Carpool: A Bufferless On-Chip Network Supporting Adaptive Multicast and Hotspot Alleviation

Xiyue Xiang[†] Wentao Shi[⋆] Saugata Ghose[‡] Lu Peng[⋆] Onur Mutlu[§‡] Nian-Feng Tzeng[†]
[†]University of Louisiana at Lafayette [⋆]Louisiana State University [‡]Carnegie Mellon University [§]ETH Zürich

ABSTRACT

Modern chip multiprocessors (CMPs) employ on-chip networks to enable communication between the individual cores. Operations such as coherence and synchronization generate a significant amount of the on-chip network traffic, and often create network requests that have one-to-many (i.e., a core *multicasting* a message to several cores) or many-to-one (i.e., several cores sending the same message to a common *hotspot* destination core) flows. As the number of cores in a CMP increases, one-to-many and many-to-one flows result in greater congestion on the network. To alleviate this congestion, prior work provides hardware support for efficient one-to-many and many-to-one flows in buffered on-chip networks. Unfortunately, this hardware support cannot be used in *bufferless* on-chip networks, which are shown to have lower hardware complexity and higher energy efficiency than buffered networks, and thus are likely a good fit for large-scale CMPs.

We propose Carpool, the first bufferless on-chip network optimized for one-to-many (i.e., multicast) and many-to-one (i.e., hotspot) traffic. Carpool is based on three key ideas: it (1) *adaptively forks multicast flit* replicas; (2) *merges hotspot flits*; and (3) employs a novel *parallel port allocation* mechanism within its routers, which reduces the router critical path latency by 5.7% over a bufferless network router without multicast support.

We evaluate Carpool using synthetic traffic workloads that emulate the range of rates at which multithreaded applications inject multicast and hotspot requests due to coherence and synchronization. Our evaluation shows that for an 8×8 mesh network, Carpool reduces the average packet latency by 43.1% and power consumption by 8.3% over a bufferless network without multicast or hotspot support. We also find that Carpool reduces the average packet latency by 26.4% and power consumption by 50.5% over a buffered network with multicast support, while consuming 63.5% less area for each router.

KEYWORDS

bufferless networks, on-chip networks, deflection routing, multicast, hotspot traffic, router design, coherence, synchronization

ACM Reference format:
X. Xiang et al. 2017. Carpool: A Bufferless On-Chip Network Supporting Adaptive Multicast and Hotspot Alleviation. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 11 pages.

1 INTRODUCTION

Chip multiprocessors (CMPs) consist of multiple processor cores that can communicate with each other. As the manufacturing process technology continues to scale down, commercial CMPs have increasing core counts, and some manufacturers are exploring cores with over a thousand cores [39]. To exploit the high degree of parallelism available on CMPs, modern applications often spawn multiple threads that execute on different cores. These threads share data with each other, by maintaining multiple copies of the data within the caches of multiple cores. To ensure correctness, the threads (1) must maintain coherence between the copies of shared data that reside in each core, and (2) coordinate with each other through the use of synchronization primitives. CMPs employ on-chip networks (NoCs) to perform the core-to-core communication necessary for coherence and synchronization.

As core-to-core communication can take place frequently, it is necessary to optimize the performance and energy efficiency of the on-chip network. Traditionally, on-chip networks maintain *buffers* at each router, which allow some requests to wait at the router when they contend for access to an output port that is being used by another request. These buffers can consume a substantial amount of the router area and a large portion of the network power. To reduce the area and power, bufferless on-chip networks were proposed [33]. A bufferless on-chip network avoids the need to buffer requests within the router. One common way of doing so is by employing a routing algorithm called *deflection routing* [5, 43]. When two requests at a router contend for the same output port, deflection routing sends one of the requests to a port other than the one it requested (i.e., it *deflects* the request to another router), avoiding the need to buffer the request [33]. Many works have built upon the initial bufferless on-chip network [33] to further reduce complexity [20] and improve performance and quality-of-service [3, 4, 9, 15, 19, 25, 28, 29, 36, 37, 50, 51]. A bufferless network delivers performance close to a buffered on-chip network [20, 33], despite employing much simpler hardware and consuming much less power. Thus, as increasing core counts require a greater number of routers in the on-chip network, bufferless on-chip networks are a compelling design choice for future CMPs, as shown by various prior works [7, 11, 20, 33, 36, 37].

A major limitation of existing on-chip networks is that they are optimized for one-to-one packet flows (i.e., each packet has a single source and a single destination), known as *unicast* traffic. This leaves much to be desired when on-chip networks are employed in CMPs, since many communications between cores exhibit a one-to-many packet flow, known as *multicast* traffic, or a many-to-one packet flow, known as *hotspot* traffic. Such multicast or hotspot traffic often occurs during coherence and synchronization operations, as we demonstrate with three examples: (1) With a directory-based coherence protocol, when a shared cache line is evicted, this often involves sending a sequence of invalidation packets from the home directory node to all cores that contain a copy of the cache line, forming a one-to-many flow. (2) Each core containing a copy of the cache line sends an ACK (acknowledgment) back to the home directory node to indicate that it performed the invalidation, forming a many-to-one flow. (3) Multiple cores can also send identical requests to a home directory node when the cores are trying to access a shared variable for synchronization, again forming a many-to-one flow.

There are many mechanisms for *off-chip* networks that improve the efficiency of one-to-many [44, 47] or many-to-one [8, 49, 52] flows. However, these mechanisms are not suitable for on-chip networks, due to stringent on-chip area and power budgets that exist for on-chip networks. Although recent works [2, 18, 26, 31, 40, 41, 48] attempt to optimize one-to-many and/or many-to-one

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079090

flows for on-chip networks, these techniques are based on *buffered* networks, and often incur a large chip area, large power budget, and complex flow control. No previous work provides multicast or hotspot traffic support for *bufferless* on-chip networks. **Our goal** is to provide efficient one-to-many and many-to-one flow support in a bufferless on-chip network, while maintaining the low design complexity and high energy efficiency that make bufferless networks very desirable for CMPs.

In this work, we propose *Carpool*, the first bufferless on-chip network with hardware support for efficiently handling multicast and hotspot traffic. Carpool is based on three key ideas. First, in order to send a network request to multiple cores, Carpool *adaptively replicates multicast requests*, doing so only when the replicated requests do *not* introduce network congestion. Second, Carpool detects and *merges hotspot requests* from multiple source cores that are destined for the same core. Third, Carpool uses a novel *parallel port allocation* mechanism that provides efficient multicast replication support while minimizing the critical path latency of the network router. Carpool uses its *adaptive multicast replication* and *hotspot merging* opportunistically to alleviate contention at both the network interface (NI) and the routers.

We evaluate Carpool on a 64-core CMP with an 8×8 mesh network, using synthetic traffic workloads that emulate the range of rates at which multithreaded applications inject multicast and hotspot requests due to coherence and synchronization. We show that with its efficient multicast and hotspot support, Carpool improves significantly over a traditional bufferless network without such support [33], reducing the average packet latency by 43.1% and power consumption by 8.3%. Carpool also outperforms a buffered on-chip network [31] with hardware support for multicasting, reducing the average packet latency and power by 26.4% and 50.5%, respectively, while consuming 63.5% less area.

We make the following contributions in this work:

- We demonstrate that the multicast and hotspot traffic injected by multithreaded applications for coherence and synchronization operations can quickly saturate a bufferless on-chip network. We show that there is a need for efficient multicast and hotspot support within bufferless on-chip networks for CMPs.
- We propose Carpool, which enables adaptive multicast request replication and hotspot request merging in a bufferless on-chip network. We demonstrate that Carpool outperforms and consumes less power than both a bufferless on-chip network without hardware support for multicast and hotspot traffic, and a buffered on-chip network with efficient multicast support.
- We develop a novel parallel port allocation mechanism that efficiently supports multicast request replication. Our allocation mechanism reduces the critical path latency of the network router by 5.7% over a router for a traditional bufferless on-chip network, even though our router provides dedicated hardware support for multicast and hotspot traffic.

2 BACKGROUND AND RELATED WORK

A core within a CMP sends a *request* to one or more other destination cores in the CMP across an on-chip network. When the core's *network interface* (NI) injects a request into the network, the request is split up into multiple *packets* that travel through the network *routers* to reach the request destination(s). Each packet is further split up into one or more *flits*, the base unit of data movement within the on-chip network. Each router receives one or more incoming flits from different packets on its *input ports*, and must arbitrate the order in which these flits are sent to the router's *output ports* (including flits whose destination is a core attached to the current router). The NI of a destination core uses *miss status holding registers* (MSHRs) to reassemble the flits of each packet when they arrive (i.e., when the network *ejects* the flits) [20].

In this section, we discuss related work in the area of on-chip networks. First, we provide background on how bufferless on-chip networks transport flits across a CMP. Then, we study existing mechanisms to support multicast traffic, broadcast traffic, and hotspot alleviation in *buffered* on-chip networks.

