Lightweight Emulation of Virtual Channels using Swaps

Mayank Parasar
Georgia Institute of Technology
Atlanta, Georgia
mparasar3@gatech.edu

Tushar Krishna
Georgia Institute of Technology
Atlanta, Georgia
tushar@ece.gatech.edu

ABSTRACT

Virtual Channels (VCs) are a fundamental design feature across networks, both on-chip and off-chip. They provide two key benefits - deadlock avoidance and head-of-line (HoL) blocking mitigation. However, VCs increase the router critical path, and add significant area and power overheads compared to simple wormhole routers. This is especially challenging in the era of energy-constrained many-core chips.

The number of VCs required for deadlock avoidance is unavoidable, but those required for mitigating HoL depend on runtime factors such as the distribution and size of single and multi-flit packets, and their intended destinations. In some cases more VCs are beneficial, while in others they may actually harm performance, as we demonstrate. In this work, we provide a low-cost micro-architectural technique to emulate the HoL mitigation behavior of VCs inside routers, without requiring the expensive data path or control path (vc state and vc allocation) for VCs. We augment wormhole routers with the ability to do an in-place *swap* of blocked packets to the head of the queue. Our design (SwapNoC) can operate at low area and power specs like wormhole designs, without incurring their HoL challenges.

1 INTRODUCTION

Networks-on-Chip (NoCs) are prevalent across manycore CMPs and SoCs today. NoCs need to provide a delicate balance between meeting application's communication latency and throughput demands, while consuming as little real estate in terms of area and power as possible. NoC power continues to remain a concern [11] in the many-core era.

Wormhole routers are the simplest routers and use a simple queue at every input port. The challenge with wormhole routers, however, is *head-of-line (HOL) blocking*. Fig. 1(a) shows an example. The brown packet which wants to go east is blocked by the yellow packet that wants to go south, due to congestion at the south output port.

To avoid HOL, the standard technique is to use Virtual Channels (VCs). VCs are like lanes on a highway, that can allow packets using different output ports to not get blocked by each other. Fig. 1(b) shows that VCs allow the brown packet to traverse to its destination without getting blocked. VCs are prevalent across commercial and research NoCs [3, 4] today[1].
The challenge with VCs, however, is three-fold:

Latency. Flits need to know which VC to sit in when arriving at a router, and require a VC allocation step [3]. More the number of VCs, the larger is the critical path for this step. Most VC routers require at least 2-3 cycles even in the most state-of-the-art designs [3]. VCs also add significant area and power overheads.

Area and Power. Fig. 2 plots the area and power requirements of a wormhole, SwapNoC (this work) and VC routers, as a function of increasing number of buffer slots. We implemented all three designs in RTL, and the numbers are from synthesis using Nangate 15nm FreePDK [7]. We can see that the VC area and power grows to more than 2× that of wormhole as the number of buffer slots go up. The reason is that VCs are often implemented as multiple independent FIFOs, each with its associated state, and muxes to write to and read from one of these FIFOs. Alternate organizations with trade-offs are discussed in Section 2.

Traffic-dependent performance. We performed a design-space exploration by stressing a NoC with myriad synthetic traffic patterns, and observed the low-load latency and saturation throughput across wormhole and VC-based designs. For a fair comparison, we assumed N buffer slots at each input port, which could either all go into one N-deep wormhole FIFO, or shallow VCs (N VCs, each 1-flit deep) or deep VCs (N/2 2-flit deep VCs, N/4 4-flit deep VCs and so on). Fig. 3 plots the distribution of latency and throughput across the designs and patterns, normalized to a wormhole NoC for each traffic pattern. We notice that single-flit packets favor shallow VC designs for throughput, while multi-flit packets favor wormhole NoCs for throughput. Moreover, wormhole NoCs always provide lower latency due to simpler routers.

These 3 observations should make us re-think the cost-benefit of VCs in many-core NoCs. In this work, we propose an alternate light-weight technique to reduce HoL, without requiring VCs. We identify that a blocked flit in a queue can in principle *swap* with the one at the head of the queue. This is possible in hardware because of the cyclic shift-register behavior, as Fig. 4(b) shows. A cyclic shift-register can allow the bits at the output of two latches to get swapped at the clock edge, without requiring another temporary latch. This is unlike the software notion of swap where a temporary buffer is required for a swap. Leveraging this principle, we allow HOL blocked flits that can leave the router to swap to the head of the queue and proceed. We enhance wormhole routers with this feature, and call our design the *SwapNoC*. Fig. 1(c) shows how the blocked packet can swap to the head and can traverse to its destination without getting blocked.

