 61 (9) (2001) 1180–1208.
[39] M.K. Qureshi, Y.N. Patt, Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches, in: Proc. of the 39th Int'l. Symp. on Microarchitecture, MICRO, IEEE Computer Society, Washington, DC, USA, 2006, pp. 423–432.

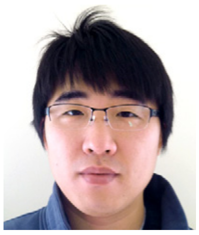
[40] D. Sanchez, G. Michelogiannakis, C. Kozyrakis, An analysis of on-chip interconnection networks for large-scale chip multiproc essors, ACM Trans. Archit. Code Optim. (TACO) 7 (1) (2010) 4.

[41] G. Suh, S. Devadas, L. Rudolph, A new memory monitoring scheme for memory-aware scheduling and partitioning, in: Proc. of the 8th Int'l. Symp. on High Performance Computer Architecture, HPCA, IEEE Computer Society, Washington, DC, USA, 2002, pp. 117–128.

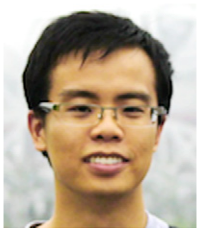
[42] D.H. Woo, H.-H.S. Lee, Compass: a programmable data prefetcher using idle GPU shaders, in: Proc. of the 15th Int'l. Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, ACM, New York, NY, USA, 2010, pp. 297–310.

[43] Y. Xie, G.H. Loh, PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches, in: Proc. of the 31st Annual Int'l. Symp. on Computer Architecture, ISCA, ACM, New York, NY, USA, 2009, pp. 174–183.

[44] Y. Yang, P. Xiang, M. Mantor, H. Zhou, CPU-assisted GPGPU on fused CPU–GPU architectures, in: Proc. of the 18th Int'l. Symp. on High Performance Computer Architecture, HPCA, IEEE Computer Society, Washington, DC, USA, 2012, pp. 103–114.



Jaekyu Lee is a Ph.D. candidate in the School of Computer Science, Georgia Institute of Technology. He received his B.S degree in computer science from Sogang University, Korea in 2007 and M.S degree from the School of Computer Science from Georgia Institute of Technology in 2009. His research interests include high performance computer architecture and high performance computing.



Si Li is a Ph.D. student at Georgia Tech working in the area of computer architecture. His research interests involve dynamic software resilience via instrumentation and the unique demand of memory and cache systems in the massively parallel environment of GPU compute architecture.

Hyesoon Kim is an Associate professor in the School of Computer Science at Georgia Institute of Technology. Her research interests include high-performance energy-efficient heterogeneous architectures, programmer–compiler–microarchitecture interaction and developing tools to help parallel programming. She received a BA in mechanical engineering from Korea Advanced Institute of Science and Technology (KAIST), an M.S. in mechanical engineering from Seoul National University, and an M.S. and a Ph.D. in computer engineering at The University of Texas at Austin.

Sudhakar Yalamanchili earned his Ph.D. degree in Electrical and Computer Engineering from the University of Texas at Austin. Upon graduation, Dr. Yalamanchili joined Honeywell's Systems and Research Center in Minneapolis where he worked as a Senior, and then Principal Research Scientist while he served as an Adjunct Faculty and taught in the Department of Electrical Engineering at the University of Minnesota. He joined the ECE faculty at Georgia Tech in 1989 where he is now a Joseph M. Pettit Professor of Computer Engineering. Since 2003 he has been a Co-Director of the NSF Industry University Cooperative Research Center on Experimental Computer Systems at Georgia Tech. Dr. Yalamanchili has served several periods as the Chair of the Computer Engineering Technical Interest Group within the School of ECE (most recently 2008–2010) and continues to contribute professionally on editorial boards and program committees. He has served as a Distinguished Visitor of the IEEE, and associate editor for the IEEE Transactions on Parallel and Distributed Processing and IEEE Transactions on Computers. He currently serves in the Research Advisory Group to the HyperTransport Consortium and on the Editorial Board of IEEE Computer Society's Computer Architecture Letters and since 2003 has been a Co-Director of the NSF Industry University Research Center on Experimental Computer Systems (CERCS). He is the author of VHDL Starters Guide, 2nd edition, Prentice Hall 2004, VHDL: From Simulation to Synthesis, Prentice Hall, 2000, and co-author with J. Duato and L. Ni, of Interconnection Networks: An Engineering Approach, Morgan Kaufman, 2003. His most recent service includes General Co-Chair of the 2010 IEEE/ACM International Symposium on Microarchitecture (MICRO) and Program Committees for the 2011 International Symposium on Networks on Chip, 2011 IEEE/ACM International Symposium on Microarchitecture (MICRO and 2011 IEEE Micro Top Picks from Computer Architecture Conferences).