2.1 Bufferless On-Chip Networks

There has been much work in the area of *bufferless* on-chip networks [3, 7, 9, 11, 15, 19–21, 25, 28, 29, 33, 36, 37, 51], which avoid the need for the large buffers in traditional networks. A deflection-based bufferless on-chip network [33] handles flit arbitration for the output ports by guaranteeing that all flits arriving at the router are sent out along one of the router's output ports. By doing so, the router does not need to buffer any of the flits. There are two major characteristics that are required to ensure deadlock-free operation in a bufferless on-chip network with deflection routing. First, the router must ensure that each incoming flit is assigned to an output port, even if the assigned output port is not the port requested by the flit [33]. This ensures that no flit is lost in the network even when a flit loses port arbitration. Second, to ensure that deflection routing can always map an incoming flit to an output port, a router must have at least as many output ports as it does input ports.[1]

A bufferless on-chip network has two key advantages over a buffered network. First, a bufferless network delivers performance that approaches the performance of a buffered network [7, 11, 20, 21, 33], despite employing much simpler hardware and consuming much less power. Second, a bufferless network does not need any *flow control* (i.e., a router does not need to exert backpressure to prevent a neighboring router from sending an incoming flit), as the input port of a neighboring downstream router is *always* able to accept a new flit, due to the lack of queued flits contending for output ports in the downstream router. As a result of these properties, a bufferless network is an attractive option for future CMPs, as it requires much simpler hardware, and is thus likely more scalable [7, 11, 21, 36, 37], than a traditional buffered network.

One drawback of a bufferless network is that its lack of buffers prevent the efficient handling of multicast operations, broadcast operations, and hotspot traffic. In the rest of this section, we examine existing mechanisms that improve the support of each of these operations within *buffered* on-chip networks. These prior works require network traffic to be routed along the *minimal* path to avoid duplicate packet delivery. Thus, they cannot be adapted easily to bufferless networks, where the lack of buffers requires techniques such as deflection routing that can often lead to non-minimal routes.

2.2 On-Chip Networks with Multicast Support

A *multicast* operation represents a one-to-many flow, where a network request is sent by one core to m different destination cores. In an extreme case, a multicast operation can be sent to *all* of the other cores in the network (known as a *broadcast* operation). Conventionally, a multicast operation with m destinations triggers m independent unicast requests, whose flits are injected *sequentially* and transferred independently across the network. The injection of a large number of flits due to a multicast operation can induce heavy network congestion. To alleviate the heavy contention caused by the flits of multicast operations, many mechanisms provide improved multicast and broadcast support in buffered on-chip networks. These mechanisms are divided into two major groups: path-based approaches [2, 23, 32] and tree-based approaches [2, 18, 26, 31, 40, 41, 48]. The fundamental trade-off between these different approaches is whether they keep the router simple at the expense of greater network traffic (path-based), or

[1] Deflection routing can lead to livelock issues. As such, livelock freedom is critical to provide in bufferless networks. Prior works [20, 33] provide simple mechanisms for ensuring livelock freedom, and we refer the reader to them.

they modify the router significantly to reduce the redundant flits due to multicast operations (tree-based).

The path-based approach does not replicate the multicast operation into m different unicast requests [2, 23, 32]. Instead, the approach issues a single multicast request, and designates each flit of the request as a *multicast flit* by tagging it with multiple destinations. The flits are transmitted from one destination to another *sequentially* through the network to perform the multicast. A broadcast operation under the path-based approach traverses *all* nodes in the network one by one, resulting in very high latency.

The tree-based approach initially also does not replicate the multicast operation [2, 18, 26, 31, 40, 41, 48]. Instead, the approach lets each multicast flit first travel along a *common path*, which aims to reduce the number of flits injected into the network. This common path consists of the routers that all the flits of all m unicast requests would have traversed. Once a multicast flit reaches the point where the routes of the unicast requests would have diverged (i.e., it reaches a *branching node*), the router at the branching node *replicates* the flit, such that each branch now has its own copy of the multicast flit. In particular, one approach [41] uses the header flit of the multicast request to set up the path (including branches) for subsequent payload flits, which is known as *virtual cut-through routing* [27]. However, virtual cut-through routing can lead to a very long network latency, as the route must be recomputed for each multicast packet injected into the network [41].

Virtual Circuit Tree Multicasting (VCTM) [18] reduces the routing overhead by reusing the same routing tree for *multiple* multicast requests. To construct a multicast tree route, VCTM converts the *first* multicast request into multiple specially-tagged unicast requests. These unicast requests construct a multicast tree progressively within the network, by identifying the routers at which the routes of the unicast requests diverge (i.e., branch). VCTM stores this branching information in a table inside each router. Subsequent multicast operations destined for the same set of destination nodes are injected into the network as multicast requests. The flits of these multicast requests are replicated at each branching node based on the existing multicast tree constructed by the initial unicast requests. VCTM reduces the time required to establish the network path for the requests. However, VCTM has two drawbacks. First, a multicast routing table is needed in each router to track which multicast requests require branching, and the table may become prohibitively expensive when the network size scales. Second, VCTM exhibits little benefit if the majority of the multicast operations do *not* share the same destination, as the tree constructed for one multicast operation is unlikely to be reused by a subsequent multicast operation. Instead, each operation must construct a new tree using multiple unicast requests, undoing the benefits of VCTM.

Two other tree-based approaches are specialized for particular types of networks. Recursive Partitioning Multicast (RPM) [48] uses a modified routing algorithm to support multicast operations. When a multicast flit is injected, RPM partitions the network into eight parts, based on the location of each node relative to the source node. RPM uses this partitioning, along with a set of fixed port priorities, to determine whether a multicast flit should be replicated (which occurs when the destination nodes of the flit reside in network partitions that cannot be efficiently accessed through only one network port). RPM repeats this decision at every router that the flit travels to, avoiding the need to set up a path for each multicast packet. While RPM performs better than VCTM by avoiding multicast route setup, RPM can provide multicast support only for a wormhole router [13]. MRR [2], on the other hand, is a specialized mechanism that provides multicast support based on the Rotary Router [1], by enabling packet replication at low load and disabling replication at high load.

Other works provide special support for *broadcast* traffic in buffered on-chip networks. Specifically, bLBDR [40] provides tree-based multicast and broadcast support within a predefined region of the network. However, bLBDR may fail to provide multicast support if the destinations are scattered across multiple regions of the network. FANIN/FANOUT is a routing mechanism [31] that attempts to achieve ideal load balancing by randomly selecting the path and branches of a broadcast routing tree. If extended for multicast traffic support, FANIN/FANOUT often incurs high hardware complexity for two reasons: (1) encoding the destination list in the flit header involves linear complexity with respect to the network size; and (2) if a multicast flit fails to get all of its requested ports at any router, the flit has to stay in the virtual channel buffer at that router. This increases the turnaround time for the virtual channel, as the channel uses credits to manage the number of flits in flight. The increased turnaround time can hurt performance, or can require deeper virtual channels (i.e., larger buffers), and hence, higher complexity, to deliver the same performance.

Aside from the limitations described above for each specific mechanism, all prior studies are based on *buffered* networks and *require* the multicast or broadcast traffic to be routed through the *minimal* path to avoid duplicate packet delivery. Due to the non-minimal nature of deflection routing, these prior approaches *cannot be directly applied to bufferless networks*. To our knowledge, there has been no prior work providing specialized multicast and broadcast support for traffic aggregation in a bufferless on-chip network.

2.3 Hotspot Alleviation in On-Chip Networks

A *hotspot* refers to a destination core that receives multiple requests originating from multiple source cores, representing a many-to-one flow. As the flits of these multiple requests (called *hotspot flits*) approach the destination core in the network, they are likely to traverse similar routes, causing congestion as they all compete for access to the same network links and resources. Adaptive routing [25, 34, 39] and source throttling [3, 4, 9, 16, 17, 36, 37, 42, 46] can alleviate the network congestion. However, such mechanisms are often reactive, and are unable to eliminate redundant traffic.

As hotspot flits have the same destination, they can be *merged* and transferred across the network at the same time [52]. In prior work [31], the flits of acknowledgments (ACKs) triggered by the same broadcast request are aggregated into a *hotspot flit*. After aggregation, each hotspot flit tracks the total number of ACKs it represents. However, simply tracking the number of ACKs may not be sufficient during synchronization operations, as the destination might require precise knowledge about the IDs of each individual ACK sender. For example, when an application uses a lock to control the execution of a critical section, multiple cores can request access to the critical section. The lock owner grants critical section access to only one of the requestors at a time, and it needs the core IDs know which requestors it must respond to. As we will see in Section 4.2, Carpool provides a more generalized approach to aggregate a variety of hotspot traffic with different requirements.

3 MOTIVATION

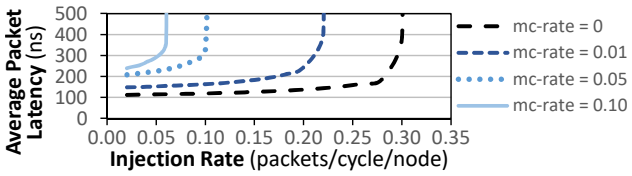
In this section, we motivate the need to support efficient multicast operations and hotspot alleviation in bufferless on-chip networks. First, we study the impact of multicast and hotspot requests on the average packet latency in a traditional bufferless network. Then, we use an example to demonstrate the potential benefits of (1) employing multicast flits that *fork* at intermediate routers in the network, and (2) merging hotspot flits in a bufferless network. We use the basic principles of this example to drive the design of Carpool, which we describe in Section 4.