SwapNoC can provide the latency, power and area benefits of wormhole NoCs, and emulate the throughput benefits of VCs. The neat feature of our design is that it can adapt to both single and multi-flit packets, without requiring explicit VCs partitioned into a pool of shallow and deep queues, which can add performance loss when done at design time [2, 13] and add complexity when done dynamically at runtime.

Compared to wormhole and VC baselines, SwapNoC demonstrates up to a 3.7× reduction in latency, and 70%-95% improvement in throughput across synthetic and real workloads. It has 2× lower

[1]This work targets the performance enhancement (HoL mitigation) aspect of VCs. Some VCs would still be required for avoiding protocol or routing deadlocks.

NoCArc'17, October 14, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-5542-1/17/10…$15.00
https://doi.org/10.1145/3139540.3139541

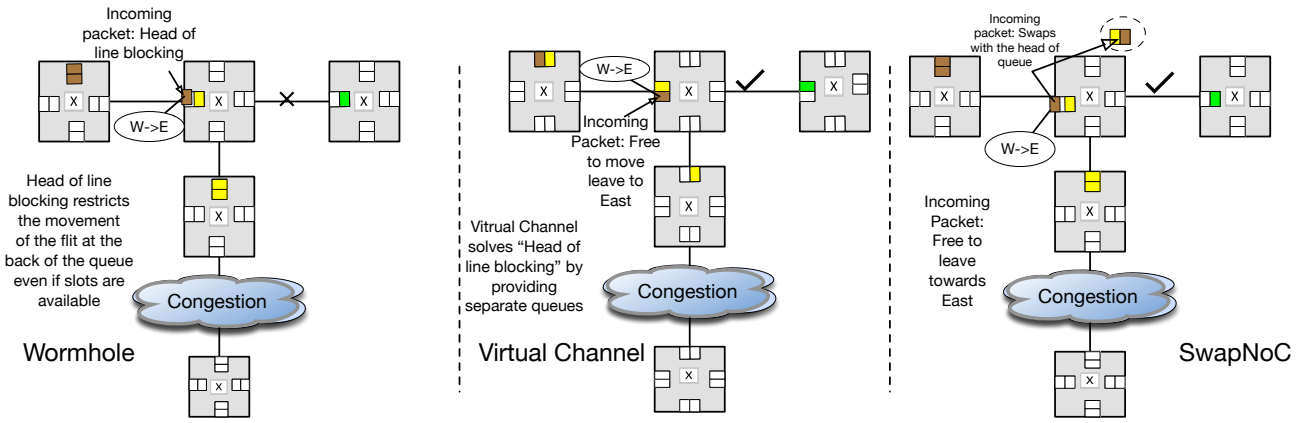


Figure 1: Wormhole vs. Virtual Channels vs. SwapNoC

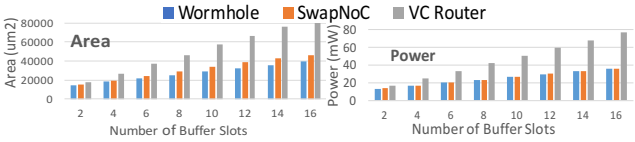


Figure 2: Router Area and Power as a function of buffer slots

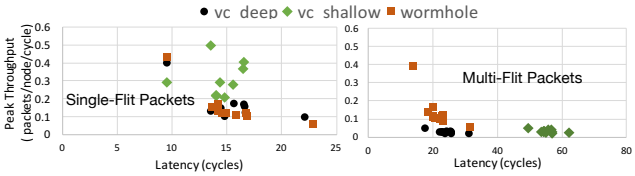


Figure 3: Performance of Wormhole vs. VC-based Designs.

area and 2× lower power than traditional VC based routers, and adds less than 1% power and 8% area overhead over a wormhole router.

Sec. 2 discusses related work. Sect. 3 presents the microarchitecture. Sec. 4 shows evaluations, and Sec. 5 concludes.

2 BACKGROUND AND RELATED WORK

2.1 Flow Control Techniques

To transmit messages within network, usually one of the following five *flow control* techniques are used, with increasing order of complexity:

Circuit Switching. The whole path for the transmission is blocked until the packet is transmitted. Since this leads to poor link bandwidth utilization, it is not preferred in NoCs.

Packet-switching with Store-and-Forward. Packet are stored completely in each router and only allowed to leave once the entire packet has arrived.

Packet-switching with Virtual cut-through (VCT). Packets are allowed to go to the downstream router as soon as an output channel is free. The buffer and link allocation is still done on a packet basis.

Packet-switching with Wormhole. This is similar to VCT, as the links is still allocated on a per packet basis. However, buffers at routers can be smaller than the size of the packet. Thus buffering is done on a flit basis. Therefore this design can work on routers with fewer buffers.

