CHIPPER: A Low-complexity Bufferless Deflection Router

Chris Fallin cfallin@cmu.edu
Chris Craik craik@cmu.edu
Onur Mutlu onur@cmu.edu

Computer Architecture Lab (CALCM)
Carnegie Mellon University

SAFARI Technical Report No. 2010-001

December 29, 2010

Abstract

As Chip Multiprocessors (CMPs) scale to tens or hundreds of nodes, the interconnect becomes a significant factor in cost, energy consumption and performance. Recent work has explored many design tradeoffs for networks-on-chip (NoCs) with novel router architectures to reduce hardware cost. In particular, recent work proposes bufferless deflection routing to eliminate router buffers. The high cost of buffers makes this choice potentially appealing, especially for low-to-medium network loads.

However, current bufferless designs usually add complexity to control logic. Deflection routing introduces a sequential dependence in port allocation, yielding a slow critical path. Explicit mechanisms are required for livelock freedom due to the non-minimal nature of deflection. Finally, deflection routing can fragment packets, and the reassembly buffers require large worst-case sizing to avoid deadlock, due to the lack of network backpressure. The complexity that arises out of these three problems has discouraged practical adoption of bufferless routing.

To counter this, we propose *CHIPPER* (Cheap-Interconnect Partially Permuting Router), a simplified router microarchitecture that eliminates in-router buffers and the crossbar. We introduce three key insights: first, that deflection routing port allocation maps naturally to a permutation network within the router; second, that livelock freedom requires only an implicit token-passing scheme, eliminating expensive age-based priorities; and finally, that flow control can provide correctness in the absence of network backpressure, avoiding deadlock and allowing cache miss buffers (MSHRs) to be used as reassembly buffers. Using multiprogrammed SPEC CPU2006, server, and desktop application workloads and SPLASH-2 multithreaded workloads, we achieve an average 54.9% network power reduction for 13.6% average performance degradation (multiprogrammed) and 73.4% power reduction for 1.9% slowdown (multithreaded), with minimal degradation and large power savings at low-to-medium load. Finally, we show 36.2% router area reduction relative to buffered routing, with comparable timing.

This technical report is an extended version of our HPCA-17 paper [15]. We have included additional background in § 2.2.4 and and more detailed descriptions of the router microarchitecture in § 4.4.

1 Introduction

In recent years, NoCs have become a focus of intense interest in computer architecture. Moore's Law compels us toward larger multicore processors. As tiled CMPs [41, 4, 27, 2] are adopted, on-chip interconnect becomes critically important. Future tiled CMPs will likely contain hundreds of cores [42, 22, 6], and one current chip already contains 100 cores [51]. At this density, a commonly proposed on-chip interconnect is the 2D mesh: it maps naturally to the tiled CMP architecture [2] and allows for simple routing algorithms and low-radix router architectures.

Traditionally, interconnection network designs have been motivated by and tuned for large, high performance multiprocessors [30, 9]. As interconnects migrate to the on-chip environment, constraints and tradeoffs shift. Power, die area and design complexity become more important, and link latencies become smaller, making the effects of router latency more pronounced. As a consequence, any competitive router design should have a short critical path, and should simultaneously minimize logic and buffer footprint.

Low-cost NoC designs have thus become a strong focus. In particular, one line of recent work has investigated how to eliminate in-router buffers altogether [38, 19, 16], or minimize them with alternative designs [25, 26, 39]. The completely bufferless designs either drop [19, 16] or misroute (deflect) [38] flits when contention occurs. Eliminating buffers is desirable: buffers draw a significant fraction of NoC power [21] and area [17], and can increase router latency. Moscibroda and Mutlu [38] report 40% network energy reduction with minimal performance impact at low-to-medium network load. For a design point where interconnect is not highly utilized, bufferless routers can yield large savings.

Bufferless deflection routing thus appears to be promising. However, that work, and subsequent evaluations [36, 19], note several unaddressed problems and complexities that significantly discourage adoption of bufferless designs. First, a long critical path in *port allocation* arises because every flit must leave the router at the end of the pipeline, and because deflection is accomplished by considering flits sequentially. Second, livelock freedom requires a *priority scheme* that is often more complex than in buffered designs: for example, in Oldest-First arbitration, every packet carries a timestamp, and a router must sort flits by timestamps. Finally, packet fragmentation requires *reassembly buffers*, and without additional mechanisms, worst-case sizing is necessary to avoid deadlock [36].

In this paper, we propose a new bufferless router architecture, *CHIPPER*, that solves these problems through three key insights. First, we eliminate the expensive port allocator and the crossbar, and replace both with a *permutation network*; as we argue, deflection routing maps naturally to this arrangement, reducing critical path length and power/area cost. Second, we provide a strong livelock guarantee through an *implicit token passing* scheme, eliminating the cost of a traditional priority scheme. Finally, we propose a simple *flow control* mechanism for correctness with reasonable reassembly buffer sizes, and propose using *cache miss buffers (MSHRs [29]) as reassembly buffers*. We show that at low-to-medium load, our reduced-complexity design performs competitively to a traditional buffered router (in terms of application performance and operational frequency) with significantly reduced network power, and very close to baseline bufferless (BLESS [38]) with a reduced critical path.