3.1 Impact of Multicast and Hotspot Flits

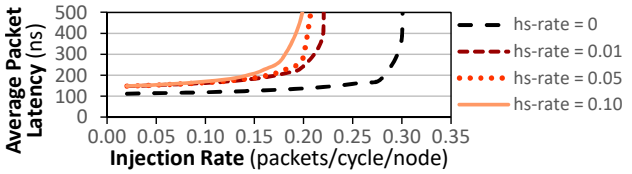
To study the impact of multicast and hotspot traffic in a bufferless on-chip network [33], we run two experiments that characterize

the impact of *network injection rates* for multicast and hotspot traffic on an 8×8 mesh based BLESS network [33]. Section 6 describes our experimental methodology.

In the first experiment, we examine the effect of the *multicast injection rate* (*mc-rate*), where *mc-rate* is the fraction of network requests injected into the network that are multicast network requests. For the multicast request, we randomly select one of the network nodes to inject the request, and we randomly pick m destination nodes (ensuring that the destination node is not the source node) for a random value of m between 1 and 63, inclusive. All other requests injected into the network are unicast requests, with a randomly-selected destination. For four different values of *mc-rate*, we measure the average packet latency as we vary the network injection rate. Figure 1a shows that as *mc-rate* increases, the injection rate at which the bufferless network saturates greatly decreases. We observe that when *mc-rate* increases from 0 to 0.10, the average packet latency increases by almost 3×, causing the network to saturate at an injection rate of only 0.06 packets/cycle/node.



(a) Impact of multicast injection rate on packet latency



(b) Impact of hotspot rate on packet latency

Figure 1: Packet latency as multicast and hotspot traffic varies in an 8×8 mesh bufferless on-chip network.

In the second experiment, we examine the effect of the *hotspot rate* (*hs-rate*), where *hs-rate* is the fraction of network requests injected into the network that are destined to a common hotspot destination node. We randomly designate one of the nodes to be a hotspot, and with probability *hs-rate*, a unicast request injected by another node in the network is sent to the hotspot node. All other requests injected into the network are unicast requests that are sent to a randomly-selected destination. For four different values of *hs-rate*, we measure the average packet latency as we vary the network injection rate. Figure 1b shows that as *hs-rate* increases, the injection rate at which the bufferless network saturates decreases significantly. We observe that when *hs-rate* increases from 0 to 0.10, the average near-saturation latency (i.e., the packet latency just before network saturation occurs) increases by 39.2%, causing the network to saturate at an injection rate of only 0.17 packets/cycle/node.

There are two key reasons why increasing *mc-rate* or *hs-rate* prolongs the average packet latency. First, flits associated with a multicast request must be converted into m unicast flits that are serialized at the injection point, due to the lack of hardware multicast support. The serialization can take up to m cycles for a multicast request with m destinations to complete network injection. Second, hotspot traffic is more likely to converge on similar paths, causing increased flit deflection. As hotspot flits often contain the same information, network bandwidth is wasted when the flits are transmitted individually. The large number of flits due to multicast requests and hotspot traffic increase the network load, leading to a high flit deflection rate in the network. For example, we find that the deflection rate increases by an average of 31.8× when *mc-rate* increases from 0 to 0.10, prior to network saturation (injection rate < 0.06). Deflected multicast and hotspot flits not only slow down the threads that generated the requests, but also deflect flits from other threads unnecessarily, degrading overall system performance [9, 19]. Therefore, providing efficient support for multicast and hotspot traffic is critical to ensure high performance for bufferless on-chip networks.

3.2 Forking and Merging Flits

In order to alleviate the high amount of network congestion caused by multicast and hotspot requests, an on-chip network can provide dedicated hardware support for both of these types of requests. For a multicast request, instead of converting the request to multiple unicast requests at the source node, the source node can inject a *single multicast request*, which gets forked at intermediate routers where the routes to the destination nodes follow two or more output ports of the router (called *multicast flit forking*). For hotspot requests, when an intermediate router detects that two requests destined for a common destination node contain the same content, the router can *merge the requests* into a single request (called *hotspot flit merging*). By forking multicast requests and merging hotspot requests at intermediate nodes, the number of inter-router transmissions decreases significantly in the entire network.

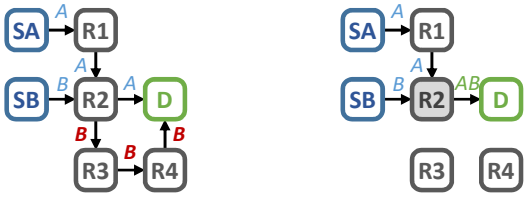
Figure 2 illustrates the potential of forking requests, using an example multicast request that needs to transmit one flit of data from source node S to destination nodes D1 through D6. Without support for multicast forking, the source node injects six independent flits sequentially into the network, as shown in Figure 2a, where each flit is destined for a different node. This requires 15 node-to-node transfers of flits in the network, and can take as many as 10 cycles to complete, if the flit that has to travel to node D6 is injected last. If the intermediate routers had the ability to fork flits from multicast requests, as shown in Figure 2b, the number of node-to-node transfers would reduce to 6, and the request would be completed in 5 cycles.

(a) Without multicast forking **(b) With multicast forking**

Figure 2: Example of how *multicast flit forking* reduces network traffic. The number on each link indicates the load of the link. Shaded routers fork the multicast flit.

In Figure 3, we show an example of hotspot requests, where two requests, from source nodes SA and SB, are both trying to send one flit with the same payload to destination node D. In this example, node SA injects its flit one cycle before node SB does. Without support for hotspot merging, the flits from the two nodes reach node R2 at the same time, and contend with each other for the output port to node D, as shown in Figure 3a. Since only one flit can win port arbitration (the flit from node SA in our example), the other flit (the flit from node SB) is deflected to node R3. As a result of the deflection, it takes 7 node-to-node transfers to complete the two requests, requiring 6 cycles. If node R2 could *merge flits* from two hotspot requests together, as shown in Figure 3b, the two flits would be merged into one flit, which would eliminate port contention and thus the deflection. As a result, the requests would require only 4 node-to-node transfers in total, and would be complete in 4 cycles.

As our examples show, a network with hardware support for forking multicast requests and merging hotspot requests can reduce both traffic and latency. Unfortunately, existing bufferless on-chip networks do *not* include such support, and thus can suffer from the



(a) Without hotspot merging (b) With hotspot merging

Figure 3: Example of how *hotspot flit merging* reduces network traffic. The letters on each link indicate the source node(s) of the packet using the link; bold letters indicate a deflected flit. Shaded router merges the two flits.

high latency and early saturation demonstrated in Section 3.1. The effectiveness of multicast forking and hotspot merging motivates our design for Carpool, which we describe in the next section.

4 CARPOOL DESIGN

We introduce a new bufferless on-chip network called Carpool, which provides efficient hardware support for *multicast forking* and *hotspot merging*. In Carpool, each flit within the network is tagged as a (1) unicast flit, (2) multicast flit, or (3) hotspot flit, using a 2-bit encoding. For a multicast request (i.e., a request destined for more than one node), Carpool injects a single set of multicast flits into the network, instead of replicating the request into multiple unicast requests and injecting the flits of each request sequentially. Carpool generates hotspot flits for special responses (e.g., ACKs for coherence, synchronization requests)[2] sent by multiple nodes to a common destination node. For all other requests that are destined to a single node, Carpool injects unicast flits.

The intermediate routers of Carpool can decide to fork new replicas of a multicast flit that is traversing the network, based on the *destination list* contained in the multicast flit header. A multicast flit is forked only when a *productive* port (i.e., one of the output ports requested by the multicast flit) is available for each copy of the flit. Likewise, the intermediate routers of Carpool merge multiple hotspot flits en route to the same destination node into a single hotspot flit when the flits arrive *simultaneously* (at the same cycle) at the router.

A flit spends two cycles in each router, and the router is pipelined. As each router does not have input or output buffers, Carpool uses deflection routing to resolve flit contention, similar to BLESS [33]. Each flit that arrives at a router is either routed to the port that it requests, allowing the flit to make forward progress toward its destination, or is deflected to another available output port in case another contending flit has acquired access to the requested output port. To ensure that deadlock does not occur, and to minimize network congestion, multicast flits are forked only when there are enough output ports available for *all* flits entering the router *and* for *all* multicast flit replicas. When a multicast flit cannot be forked, the flit travels to its destination nodes sequentially.

In the rest of this section, we describe in detail how Carpool performs multicast flit forking (Section 4.1) and hotspot flit merging (Section 4.2).

4.1 Multicast Flit Forking

In Carpool, a multicast flit can be forked as long as a productive output port is available for each replica, thus ensuring deadlock-free behavior. In a bufferless network router, all flits that arrive at the same cycle are sent out on their assigned output ports simultaneously. As a result, we can avoid deadlock by simply guaranteeing that the number of flits (including replicas) that request output ports at any cycle is less than or equal to the total number of output ports in the router. This guarantee can be expressed as:

$$incoming - removed + replicas \leq outPorts \quad (1)$$

where *incoming* is the number of flits that arrived together (i.e., at the same cycle) to the router (including flits injected from the local input port), *removed* is the number of the incoming flits that are being removed from the network by the router, *replicas* is the number of multicast flit replicas that the router can create during forking, and *outPorts* is the number of output ports available at the router. The router removes a flit from the network if the flit is (1) at its destination node, or (2) a hotspot flit and is merged with another hotspot flit. This equation ensures that all incoming flits and their replicas are able to move through the router pipeline and onto the next router without stalling and without requiring any buffering or dropping.