Head-of-Line Blocking. Despite obvious benefits of lesser buffering compared to other packet-switched techniques, wormhole routing suffers from Head-of-Line (HoL) blocking. Consider the case where a tail flit of certain packet is blocked at the head of the queue because the requested output channel for that flit is not free due to congestion. Even if the desired outports of the flits of the packets waiting behind the blocked flit are free, they cannot leave. This is called Head-of-Line (HoL) blocking. As this phenomenon is unpredictable, it can leads to performance degradation.

Packet-switching with Virtual Channels (VCs). VCs reduces HOL, as Fig. 1(b) demonstrates, by assigning separate queues for different packets. Lots of *shallow VCs* are better at heavy traffic; multiple *deep VCs* are better when there is low traffic and each packet has multiple flits. The VC queues are typically partitioned in a static manner during design time. VC-based flow control is one of the most prevalent flow control techniques in use today, and has been used across NoC prototypes [3, 4].

2.2 Buffer Management

In an on-chip scenario, wires are often abundant while real-estate for buffers is expensive. There has thus been a lot of prior work that tries to optimize for efficient usage of buffers - either for performance or for energy-efficiency.

Low-Cost Buffers. One class of solutions has focused on reducing the cost of VC buffers. These span from using bufferless routers [8] to intelligent bypassing of router pipeline [6, 9] to using multiple physical networks [14]. Elastic buffers [12] can also help reduce buffer area and power. These techniques are complementary to our approach and can augment it to reduce area and power further.

Dynamic VC Partitioning. The works most related to ours fall within the category of dynamic VC partitioning [2, 10, 13]. DAMQ [13] was one of the early works in this space that advocated for organizing the available buffers in the form of a linked list, allowing flits of a packet to dynamically allocate one to all of the slots at that input port. Follow-up research has tried to efficiently utilize the buffers for higher throughput [10] by emulating output buffered routers while operating the router at considerable frequencies. ViChaR [2] implements a Unified Buffer Structure (UBS) at each input port, controlled by a Unified Control Logic (UCL) structure, that dynamically maps flits to a given virtual channel to maximize buffer utilization.

In these designs, the router controls each buffer slot individually to dynamically create shallow VCs or deep VCs. The challenge is that full flexibility requires the router to pay the *control overhead of N VCs* if there are N buffer slots at each input port to allow each buffer to act as its own VC and get direct access to the switch if required. This incurs overheads both in terms of complexity, and