Our contributions are:

- Cheap deflection routing by replacing the allocator and crossbar with a *partial permutation network*. This design parallelizes port allocation and reduces hardware cost significantly.
- A strong livelock guarantee through *implicit token passing*, called Golden Packet (GP). By replacing the Oldest-First scheme for livelock freedom [38], GP preserves livelock freedom while eliminating the need to assign and compare timestamps.
- A flow-control scheme, *Retransmit-Once*, that avoids worst-case reassembly buffer sizing otherwise necessary to avoid deadlock. Use of *MSHRs as reassembly buffers*, allowing packet fragmentation due to deflection routing without incurring additional buffering cost.
- Evaluation over multiprogrammed SPEC CPU2006 [49] and assorted desktop and server (web search, TPC-C) applications, and multithreaded SPLASH-2 [57] workloads, showing minimal performance degradation at low-to-medium network load with significant power, area and complexity savings.

2 Bufferless Deflection Routing

2.1 Why Bufferless? (and When?)

Bufferless[1] NoC design has recently been evaluated as an alternative to traditional virtual-channel buffered routers [38, 19, 16, 31, 52]. It is appealing mainly for two reasons: reduced hardware cost, and simplicity in design. As core count in modern CMPs continues to increase, the interconnect becomes a more significant component of system hardware cost. Several prototype manycore systems point toward this trend: in MIT RAW, interconnect consumes ~40% of system power; in the Intel Terascale chip, 30%. Buffers consume a significant portion of this power. A recent work [38] reduced network energy by 40% by eliminating buffers. Furthermore, the complexity reduction of the design at the high level could be substantial: a bufferless router requires only pipeline registers, a crossbar, and arbitration logic. This can translate into reduced system design and verification cost.

Bufferless NoCs present a tradeoff: by eliminating buffers, the peak network throughput is reduced, potentially degrading performance. However, network power is often significantly reduced. For this tradeoff to be effective, the power reduction must outweigh the slowdown's effect on total energy. Moscibroda and Mutlu [38] reported minimal

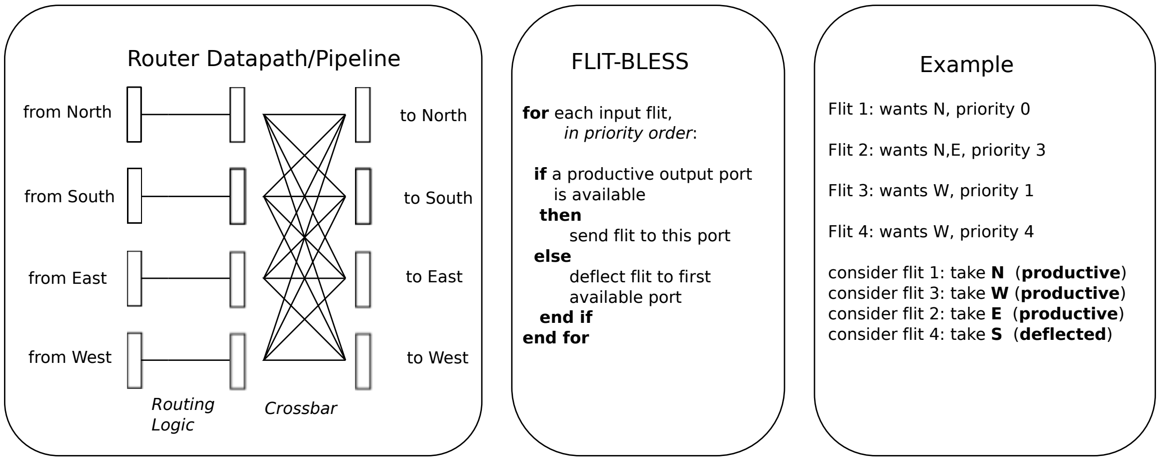
[1]More precisely, a "bufferless" NoC has no in-router (e.g., virtual channel) buffers, only pipeline latches. Baseline bufferless designs, such as BLESS [38], still require reassembly buffers and injection queues. As we describe in § 4.3, we eliminate these buffers as well.

performance reduction with bufferless when NoC is lightly loaded, which constitutes many of the applications they evaluated. Bufferless NoC design thus represents a compelling design point for many systems with low-to-medium network load, eliminating unnecessary capacity for significant savings.