Destination Encoding. A multicast flit must carry a list of all of its destination nodes within its header, as an intermediate router uses the destination list to determine which output ports are productive ports for the flit, and whether the multicast flit should be replicated. Prior approaches to multicasting in *buffered* on-chip networks either use an identifier that indicates the pre-established path a multicast flit should take through the network [18, 41], or store an n-bit vector in the flit header to indicate all n potential destination nodes in the network [48].[3] As discussed in Section 2, setting up a pre-established path for multicast flits usually incurs a long latency, fails to exploit path diversity, and cannot adapt to changes in network conditions during packet transmission. Directly encoding all n potential destinations with an n-bit vector does not scale well [10, 48], as we must increase the size of the flit, and therefore the width of each channel, linearly with the increase in network node count.

To reduce the impact of scaling the network node count, Carpool uses a two-level hierarchical representation for the destination list. The destination list is represented as the tuple $\{clusterID, dstList\}$. In Carpool, we partition a network into 2^c clusters, each of which contains multiple routers and is indexed by a c-bit *clusterID* number. Each node belongs to one of the clusters in the network. Within a cluster, we can have up to m nodes. The nodes inside a cluster are identified using an m-bit vector called *dstList*. By using an m-bit encoding for m nodes, we can select multiple destination nodes within a cluster for each multicast flit. To improve scalability, a multicast flit can be destined to only one cluster, but is allowed to go to multiple destinations within the destination cluster. If a source node needs to send a request to multiple destinations spread across k clusters, its network interface injects k multicast requests, one for each of the clusters.

With the hierarchical destination representation in Carpool, each multicast flit must carry a destination list of size $c+m$ bits. To avoid the need for additional wires, Carpool transmits the m bits used to represent *dstList* along wires that were previously used by a unicast flit for the payload. As a result, Carpool often needs more flits to send a multicast request than it does for a *single* unicast request. Even then, a multicast request destined for d nodes transmits much fewer flits in total than sending d unicast requests. Let us look at an example where a b-bit request is being sent to d nodes within a single cluster, where the network channel uses h bits to transmit the data. Carpool uses a single multicast request, consisting of $b/(h-m)$ flits, to send the data to all d nodes, where $h-m$ represents the payload size of a multicast flit after we subtract the m bits used for the destination list. In a traditional bufferless on-chip network, which can send only unicast requests, the network uses $d \times b/h$ flits

[2] Carpool focuses on special response flits since they are easier to merge, because their payload is small or non-existent.

[3] Prior works [31, 40] that provide only support for broadcasting, *not* multicasting, do *not* carry any destination information, but they also *cannot* support efficient multicast operations.

to send the data using d separate requests. Carpool uses fewer flits to send the data as long as the following relationship holds:

$$m < h - \frac{h}{d} \quad (2)$$

In our work, we assume that Carpool uses a 4-bit *clusterID* and a 64-bit *dstList* (i.e., $c = 4$ and $m = 64$), which can uniquely identify up to 1024 nodes. For a network channel where 128 bits are used to transmit data, a multicast request costs no more than a series of unicast requests as long as d is at least 2. In other words, a multicast request in Carpool *never* transmits more flits than sending d separate unicast requests to the d destinations.

Adaptive Forking. Deciding *when to fork* a multicast flit is critical in determining the performance of the network. We observe that when there is a low overall load on the network, using multicast flits and forking flit replicas can reduce network latency, improve network utilization, and increase the overall network throughput. However, we observe that when the network load is high, multicast flit replication can increase the probability of reaching network saturation. This is because, after a flit is forked, its replicas increase the *total number of flits* in the network, which can lead to saturation at high load.

During periods of high load, bufferless networks employ techniques to control the network injection rate [9, 15, 36, 37] to avoid reaching saturation. Many metrics have been proposed to quantify network congestion [9, 12, 15, 24, 30, 36, 37]. In particular, the *injection starvation rate* (σ, the fraction of cycles where a node attempts to inject a flit but is unable to) is considered to be a good congestion indicator in a bufferless on-chip network [15, 36, 37]. When the network load increases, flit injection is more likely to fail due to congestion, which in turn increases the starvation rate. The starvation rate can be obtained easily by maintaining a local counter within each router.

Carpool extends upon existing injection control techniques [36, 37] to adaptively determine when each router should disable multicast flit support. A router disables multicast flit support when its injection starvation rate exceeds a predetermined threshold.[4] When multicast flit support is disabled, a source node attached to the router is not allowed to inject a multicast flit. Instead, the node must inject multiple unicast flits, as is done in traditional bufferless on-chip networks. By using unicast requests, we effectively ensure that all necessary flit replicas are already created at the time of injection, and that the flits are injected using existing injection control techniques to avoid saturation. If a multicast flit arrives from another router into a router where multicast flit support is disabled, the router cannot fork the multicast flit. Instead, only a single output port is allocated to the multicast flit,[5] and the multicast flit is delivered to each of its destinations sequentially (i.e., when the flit reaches a router that is attached to a destination node, the flit forks a response to the output port connected to the destination node, and continues onto its next destination).

4.2 Hotspot Flit Merging

When a source node sends a special response (e.g., an ACK for a coherence request, a synchronization request) to a certain destination node, there are likely other nodes sending an identical message to the same destination node. In Carpool, such special responses are sent using hotspot flits. A hotspot flit uses the same encoding format for its *source* nodes as a multicast flit does for its destination nodes (see Section 4.1). For source encoding, we call the m-bit list of nodes within a cluster the *srcList*.

[4]In our work, we empirically select the injection starvation rate threshold $\sigma = 0.00006$.
[5]When multicast flit support is disabled, the routing algorithm selects a port with the following priority: *E, W, N, S.*

An intermediate router in Carpool merges hotspot flits together if the flits have the same destination node, originate from the same network cluster, and contain the same payload. We number the input ports of each router (*N, E, S, W,* and *Local* in Carpool) in ascending order. When a hotspot flit (which we call Flit A) arrives at one of the input ports, the router checks any higher-numbered input ports to search if there are other hotspot flits that Flit A can merge with. If another hotspot flit with the same payload (which we call Flit B) is found, the router (1) updates the *srcList* of Flit A to include the list of sources from Flit B, and (2) drops Flit B.

In order to check whether two hotspot flits match, we need to add comparators into the router. Each comparator needs to check whether (1) the destinations match (6 bits in an 8×8 network), (2) the flits come from the same cluster (4 bits in our work), (3) the flits have matching flit sequence numbers (4 bits), and (4) the data matches (we restrict the data width of a hotspot flit to 48 bits, as the data typically represents a memory address). In all, each comparator needs to be 62 bits wide, and each router in Carpool requires 10 comparators, consuming 11.5% of the total area of the router (see Section 6 for our methodology and Section 7 for our analysis).

5 ROUTER MICROARCHITECTURE

In this section, we describe the underlying microarchitecture of each network router in Carpool. The router microarchitecture is based on the microarchitecture used in BLESS [33], to which we add support for multicast flit forking and hotspot flit merging.

5.1 Pipeline Design

Figure 4 shows the microarchitecture of a Carpool router. We organize the major functional blocks of the router into a two-stage pipeline. The first stage consists of three blocks: (1) *merge, eject, and inject* (MEI), (2) *route computation* (RC), and (3) *permutation sort* (PS). The second stage consists of three blocks: (1) *port allocation* (PA), (2) *switch traversal* (ST), and (3) *destination management* (DM). In addition to the two stages of the router pipeline, each flit must perform *link traversal* (LT).

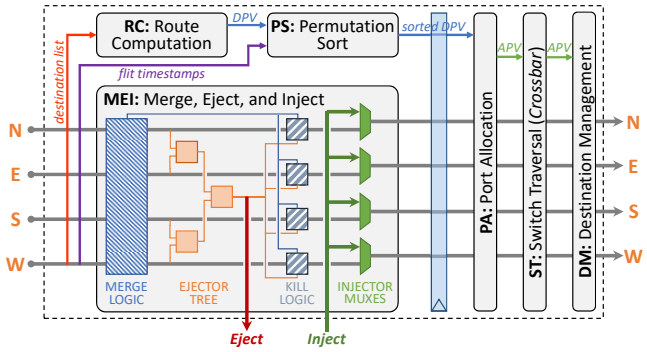


Figure 4: Carpool router microarchitecture.

Merge, Eject, and Inject (MEI). The router first accepts flits from the non-local input ports (i.e., *N, E, S, W*), assigning each flit to one of four *channels* within the router. The router checks the flits on these channels to see if any of them are hotspot flits that can be merged, and performs merging if possible (see Section 4.2). Next, the router looks to see if any of the flits are destined for the local node(s) attached to the router. The router ejects at most one of the flits with a local destination; any other flits with local destinations are deflected in the second pipeline stage. When a flit is ejected, the local node network interface uses miss status holding registers (MSHRs) to reassemble the packet [3, 4, 19, 20, 33].