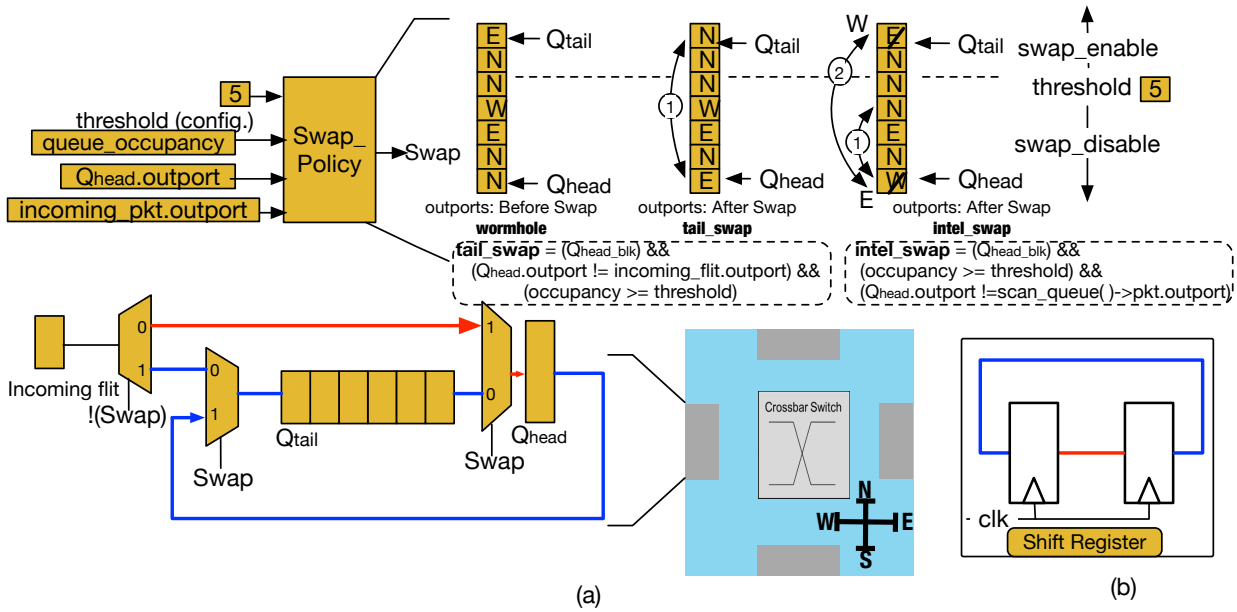


Figure 4: Swap NoC Microarchitecture. For illustration purposes, we show the swap for a single flit packet. Presented above is an example of tail_swap **and** intel_swap **policies. Suppose the North output port is blocked.** tail_swap **enables the packet going East at Q_{tail} to swap with the one at Q_{head} going North (Step 1). With** intel_swap**, a scan of the queue results first in the flit at location Q_{head}+3 (i.e., outport West) getting swapped with Q_{head} (Step 1), and subsequently, if West is also blocked, this gets swapped with the flit at Q_{tail} going East.** intel_swap **enables more number of swaps.**

in terms of area and power. Our work, in contrast does the exact opposite. Rather than using individual buffer slots to dynamically construct VCs, we use a single FIFO and enable it to "act" like multiple VCs by swapping blocked packets dynamically to the head of the queue.

3 THE SWAPNOC

Wormhole routers have two key bottlenecks: (1) HoL blocking at the input port, and (2) losing arbitration for the switch output port due to flits at other input ports contending for the same output port. VCs directly address (1) since each input port has multiple candidates to choose from, not just one. VCs indirectly address (2) as well since VCs provide multiple choices to the switch allocator (SA) which can then try to find the best possible match between input requests and available output ports. However, VCs add tremendous area and power overheads as we have motivated so far. Moreover, designing a switch allocator that does a good matching is a NP-hard problem [5] and most hardware implementations use a simpler separable allocator [5] which independently arbitrates for input and output ports and cannot address (2).

We make a case for solving HoL by a much simpler solution - allow the blocked flit/packet to perform an in-place swap with the head of the queue. We also present policies that help mitigate the output port arbitration challenge.

3.1 Microarchitecture

The fundamental rule in digital hardware design that is used extensively for building state machines is as follows - if multiple flip flops are connected to each other in series, at the rising edge of the clock all inputs move forward by one in parallel without clobbering the old values[2]. For instance, in the circular shift register in Fig. 4(b), at each clock edge, the blue and red signals can keep swapping with each other, without requiring any additional storage for performing the swap. Leveraging this principle, the microarchitecture of the SwapNoC router is shown in Fig. 4(a). In this example, we enable the incoming flit being enqueued at the tail of the queue (Q_{tail}) to swap with the one at Q_{head}. We also allow swaps to occur from intermediate points inside the queue, as we describe later. Swaps occur at a packet granularity (i.e., if enabled, all flits of a blocked packet get swapped to Q_{head} one behind the other), as explained in Section 3.3.

The swap control signals are setup by a swap policy controller. Different policies use different metrics to determine whether to swap or not, which are shown as inputs to the controller in Fig. 4(a).

3.2 Swap Policies

In this work we present 5 policies for swapping. However, our proposed idea of swapping packets is quite powerful and there can be a lot more policies. For all the policies, we assume that the flit at Q_{head} is unable to leave due to zero credits at its outport. If not, a swap will not be triggered.

Tail_Swap. In tail_swap we swap the head of the queue with the incoming packet if the outport of the incoming packet (i.e., at Q_{tail}) is different than at Q_{head}. We trigger tail_swap if the current queue occupancy is greater than a preset *threshold* value. For this policy, we assume lookahead routing [5], i.e., the incoming packet comes with a specific output port that was computed at the previous router. If the output port of the incoming packet is different from that of the packet waiting at Q_{head}, it becomes the candidate for swapping. The swap is initiated if the current queue occupancy is greater than or equal to the threshold.

Intel_Swap. In intel_swap we do not restrict ourselves to swap Q_{head} with the incoming (Q_{tail}) packet; instead upon reaching the threshold intel_swap proactively scans the queue from back to front to find any packet with outport different than that of its head.

[2]Clock synthesis by CAD tools ensures this behavior for correct operation of all synchronous digital circuits.

On finding the first packet during scan with different outport than head, we swap it with Q_{head}.

Fig. 4 demonstrates the tail_swap and intel_swap policies with an example. The *threshold* parameter is only used in the tail_swap and intel_swap policies, not by the other policies. The implicit difference between intel_swap policy and tail_swap policy is that swapping is done more often in intel_swap because after threshold is reached, there are more candidates (packet with different outport than Q_{head}) to choose from as compared to tail_swap.

The cost of scanning can be reduced using a per inport structure which holds the updated outport of all the packets present in the queue with their position. This structure will get updated whenever any packet enqueues, or leaves the queue or get swapped within the queue.

Credit_Swap. credit_swap is based on the insight that a flit with zero credits at its output port could get stuck for many cycles since zero credits is a likely indication of congestion at its downstream router. If such a flit were to move to the Q_{head}, it would cause HoL for other packets. The credit_swap policy is centered around finding such packets and pushing them to the tail of the queue.

credit_swap keeps track of the credit count at all the outports. Whenever any of the outport's credit becomes 0 (which means there is no buffer space available at the given inport of the downstream router), it scans the inport queue starting from front till back. During the scan, if it finds a packet with same outport as the one which has 0 credit, it swaps it with the tail of the queue. This is done at all the inports in the router, to make sure the outport which has no credit has its packets shifted towards the tail of the input queues. This also helps in reducing network congestion.

Random_Swap. random_swap policy tries to shuffle all the inport queues periodically. Shuffling is done by choosing a packet from the queue randomly and swapping it with the head of the queue. This is a heuristic, but can reduce the effect of HoL blocking as each packet comes at the head of the queue with equal probability.

Shuffle_Swap. Recall that there could be two reasons that the flit at Q_{head} is unable to leave the queue, as discussed before - HoL blocking or losing SA in the router. shuffle_swap policy also tries to shuffle all its inport queues periodically like the random_swap policy. The only difference is that when it selects the candidate packet to swap with Q_{head}, it makes sure that the selected packet does not have the same outport as the one at the head. This make head of all inport queues randomly distributed, thus not only reducing the effect of HoL blocking, but also helping increase the chance of winning the SA.

3.3 Multi-flit Packet Swaps

Full-packet Swap. Swaps nominally occur at a packet granularity (i.e., if enabled, all flits of a blocked packet get swapped to Q_{head} one behind the other). During a full packet swap, all flits of the packet can be swapped either serially (if there is only one bypass connection to Q_{head}) or in parallel if there are multiple connections. All flits of a packet are swapped in-order, so there is no re-ordering of flits within a packet.

Since our base design is a wormhole router, we do not allow partial swaps since the body and tail flits of a packet do not carry routing information and rely on following the flit right before it. This is because there is no "VC" to store the per-packet routing information. All links are allocated on a packet granularity. Swaps are disallowed under 2 conditions:

(1) If the flit at the Q_{head} is not a head flit of a packet, then the swap is not allowed. The reasoning is as follows: if the flit at Q_{head} is a body or tail flit, that means that the head flit of this packet already left the router. This is implemented by setting a `head_blk` whenever a head flit reaches Q_{head}, and resetting it when the tail flit leaves.

(2) If the queue does not have enough slots to hold the entire incoming packet, swaps to Q_{head} are not allowed since part of the packet would have been swapped to the front, while the remaining would still be at the previous router waiting for credits.

Flit-level Swap. The two conditions listed above can be relaxed, i.e., partial swaps can be allowed, if each body and tail flit also carries the output port (encoded in 3-bits) at this router. The body and tail flits do not need to carry the full header (which would essentially make each body and tail flit a packet in itself reducing effective bandwidth). In such a scenario, the body and tail flits, that are no longer right behind the head flit, would know which output port to go out from when they eventually arrive at Q_{head} again.

This does not break the correctness of the design, as we describe with an example. Suppose a Packet A is waiting for the East output port and stalled. The head-flit of Packet B going towards South gets swapped and moves to the head of the queue and leaves. Now the East output port becomes free and flits from Packet A start getting sent out. Since the head of the queue is no longer blocked, the other flits of Packet B get queued behind Packet A. This is an acceptable outcome. The goal of the SwapNoC, like VCs, is to ensure that some flit can leave from an input port and output port every cycle. While Packet A was stuck, Packet B was allowed to swap and fulfill this requirement. Once Packet A becomes free, it satisfies this requirement. At the next router at the East output port, all flits of Packet A are still going to be together in the correct order.

3.4 Comparison to VCs.

The goal of SwapNoC is to emulate the behavior of VCs, i.e., the ability for flits to different output ports not get blocked by each other. To that end, different policies for swaps can emulate different behaviors. We give an intuition on how we can seamlessly model VC behavior without any control except the notion of a threshold and the ability for an input flit to swap with the head of the queue.

How to emulate shallow VCs? If a network has a lot of 1-flit packets, a small value of threshold can essentially emulate the behavior of shallow VCs by allowing every new packet going to a different output port to have the ability to bypass a blocked packet.

How to emulate deep VCs? If a network has a lot of multi-flit packets, then having a threshold equal to or greater than the size of the packets can allow new packets to bypass blocked packets.

How is the threshold set? The *threshold* is meant to be a dynamic knob available with the router to tune the swap frequency for tail_swap and intel_swap based on traffic rate and size of packets. In our design, we support both a static version and a dynamic version. In the static version, the threshold is set in the router at reset, based on offline profiling of traffic. In the dynamic version, the router monitors the number of failed switch arbitrations and adjusts the threshold accordingly. When the number of failed arbitrations are high, the threshold is lowered, else it is raised. We demonstrate the impact of the threshold in our evaluations.