After merging and ejection are complete, the router accepts at most one flit for injection by the local node(s) into the network. Injection can take place only if there is at least one free channel

within the router. By performing merging and ejection before injection, we maximize the probability that one of the channels is free when injection needs to be performed, alleviating starvation at the node network interface and enabling higher network throughput.

Carpool handles ejection to and injection from the local ports separate from the non-local ports, which are handled during ST using a crossbar. This is because the crossbar complexity increases quadratically with the number of ports [20, 51]. Thus, in order to reduce the complexity of the crossbar switch, we perform ejection and injection for the local ports earlier in the pipeline, similar to techniques adopted by other router designs [20, 51].

Route Computation (RC). In parallel with MEI, the router performs route computation and permutation sort. Route computation determines the *desired* output ports (i.e., the productive ports) of each flit, based on the destination(s) listed within the flit header. Carpool uses X-Y routing [14] to determine the desired output ports for each unicast or hotspot flit. The desired output ports are marked in a 4-bit *desired port vector* (DPV).

To compute the DPV of a multicast flit, we first partition the network into four quadrants (NE, SE, SW, and NW), which are mapped to the four output ports of the router (*N*, *E*, *S*, and *W*, respectively). For each quadrant and each cluster, the router contains a bitmask ($MASK_{c,q}$), which indicates all of the nodes assigned to the quadrant q for cluster c. The router uses *dstList* and $MASK_{c,q}$ to compute a DPV for the multicast flit that contains all of the output ports for which at least one of the flit's destination nodes is assigned. $MASK_{c,q}$ depends solely on the network topology, and thus needs to be computed only once, at network configuration time. Note that performing reconfiguration at runtime based on link availability improves the fault tolerance of the network [22], but this is beyond the scope of our work.

RC uses fixed port priorities to assign a flit to an output port, based on which quadrant(s) contain destination nodes of the flit. In a buffered on-chip network, routing flits with fixed port priorities causes unbalanced link utilization, which further increases latency and degrades throughput [31]. However, in a bufferless on-chip network, deflection routing provides a source of randomization, which improves the load balance within the network. Therefore, routing multicast flits with a fixed port priority in Carpool avoids load imbalance while benefiting from low design complexity.

Permutation Sort (PS). Once RC is complete, it generates a list that contains the 2-bit channel ID of each flit and the 4-bit DPV for the channel. In the permutation sort block, the router uses this list to rank the flits. Prior work [20, 33, 51] often rearranges the mapping of flits to channels. Rearranging the flit-to-channel mapping can require high complexity due to the number of bits of data that need to be moved. Instead, Carpool sorts the list of DPVs, using the age of each flit (based on the flit timestamp), and uses the DPV ordering to enforce priority in the second pipeline stage. To reduce the complexity of the sorting hardware, Carpool employs a *partial* bitonic sort algorithm using a two-stage permutation network [20, 51], which identifies the highest-priority flit and only partially sorts the remaining flits [51].

Port Allocation (PA). At the beginning of the second router pipeline stage, the router uses the sorted list of desired port vectors (DPVs) from PS to allocate an output port to each flit. We first describe a naive, sequential approach to port allocation, as shown in Figure 5a. We improve upon this with our *parallel port allocation* mechanism, shown in Figure 5b, which we discuss in Section 5.2.

In *sequential port allocation* (SPA), the output ports are assigned sequentially to each flit. SPA starts with the highest-priority flit (Flit 0 in Figure 5a), as ranked by PS, and advances to the next highest-priority flit (Flit 1) only after all ports are allocated to the highest-priority flit. For each flit, SPA uses the DPV to determine

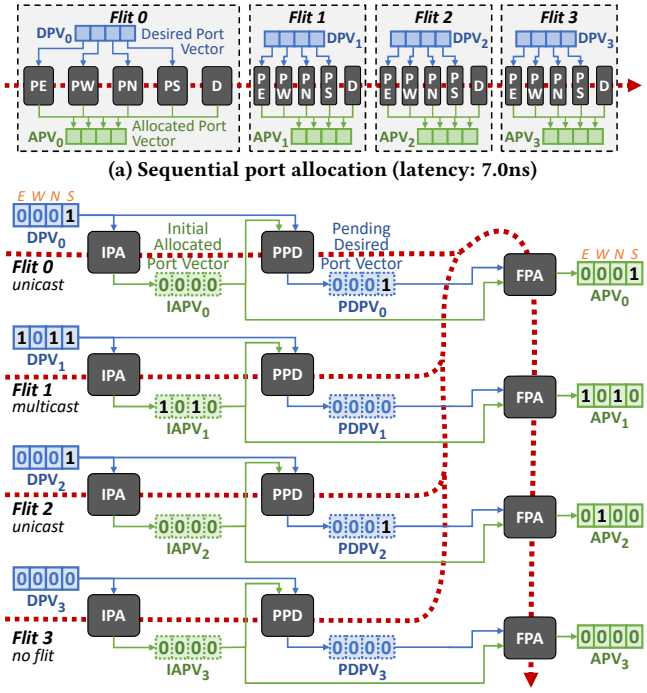


Figure 5: Two implementations for port allocation (PA) in Carpool. Critical path for each PA implementation shown with a dotted red line.

if the productive port(s) requested by the flit are available. SPA checks the ports one at a time. In the block labeled *PE*, SPA checks if (1) the flit requested Port *E* and (2) Port *E* has not yet been allocated to an earlier flit, and then allocates the port to the flit if both conditions are true. SPA repeats this process for Port *W* (*PW*), Port *N* (*PN*), and Port *S* (*PS*). If the flit is a multicast flit, and more than one of its desired ports is available for allocation, the router forks the multicast flit by allocating multiple ports to the flit, as long as the number of ports allocated does *not* violate our constraint in Equation 1 to ensure deadlock-free operation. If no ports are allocated to the flit after the *PS* block completes, the flit needs to be deflected, and is assigned to the next available port in block *D*. For each flit, SPA generates an *allocated port vector* (APV), which lists the output ports assigned to the flit.

Switch Traversal (ST). Based on the APVs generated by PA, the router configures the crossbar switch to transfer a flit from any one of its input channels to one or more output ports. A multicast flit that has been allocated to more than one productive port is replicated in the crossbar by simply assigning one channel to multiple output ports. We use a conventional multiplexer-based crossbar switch, due to its straightforward design. A more advanced crossbar switch [31] can be adopted to further reduce the hardware cost and energy consumption beyond what we report.

Destination Management (DM). If a multicast flit is replicated during ST, the router must update the destination list of each replica to ensure that multiple copies of the same flit do not visit the same destination node. The router designates one copy of the multicast flit as the *original*, and all other copies as replicas. The router modifies the *dstList* (see Section 4.1) of each replica to contain only the destinations in the quadrant assigned to the replica's output port, by masking *dstList* with $MASK_{c,q}$. For the *original* flit, the router zeroes out the destination nodes in the *dstList* that have been assigned to replicas. The zeroing can be done by bitmasking *dstList*

with the inverse of the $MASK_{c,q}$ for the quadrants being visited by the replicas. Once DM is complete, the flits perform link traversal (LT) and go to the next router along their path.

5.2 Parallel Port Allocation

Sequential port allocation, as shown in Figure 5a, leads to a large latency in the router, for two reasons. First, in order to enforce a strict flit priority, a port request from a low-priority flit (as ranked by PS) cannot be processed *until* requests from *all* higher-priority flits are resolved. This sequential dependency between flits during port allocation has been identified in prior studies [20, 51]. Second, due to the existence of multicast flits, the output ports must be allocated one at a time to avoid deadlock, as explained in Section 4.1.

Using Cadence Encounter (see Section 6), we find that the latency of our round-robin implementation of the PA block using sequential PA is 7.0ns, and that this block falls along the critical path of the pipeline stage latency. As a result, it limits the router clock rate, which in turn limits the network throughput. Inspired by prior work [20, 51], we design a *parallel port allocation* (PPA) mechanism for Carpool that significantly reduces the latency of PA. The key idea of parallel port allocation is to perform multi-stage port allocation, where only *uncontended* ports are initially assigned to each flit (which can be performed on all flits in parallel), and then the remaining ports are assigned using a simple priority ordering. Our parallel port allocation requires three steps, as shown in Figure 5b: (1) *initial port allocation* (IPA), (2) *pending port determination* (PPD), and (3) *final port allocation* (FPA).

In IPA, the first step, the router allocates as many *uncontended* productive ports (i.e., ports for which only one flit is making a request) as possible to the flits. As is done with SPA, a multicast flit can be assigned to multiple uncontended ports, as long as the constraint for deadlock-free operation (Equation 1) is not violated. The uncontended port allocations are saved to an *initial allocated port vector* (IAPV). By using bitwise operations, PPA can perform initial port allocation on all flits in parallel.

In PPD, the second step, the router generates a *pending desired port vector* (PDPV) for each flit. If a flit was not assigned any ports during IPA, its PDPV is the same as its initial DPV. If a flit was granted at least one port during IPA, the router zeroes out its PDPV, preventing the flit from acquiring any more ports to ensure that the other flits have the chance to claim a productive port in the final step. For a multicast flit, this means that it can fork replicas only during IPA, when the replica can be assigned to an uncontended output port. Like IPA, PPD can be performed on all flits in parallel.