Adaptive Schemes. Schemes like credit_swap, rand_swap and shuffle_swap do not need the *threshold* knob to tune and aggressively try to adapt with traffic.

Table 1: Network Configurations (1-cycle router in each)

Wormhole	Wormhole router with a N-flit deep queue.
VC-shallow	N 1-flit VCs (max number of VCs).
VC-deep	2 $N/2$-flit deep and 4 $N/4$-flit-deep VCs.
SwapNoC	N-flit deep queue with Swaps.

Summary. SwapNoC cannot beat VCs cycle-by-cycle in terms of throughput, since fundamentally it operates on heuristics to get a non-blocked flit to Q_{head} while VCs can essentially arbitrate for and choose the best possible candidate every cycle. But the SwapNoC has a lower cycle time, and much lower power and area than VC routers, as we show next. Moreover, as we observe in our results, static partitioning of VCs actually performs worse than SwapNoC, especially with large packet sizes.

4 EVALUATION

4.1 Methodology

Our target NoCs are described in Table 1. We equalize the total buffers (say N) in each router across all designs. All other possible VC configurations should perform between VC-shallow and VC-deep which represent two extremes of the design space for VCs. All routers - wormhole, Swap and VC have a state-of-the-art 1-cycle pipeline. This is an aggressive assumption for VC routers which typically take 2+ cycles due to input and output VC arbitrations [3] that are not required in wormhole and SwapNoC.

We implemented all NoCs in RTL to get pipeline delay, area, and power results post-synthesis using the 15nm Nangate FreePDK library [7]. For design-space exploration inside multicores, we used the gem5 [1] simulator with the Garnet on-chip network model where we modeled the SwapNoC. We assume a 8×8 mesh in all our evaluations.

Traffic Patterns. We evaluate our designs across a suite of synthetic and real traffic patterns. We also define two new synthetic patterns to stress HoLs in the NoC. Tornado_Random_30 is the traditional Tornado pattern - which always sends traffic halfway across the mesh in the *same dimension* without turning, with 30% of the traffic being random, i.e., may want to turn. Edge_50 sends 50% of the traffic to the rightmost node in the same row as the source, and 50% to a random destination. The synthetic traffic runs are done with both single-flit and 5-flit packets, to demonstrate the impact of our NoCs. We also run full-system simulations with PARSEC benchmarks over a MOESI directory protocol. The protocol requires 4 virtual networks (request, response, forward, unblock) for deadlock-avoidance. Data (cacheline) packets are 5-flit, the rest of the packets are 1-flit. Wormhole and SwapNoC use a single FIFO within each vnet, while the VC-based designs use 4 VCs within each vnet.

4.2 Critical Path, Area and Power

RTL synthesis of the SwapNoC router demonstrates that it increases the critical path over a wormhole router by only 8.4-9.4% across queue depths from 2 to 16. This is due to the mux that can read a flit from either the head, or the threshold size depth in the queue. This overhead was well within the timing slack at 1ns, enabling a 1-cycle operation at 1GHz. The VC router, on the other hand, has a critical path close to 2ns when N=16.

Fig. 2 shows that the SwapNoC router is 2× smaller in area and consumes 2× lower power compared to VC routers, as the number of buffer slots in each port goes up. SwapNoC adds 1% power and 8% area overhead over the wormhole router.[3]

[3]We thank Hyoukjun Kwon from Georgia Tech for help with RTL implementation and synthesis of the Swap NoC

4.3 Performance: Synthetic Traffic

Multi-flit Packets. Fig. 5 evaluates the performance of SwapNoC across synthetic traffic patterns using 5-flit packets. With multi-flit packets, the shallow VC design has the highest delay, due to heavy serialization. At low loads, the flit of each packet needs to wait for the credit round trip for sending every flit of the packet. At high loads, more VCs helps push the throughput. Thus this design has the highest throughput across most patterns. The deep VC design on the other hand provides much better low-load latency, and saturates at the same or slightly lower injection rate than the shallow VC. Wormhole saturates the earliest across all patterns, which is its key shortcoming.

The SwapNoC policies provide the best low-load latencies, providing a 61% reduction in latency compared to the shallow VC design, and 28% lower than the deep-VC design. In terms of throughput, SwapNoC provides 88.2-87.6-88.1% better throughput than the VC-based designs for bit_reverse, transpose, and edge_50. With bit_rotation, shuffle and tornado, the throughput of SwapNoC is comparable to that of the VC-based designs. For uniform_random, bit_complement, and tornado_random_30 SwapNoC provides throughput that is in between that of wormhole and VCs.

tornado is an interesting traffic pattern for the SwapNoC since traffic never turns, which would seem to imply that there would never be any HoL blocking. However, there is still HoL blocking for packets that want to get ejected. This is the reason SwapNoC actually improves throughput over the wormhole design even for tornado.

Among the SwapNoC policies, random_swap and intel_swap have slightly better performance than the others.

In summary, SwapNoC provides the performance of wormhole and deep VCs at low-loads, and close to or better throughput than shallow VCs at high loads, essentially modeling a dynamic VC partitioning design without the overheads of managing each buffer slot independently. **Single-flit Packets.** Fig. 6(a) demonstrates the performance of SwapNoC with single-flit packets. With single-flit packets, there are no credit turnaround issues for shallow VCs which provide the best latency and throughput. As discussed earlier in Section 4.2, VCs provide much better opportunities for flits going out of unblocked outports to arbitrate for and leave, compared to the SwapNoC which relies on heuristics to come to the head of the queue. SwapNoC provides about 15% improvement in throughput over wormhole for edge_50 and shuffle, and comparable performance with bit_rotation and other patterns not shown in the interest of space. SwapNoC still beats the Deep-VC design in throughput by 40-50% in edge_50 and bit_rotation.

In summary, with single-flit packets, a deep-VC design suffers tremendously while a shallow VC design performs the best. The SwapNoC is an elegant design choice for providing better throughput than deep VC designs.

Impact of buffer depth and threshold Fig. 6(b) plots the throughput as a function of the *threshold* parameter across multiple buffer depths for the tail_swap policy. For a buffer depth of 4, a threshold of 3 gives a spike in performance. For depths of 8 and 12, the impact of threshold is almost negligible. But with a buffer depth of 16, a threshold of 11-13 gives the best performance.

4.4 Performance: Full-System PARSEC

Fig. 7 demonstrates the full-system performance with PARSEC benchmarks. For benchmarks such as blackscholes and bodytrack, SwapNoC provides 12% lower runtime than VCs, primarily due to the faster router. Its performance is same as that of wormhole as

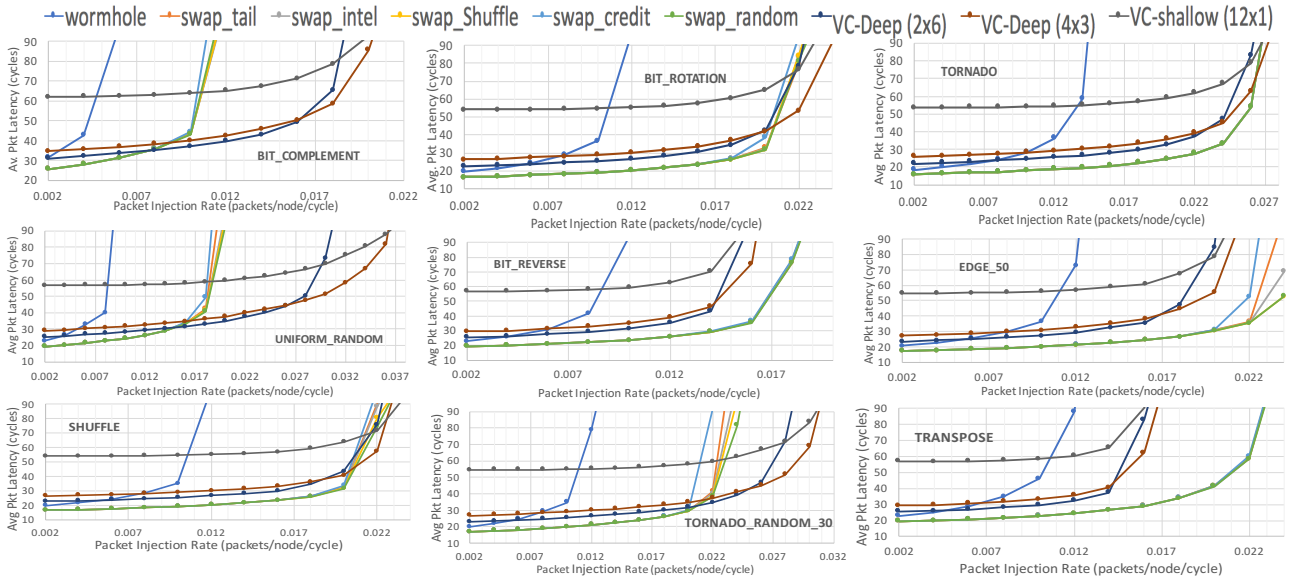
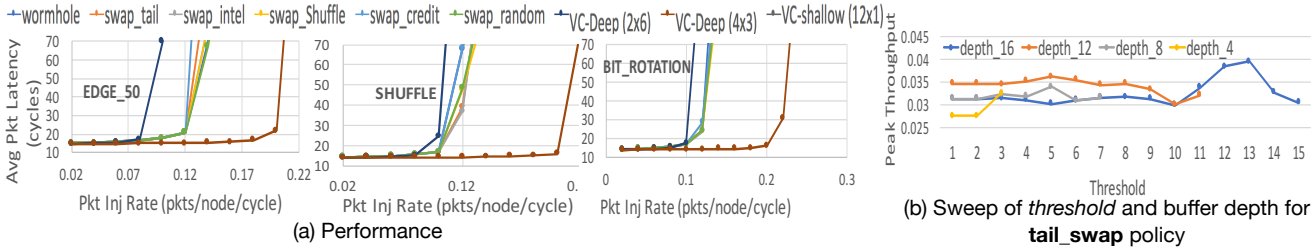