In FPA, the final step, the router uses PDPV to allocate the remaining output ports, where a single port is allocated to any flit that was not allocated a port during IPA. PPA performs final port allocation one flit at a time, in order of flit priority (as ranked by PS). For each flit, a pending desired port (as listed in PDPV) is allocated to the flit if (1) the port has not been allocated already and (2) higher-ranked flits are not deflected. If either condition is false, the router deflects the flit. Deflected flits are assigned to the first unallocated port in the following order: *N*, *E*, *S*, *W*. We implement port allocation in FPA using a Boolean logic function, which takes in PDPV, a bit vector of unallocated ports, and a one-bit status indicating if a higher-ranked flit was deflected, and outputs the port to assign to the current flit. The port assignments made during FPA are combined with the IAPV list from IPA to generate the final port assignments for each flit.

Example Walk-through. We walk through an example allocation under PPA, as shown in Figure 5b. In this example, PPA needs to allocate output ports to three flits, where Flits 0 and 2 are unicast flits and Flit 1 is a multicast flit, and has received the desired port vectors (DPVs) from PS (see Section 5.1). There is no Flit 3 (which can happen when only three flits arrive at the router, or when a flit is merged or killed during MEI; see Section 5.1), so DPV_3 is set to zero. A lower-numbered flit has higher priority, as determined by PS. IPA uses the DPVs to determine that Ports *E* and *N* are uncontended, as they are requested only by Flit 1. Therefore, IPA sets $IAPV_1$ (i.e., the initial allocated port vector for Flit 1) to 1010, assigning the two requested ports to Flit 1. All three of the flits contend for Port *S*, so the port is not allocated by IPA.

During PPD, since Flit 1 already received a port, its pending desired port vector ($PDPV_1$) is zeroed out, to prevent it from receiving any more ports. This means that Flit 1 cannot acquire access to Port *S* as it had initially requested, and thus will need to be replicated to the nodes in the SW quadrant (see RC in Section 5.1) by a subsequent router. DM, which takes place after port allocation, ensures that the *dstList* of Flit 1 still contains the destination nodes in the SW quadrant. $PDPV_0$ and $PDPV_2$ are the same as DPV_0 and DPV_2, since Flit 0 and Flit 2 were not assigned any ports by IPA.

Finally, FPA takes place, starting with Flit 0 (the highest-ranked flit). Since $PDPV_0$ contains a request for Port *S*, which has not yet been allocated, FPA assigns the port to Flit 0, setting its allocated port vector (APV) to 0001. FPA skips Flit 1, since $PDPV_1$ is zero, indicating that the flit does not need any more ports. Instead, FPA just copies $IAPV_1$ to APV_1, assigning the flit to the uncontended ports it was allocated during IPA. For Flit 2, FPA cannot allocate Port *S* to Flit 2, since it was already allocated to Flit 0. As a result, FPA deflects Flit 2, assigning it to the last unallocated port (Port *W*). Since there is no Flit 3, APV_3 is set to zero, and FPA finishes.

Latency. We synthesize the RTL for our parallel port allocation using Cadence Encounter [6] with the FreePDK 35nm standard library [38]. We find that the latency of parallel port allocation is 3.2ns, which is 54.3% less than the latency of sequential port allocation (see PA in Section 5.1). As a result of the reduced latency, the critical path latency of the router pipeline shifts from the second pipeline stage to the first pipeline stage, resulting in a 24.1% decrease in the router clock cycle latency (see Section 7.5).

6 METHODOLOGY

We evaluate Carpool using a modified version of NOCulator [3, 4, 22, 35], an open-source cycle-accurate network-on-chip simulator. We perform our evaluations on a CMP that uses an 8×8 mesh on-chip network. Each network hop takes three cycles: two cycles in the router, and one cycle for link traversal. The network is clocked at the maximum possible frequency (i.e., we use the router critical path latency of each network in Section 7.6 as the clock period).

For a multicast packet, our simulator randomly selects a value *m*, and randomly chooses *m* network nodes (not including the source node) as the destinations for the packet. For hotspot packets, the simulator randomly chooses a common destination node, selects a random value *h*, and randomly selects *h* source nodes to generate a hotspot packet that targets the common destination node. As Carpool utilizes half of the payload field to carry the 64-bit destination or source list (*dstList*/*srcList*), each multicast and hotspot packet requires two flits. Each unicast packet requires only one flit.

We evaluate the network using synthetic workloads that replicate the packet injection rate, multicast, and hotspot behavior observed in real systems running multithreaded applications [18, 31]. We sweep the injection rate at which each node can generate a network packet, from an injection rate of 0.02 packets/cycle/node up to the rate at which the network reaches saturation. This generated packet has a set probability (*mc-rate*) of being a multicast packet and a set probability (*hs-rate*) of being a hotspot packet, chosen with a uniform random distribution. All other packets injected into the network are unicast packets. Prior work [18, 31] has demonstrated that for a real system running multithreaded applications, both *mc-rate* and *hs-rate* are typically between 0.01 and 0.10. As a result, we perform our injection rate sweeps using nine different combinations

of *mc-rate* and *hs-rate*, where *mc-rate* and *hs-rate* can be set to 0.01 (*low*), 0.05 (*medium*) or 0.1 (*high*). For brevity, we show latency, throughput, and power results for when *mc-rate* and *hs-rate* are both low (which we call *LowMC–LowHS*), and when they are both high (which we call *HighMC–HighHS*). We find that for the other seven combinations that we do not show, we observe the same trends that we see for the two combinations shown. Each of our experiments runs until the network retires 10 million packets.

In order to quantify the area and latency of the three evaluated networks, we implement each network in Verilog, and synthesize the RTL of each network using Cadence Encounter [6] with the FreePDK 35nm standard cell library [38] (where we set V_{dd}=1.1V and assume a temperature of 25°C). We report the critical path timing and area of each design in Section 7.6. The static and dynamic power consumption of each major network component is faithfully obtained from RTL synthesis. The link power is estimated using DSENT [45] assuming that the wire length between adjacent nodes is 2.5mm, and that the wire is driven at the same clock frequency as the network router. As DSENT does not provide a library for the 35nm process technology that we use for RTL synthesis, we select the 32nm library in DSENT to estimate the link power.

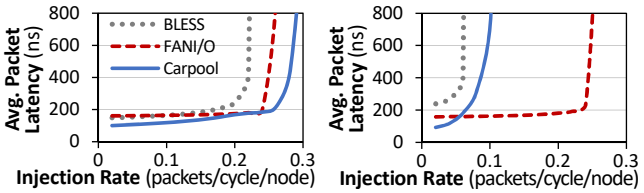
7 EVALUATION

We compare Carpool with (1) BLESS [33], a baseline bufferless on-chip network *without* support for multicast flit forking or hotspot flit merging, and (2) a modified version of FANIN/FANOUT [31] (which we abbreviate as FANI/O). As originally proposed, FANI/O is a routing algorithm for buffered networks that supports flit broadcasting (a one-to-all flow) and hotspot flit merging when *all* nodes are sending a message to a common destination node (an all-to-one flow). We modify FANI/O to support multicasting and many-to-one hotspot merging, by adding a destination list or a source list, respectively, to the payload of each flit using the same encoding as Carpool (see Section 4.1). For FANI/O, we assume that each physical port has four virtual channels, each of which has a single buffer for a flit and independent flow control.

7.1 Latency

Figure 6a shows the average packet latency as we vary the injection rate when *mc-rate* and *hs-rate* are both low (*LowMC–LowHS*). We make three key observations from the figure. First, Carpool consistently outperforms BLESS, reducing the average packet latency by 28.9% prior to saturation (injection rate < 0.24 packets/cycle/node). In Carpool, each multicast packet requires only two cycles to inject two multicast flits from the source node, whereas BLESS requires m (i.e., the number of destinations) cycles to inject m separate unicast flits. Carpool further reduces network contention over BLESS by opportunistically merging hotspot flits. Even when there are not many multicast and hotspot packets in the network, this flit consolidation in Carpool greatly improves network performance, due to the high penalty incurred from transporting multiple unicast flits for multicast or hotspot requests in BLESS. Second, even though it lacks buffers, Carpool consistently outperforms FANI/O. FANI/O buffers each request until a productive port for the request becomes available, using a fixed routing protocol. As FANI/O does *not* employ deflection, it is unable to exploit path diversity within the on-chip network. As a result, FANI/O has a 22.3% higher average latency than Carpool. Third, we observe that Carpool reaches saturation at a higher injection rate (0.30 packets/cycle/node) than both BLESS (0.24) and FANI/O (0.26).

Figure 6b shows the average packet latency as we vary the injection rate when *mc-rate* and *hs-rate* are both high (*HighMC–HighHS*). We make two key observations from the figure. First, the average packet latency is 57.3% lower in Carpool than the latency in BLESS, prior to saturation (injection rate < 0.06 packets/cycle/node), as Carpool enjoys even further benefits from multicast flit forking



(a) *LowMC–LowHS* (0.01) **(b) *HighMC–HighHS* (0.10)**

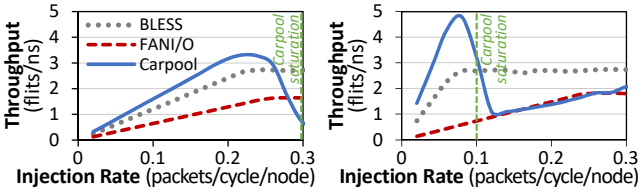
Figure 6: Average packet latency vs. injection rate.

and hotspot flit merging when *mc-rate* and *hs-rate* increase. Second, Carpool has 34.5% lower latency than FANI/O when the injection rate is low (≤ 0.06), but approaches saturation soon after (at an injection rate of 0.10 packets/cycle/node). One issue with multicast flits is that towards the end of a multicast flit's route, the flit forks into several replicas, eventually creating m copies for its m destination nodes. While Carpool improves over BLESS because the m copies do not exist for much of the flit's route, the network can still incur heavy congestion close to the destination nodes. Likewise, hotspot flits incur heavy network congestion when they are first generated. With its input buffers, complex flow control logic, and arbitration across virtual channels, FANI/O is more successful at handling the heavy congestion close to the multicast flit destinations and hotspot flit sources. However, to achieve this, FANI/O consumes 2.7× more area (see Section 7.6). With the same area budget, we can fragment Carpool into multiple sub-networks to enhance its performance, as prior work has done for bufferless networks [51]. After fragmenting, each sub-network can run below saturation to deliver lower latency and larger capacity (i.e., the overall network has a higher saturation point). We leave such fragmentation for future work.

Across all nine combinations of low/middle/high *mc-rate* and *hs-rate* values (not shown), Carpool reduces the average packet latency by 43.1% over BLESS, and by 26.4% over FANI/O. We conclude that Carpool successfully improves the performance of bufferless on-chip networks, with a low area overhead.

7.2 Throughput

Figures 7a and 7b show the throughput (expressed as flits completed per ns) of the three on-chip networks we evaluate for *LowMC–LowHS* and *HighMC–HighHS*, respectively. We make two key observations from these two figures.



(a) *LowMC–LowHS* (0.01) **(b) *HighMC–HighHS* (0.10)**

Figure 7: Throughput vs. injection rate.

First, compared with BLESS, Carpool has 31.5% and 85.3% higher throughput for *LowMC–LowHS* and *HighMC–HighHS*, respectively, prior to saturation. Carpool increases network throughput by waiting to fork a multicast flit until it reaches an intermediate node, and by merging hotspot flits. Near saturation, Carpool experiences a sudden drop in throughput because the packet deflection rate increases. When a multicast flit is deflected, multiple nodes must wait longer to receive the flit, causing high throughput loss.

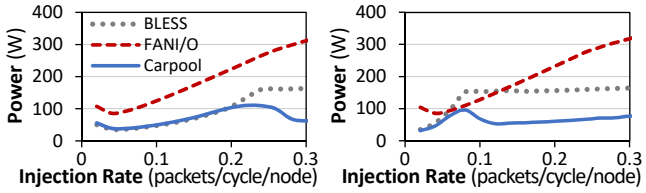
Second, compared with FANI/O, Carpool has 1.5× and 8.5× higher throughput for *LowMC–LowHS* and *HighMC–HighHS*, respectively, prior to saturation. In FANI/O, multicast flits fail to exploit the path diversity of the network, leaving many nearby links unused while the flits wait at the input buffer for a productive port to

become available. Because the flits wait at the buffer, the credit turnaround time increases, which in turn decreases the throughput.

We conclude that by reducing multicast/hotspot flit congestion and exploiting path diversity better through multicast flit forking, Carpool provides better throughput than BLESS and FANI/O.

7.3 Power Consumption

Figures 8a and 8b show the power consumption of the on-chip networks we evaluate for *LowMC–LowHS* and *HighMC–HighHS*, respectively. We make three key observations from these figures. First, compared to BLESS, Carpool has similar power consumption for *LowMC–LowHS* prior to saturation, despite requiring a greater circuit area to support multicast flit forking and hotspot flit merging (see Section 7.6). Second, for *HighMC–HighHS*, Carpool consumes 16.3% less power than BLESS because Carpool injects fewer packets into the network and delivers each packet faster, reducing the overall activity and latency of the network. Third, Carpool uses much less power than FANI/O, reducing the average power consumption prior to saturation by 56.7%/44.3% for *LowMC–LowHS*/*HighMC–HighHS*. These power savings are the result of the reduced circuit complexity (e.g., lack of buffers) of Carpool over FANI/O.



(a) *LowMC–LowHS* (0.01) (b) *HighMC–HighHS* (0.10)

Figure 8: Power consumption vs. injection rate.

Across all nine combinations of low/middle/high *mc-rate* and *hs-rate* values (not shown), Carpool reduces the average on-chip network power consumption by 8.3% over BLESS, and by 50.5% over FANI/O. We conclude that Carpool is an effective solution to provide a low-power on-chip network under a wide and diverse range of network traffic patterns.

7.4 Performance Breakdown

To understand the individual benefits of enabling multicast flit forking and hotspot flit merging separately, we compare the performance of Carpool to two bufferless on-chip networks: *ForkOnly*, which enables only the multicast flit forking capability of Carpool; and *MergeOnly*, which enables only the hotspot flit merging capability of Carpool. Figure 9 shows the normalized latency, throughput, and deflection rate of these networks, normalized to Carpool for each metric, for *HighMC–HighHS*. We make two observations from this figure. First, for both ForkOnly and MergeOnly, the flit deflection rate increases significantly, resulting in latency increases of 1.9× and 2.6×, respectively, over Carpool. Second, ForkOnly retains most of the throughput of Carpool, while MergeOnly has much lower throughput, allowing ForkOnly to reduce network congestion (and thus, packet latency) despite having a higher deflection rate.

We also study the impact of employing adaptive multicast flit forking (see Section 4.1). Figure 10 shows the average packet latency of Carpool without and with adaptive forking, for medium values (0.05) of *mc-rate* and *hs-rate* (which we call *MedMC–MedHS*). We observe that both networks have similar latency at low injection rates, when the network load is light. When the injection rate exceeds 0.10, Carpool without adaptive forking leads to heavy network congestion, causing the network to saturate at an injection rate of 0.12. Adaptive forking is successful at reducing congestion when the injection rate is higher, thus allowing the network to sustain a much higher injection rate before it reaches saturation.

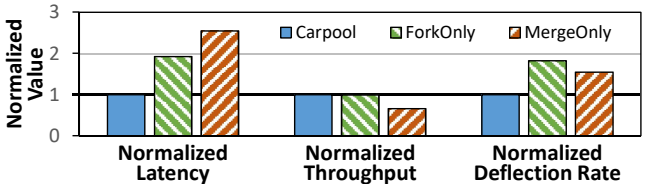


Figure 9: Normalized latency, throughput, and deflection rate of Carpool, ForkOnly, and MergeOnly networks, normalized to Carpool, for *HighMC–HighHS*.

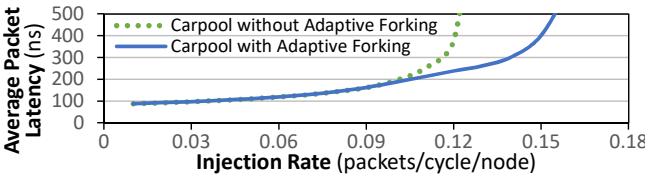


Figure 10: Effect of adaptive forking on average packet latency, for *MedMC–MedHS*.

We conclude that multicast flit forking, hotspot flit merging, and adaptive forking are all essential to enabling the high performance provided by Carpool.

7.5 Effect of Parallel Port Allocation

To study the effect of parallel port allocation (Section 5.2), we compare Carpool to a version of Carpool that uses sequential port allocation (called *CarpoolSPA*) for *MedMC–MedHS*, as shown in Figure 11. For unicast traffic, prior work found that a relaxed priority ordering for port allocation can increase the deflection rate and latency slightly [51]. We observe that due to parallel port allocation, both the deflection rate and the latency are lower for Carpool than for CarpoolSPA. In CarpoolSPA, when a multicast flit has a higher priority, the flit is replicated to *all* of its desired output ports (provided that the constraint in Equation 1 is not violated), even when lower-priority flits are contending for those output ports. As a result, a lower-priority flit is more likely to be deflected, increasing network contention. In contrast, parallel port allocation replicates a multicast flit only when its desired output ports are *uncontended*, thus ensuring that the replicas do not inadvertently deflect lower-priority flits. As shown in Figure 11, Carpool has 14% fewer forking operations than CarpoolSPA. In addition to the reduced forking and deflection, parallel port allocation can shorten the latency of port allocation by 54.3% over sequential port allocation (see Section 5.2). We conclude that by better managing multicast fork replication, parallel port allocation provides significant performance benefit over sequential port allocation in Carpool.

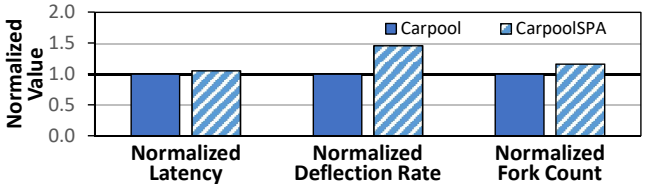


Figure 11: Effect of parallel port allocation.