Figure 5: Performance of SwapNoC with multi-flit packets.



(a) Performance

(b) Sweep of *threshold* and buffer depth for **tail_swap** policy

Figure 6: Performance of SwapNoC with single-flit packets.

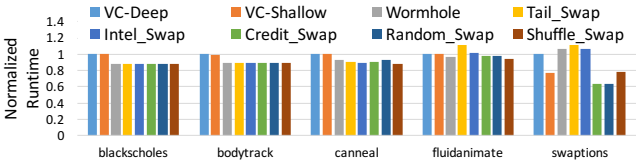


Figure 7: Normalized Full-System Runtime with PARSEC.

there is not enough NoC traffic to cause much HoL blocking. This is the same reason both deep VC and shallow VC have similar performance. With canneal and fluidanimate we see about 5% reduction in overall runtime with shuffle_swap compared to wormhole. But in some cases, tail_swap actually results in a drop in performance. swaptions shows the most dynamic behavior, demonstrating up to 36% reduction in runtime over both VCs and wormhole with credit_swap and random_swap.

In summary, for applications with low network traffic, which is often the case with real workloads, the overheads of VCs is an overkill for many-core systems. SwapNoC has similar overheads as a wormhole router, but can step in to provide higher performance than the wormhole in case of higher traffic, making it a win-win.

5 CONCLUSIONS

We provide a light-weight technique to mitigate HoL without requiring VCs. Our key novelty is the ability for blocked flits to swap with the head of the queue in a wormhole router, without any additional buffers to manage this swap. We add minimal control overhead to perform this swap, and also describe multiple heuristic policies for managing when swaps occur. The SwapNoC shows significant performance, energy, and area benefits over VC-based router designs. We believe that the idea of leveraging swaps goes beyond the policies presented in this paper, and can open up a suite of optimizations for NoC architects and designers.

REFERENCES

[1] N. Binkert et al. 2011. The Gem5 Simulator. SIGARCH Comput. Archit. News 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718
[2] N. Chrysostomos et al. 2006. ViChaR: A dynamic virtual channel regulator for network-on-chip routers. In MICRO' 06. IEEE, 333–346.
[3] C. Clauss et al. 2011. Evaluation and improvements of programming models for the Intel SCC many-core processor. In HPCS.
[4] B. Daya et al. 2014. SCORPIO: a 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering. In ISCA.
[5] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. On-chip Networks. Morgan & Claypool Publishers.
[6] A. Kumar et al. 2007. Express virtual channels: Towards the ideal interconnection fabric. In ISCA.
[7] M. Martins et al. 2015. Open Cell Library in 15nm FreePDK Technology. In ISPD. ACM, 171–178.
[8] Thomas Moscibroda and Onur Mutlu. 2009. A case for bufferless routing in on-chip networks. In ISCA.
[9] A. Psarras et al. 2016. ShortPath: A Network-on-Chip Router with Fine-Grained Pipeline Bypassing. *IEEE Trans. Comput.* 65, 10 (2016), 3136–3147.
[10] R. Ramanujam et al. 2010. Design of a high-throughput distributed shared-buffer NoC router. In NOCS.
[11] A. Samih et al. 2013. Energy-efficient interconnect via Router Parking. In HPCA.
[12] I. Seitanidis et al. 2015. ElastiStore: Flexible elastic buffering for virtual-channel-based networks on chip. TVLSI 23, 12 (2015), 3015–3028.
[13] Yuval Tamir and Gregory L Frazier. 1988. High-performance multi-queue buffers for VLSI communications switches. IEEE Computer Society Press.
[14] D. Wentzlaff et al. 2007. On-chip interconnection architecture of the tile processor. IEEE micro 27, 5 (2007), 15–31.