7.6 Hardware Complexity

Using Cadence Encounter [6] to perform RTL synthesis, we obtain the critical path latency and area of a router in Carpool, CarpoolSPA, BLESS, FANI/O (where each of the four VCs can buffer a single flit), and FANI/O-4 (a variant of FANI/O where each of the four VCs can buffer *four* flits), as shown in Table 1. Compared with CarpoolSPA, Carpool reduces the router critical path latency by 24.1% due to parallel port allocation, at the expense of only 2.4%

more router area. Although a router in Carpool consumes 19.2% more area than in BLESS, its critical path latency is 5.7% lower, and Carpool significantly reduces the average packet latency and power consumption over BLESS. Carpool reduces the router critical path latency by 34.7% (or 35.9%), and the router area by 63.5% (or 77.9%), over FANI/O (or FANI/O-4). The small area and fast timing of Carpool over FANI/O are mainly due to removed buffers and simplified control. Despite the much smaller area, Carpool performs competitively with FANI/O, and even outperforms it when *mc-rate* and *hs-rate* are not very high (see Figures 6a and 7a). We conclude that Carpool delivers significant network performance improvement with low hardware complexity.

Table 1: Hardware cost comparison of a single router. Numbers in parentheses show cost normalized to Carpool.

	Carpool	CarpoolSPA	BLESS	FANI/O	FANI/O-4
Critical Path Latency (ns)	6.6 (1.00×)	8.7 (1.32×)	7.0 (1.06×)	10.1 (1.53×)	10.3 (1.56×)
Area (μm^2)	2,746,396 (1.00×)	2,683,216 (0.98×)	2,304,816 (0.84×)	7,515,436 (2.74×)	12,401,068 (4.52×)

8 CONCLUSION

Chip multiprocessors (CMPs) generate a significant amount of on-chip network requests that have either a one-to-many (i.e., multicast) or many-to-one (i.e., hotspot) flow. As the number of cores within a CMP increases, one-to-many and many-to-one flows result in greater network congestion. Bufferless on-chip networks, whose lower hardware complexity is a good fit for CMPs as the core count increases, do not currently provide efficient hardware support for these one-to-many or many-to-one flows. In this work, we propose *Carpool*, the first bufferless on-chip network optimized for one-to-many and many-to-one traffic. The key ideas of Carpool are to (1) *adaptively fork* flit replicas for *multicast traffic* and (2) *merge hotspot flits*, at intermediate routers within the network, both of which reduce network contention and improves throughput, and to (3) perform *parallel port allocation* to reduce router latency. We conclude that with the multicast flit forking and hotspot flit merging support we introduce, we can improve the latency, throughput, and power consumption of bufferless on-chip networks without requiring significant additional circuit area.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful comments and suggestions. This research was partially supported by the NSF (grants 1212962, 1423302, 1527051, and 1527318).

REFERENCES

[1] P. Abad *et al.*, "Rotary Router: An Efficient Architecture for CMP Interconnection Networks," in *ISCA*, 2007.
[2] P. Abad *et al.*, "MRR: Enabling Fully Adaptive Multicast Routing for CMP Interconnect Networks," in *HPCA*, 2009.
[3] R. Ausavarungnirun *et al.*, "Design and Evaluation of Hierarchical Rings with Deflection Routing," in *SBAC-PAD*, 2014.
[4] R. Ausavarungnirun *et al.*, "A Case for Hierarchical Rings with Deflection Routing: An Energy-Efficient On-Chip Communication Substrate," *PARCO*, 2016.
[5] P. Baran, "On Distributed Communications Networks," *TCOM*, 1964.
[6] Cadence Design Systems, Inc., "SOC Encounter User Guide," http://www.cadence.com/products/di/firstencounter/.
[7] Y. Cai *et al.*, "Comparative Evaluation of FPGA and ASIC Implementations of Bufferless and Buffered Routing Algorithms for On-Chip Networks," in *ISQED*, 2015.
[8] J. G. Castanos *et al.*, "Evaluation of a Multithreaded Architecture for Cellular Computing," in *HPCA*, 2002.
[9] K. K. Chang *et al.*, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks," in *SBAC-PAD*, 2012.
[10] C.-M. Chiang and L. M. Ni, "Multi-Address Encoding for Multicast," in *PCRCW*, 1994.
[11] C. Craik and O. Mutlu, "Investigating the Viability of Bufferless NoCs in Modern Chip Multi-Processor Systems," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. 2011-004, 2011.
[12] W. J. Dally and H. Aoki, "Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels," in *TPDS*, 2005.
[13] W. J. Dally and C. L. Seitz, "The Torus Routing Chip," *Distributed Computing*, 1986.
[14] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2003.
[15] B. K. Daya *et al.*, "Quest for High-Performance Bufferless NoCs with Single-Cycle Express Paths and Self-Learning Throttling," in *DAC*, 2016.
[16] E. Ebrahimi *et al.*, "Prefetch-Aware Shared Resource Management for Multi-Core Systems," in *ISCA*, 2011.
[17] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.
[18] N. Enright Jerger *et al.*, "Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Support," in *ISCA*, 2008.
[19] C. Fallin *et al.*, "MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect," in *NOCS*, 2012.
[20] C. Fallin *et al.*, "CHIPPER: A Low-Complexity Bufferless Deflection Router," in *HPCA*, 2011.
[21] C. Fallin *et al.*, "Bufferless and Minimally-Buffered Deflection Routing," in *Routing Algorithms in Networks-on-Chip*. Springer, 2014.
[22] M. Fattah *et al.*, "A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips," in *NOCS*, 2015.
[23] K. Goossens *et al.*, "AEthereal Network on Chip: Concepts, Architectures, and Implementations," in *IEEE D&T*, 2005.
[24] P. Gratz *et al.*, "Regional Congestion Awareness for Load Balance in Networks-on-Chip," in *HPCA*, 2008.
[25] M. Hayenga *et al.*, "SCARAB: A Single Cycle Adaptive Routing and Bufferless Network," in *MICRO*, 2009.
[26] Y. Jin *et al.*, "A Domain-Specific On-Chip Network Design for Large Scale Cache Systems," in *HPCA*, 2007.
[27] P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, 1979.
[28] H. Kim *et al.*, "Clumsy Flow Control for High-Throughput Bufferless On-Chip Networks," in *CAL*, 2013.
[29] H. Kim *et al.*, "Extending Bufferless On-Chip Networks to High-Throughput Workloads," in *NOCS*, 2014.
[30] J. Kim *et al.*, "A Low Latency Router Supporting Adaptivity for On-Chip Interconnects," in *DAC*, 2005.
[31] T. Krishna *et al.*, "Towards the Ideal On-Chip Fabric for 1-to-Many and Many-to-1 Communication," in *MICRO*, 2011.
[32] Z. Lu *et al.*, "Connection-Oriented Multicasting in Wormhole-Switched Networks-on-Chip," in *ISVLSI*, 2006.
[33] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-Chip Networks," in *ISCA*, 2009.
[34] C. A. Nicopoulos *et al.*, "ViChaR: A Dynamic Virtual Channel Regulator for Network-on-Chip Routers," in *MICRO*, 2006.
[35] "NOCulator," https://github.com/CMU-SAFARI/NOCulator/.
[36] G. Nychis *et al.*, "On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects," in *SIGCOMM*, 2012.
[37] G. Nychis *et al.*, "Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?" in *HotNets*, 2010.
[38] Oklahoma State Univ., "FreePDK Standard Cell Libraries," https://vlsiarch.ecen.okstate.edu/flows/.
[39] A. Olofsson *et al.*, "A 1024-Core 70 GFLOP/W Floating Point Manycore Microprocessor," in *HPEC*, 2011.
[40] S. Rodrigo *et al.*, "Efficient Unicast and Multicast Support for CMPs," in *MICRO*, 2008.
[41] F. A. Samman *et al.*, "Multicast Pipeline Router Architecture for Network-on-Chip," in *DATE*, 2008.
[42] A. Singh *et al.*, "GOAL: A Load-Balanced Adaptive Routing Algorithm for Torus Networks," in *ISCA*, 2003.
[43] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *SPIE*, 1982.
[44] C. B. Stunkel *et al.*, "A New Switch Chip for IBM RS/6000 SP Systems," in *SC*, 1999.
[45] C. Sun *et al.*, "DSENT – A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *NOCS*, 2012.
[46] M. Thottethodi *et al.*, "Self-Tuned Congestion Control for Multiprocessor Networks," in *HPCA*, 2001.
[47] J. S. Turner, "An Optimal Nonblocking Multicast Virtual Circuit Switch," in *INFOCOM*, 1994.
[48] L. Wang *et al.*, "Recursive Partitioning Multicast: A Bandwidth-Efficient Routing for Networks-on-Chip," in *NOCS*, 2009.
[49] M. A. Watkins and D. H. Albonesi, "ReMAP: A Reconfigurable Heterogeneous Multicore Architecture," in *MICRO*, 2010.
[50] X. Xiang *et al.*, "A Model for Application Slowdown Estimation in On-Chip Networks and Its Use for Improving System Fairness and Performance," in *ICCD*, 2016.
[51] X. Xiang and N.-F. Tzeng, "Deflection Containment for Bufferless Network-on-Chips," in *IPDPS*, 2016.
[52] P.-C. Yew *et al.*, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," in *TC*, 1987.