2.2 BLESS: Baseline Bufferless Deflection Routing

Here we briefly introduce bufferless deflection routing in the context of BLESS [38]. BLESS routes flits, the minimal routable units of packets, between nodes in a mesh interconnect. Each flit in a packet contains header bits and can travel independently, although in the best case, all of a packet's flits remain contiguous in the network. Each node contains an injection buffer and a reassembly buffer; there are no buffers within the network, aside from the router pipeline itself. Every cycle, flits that arrive at the input ports contend for the output ports. When two flits contend for one output port, BLESS avoids the need to buffer by misrouting one flit to another port. The flits continue through the network until ejected at their destinations, possibly out of order, where they are reassembled into packets and delivered.

Figure 1 summarizes router operation in such a network, and Figure 2 shows an example of deflection routing in a 3x3-mesh network (a small configuration for the sake of the example). The example demonstrates deflection: when two flits contend for the same output link at a router, one of them takes a *non-productive* path, adding two cycles to its path. In contrast, a traditional buffered network would have buffered one of the flits at the center router $(1,1)$ for one cycle. Bufferless deflection routing introduces this *non-minimal routing* in order to eliminate such in-router buffers, so that when contention is rare, deflection will yield less penalty than the static and dynamic hardware costs of router buffers. In a sense, bufferless routers spread contention in *space* rather than in *time*, as buffered routers do.



FLIT-BLESS

```
for each input flit,
    in priority order:

  if a productive output port
     is available
   then
      send flit to this port
  else
      deflect flit to first
      available port
   end if
end for
```

Example

Flit 1: wants N, priority 0

Flit 2: wants N,E, priority 3

Flit 3: wants W, priority 1

Flit 4: wants W, priority 4

consider flit 1: take **N** (**productive**)
consider flit 3: take **W** (**productive**)
consider flit 2: take **E** (**productive**)
consider flit 4: take **S** (**deflected**)

Figure 1: Summary of bufferless deflection routing (BLESS) as posed in [38]. The baseline router has a 2-cycle latency, in which the routing algorithm (FLIT-BLESS shown here) arbitrates between flits. Injection and ejection are omitted for clarity.

Deflection routing is not new: it was first proposed in [3], and is used in optical networks because of the cost of optical buffering [8]. It works because a router has as many output links as input links (in a 2D mesh, 4 for neighbors and 1 for local access). Thus, the flits that arrive in a given cycle can always leave exactly N cycles later, for an N-stage router pipeline. If all flits request unique output links, then a deflection router can grant every flit's requested output. However, if more than one flit contends for the same output, all but one must be deflected to another output that is free.

2.2.1 Livelock Freedom

Whenever a flit is deflected, it moves further from its destination. If a flit is deflected continually, it may never reach its destination. Thus, a routing algorithm must explicitly avoid livelock. It is possible to probabilistically bound network latency in a deflection network [28, 7]. However, a deterministic bound is more desirable. BLESS [38] uses an *Oldest-First* prioritization rule to give a deterministic bound on network latency. Flits arbitrate based on packet timestamps. Prioritizing the oldest traffic creates a consistent total order and allows this traffic to make forward progress. Once

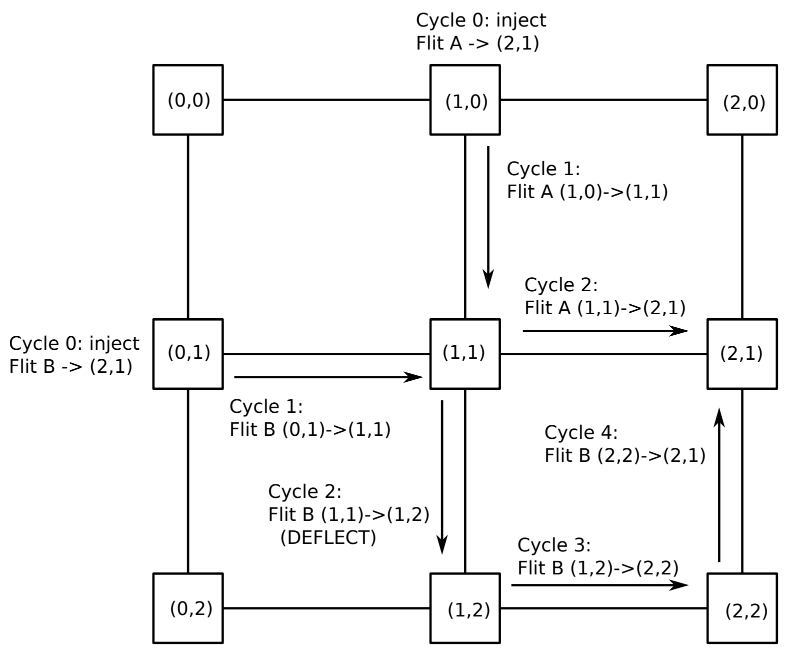


Figure 2: An example of bufferless deflection routing. Two nodes, $(0,1)$ and $(1,0)$, inject flits destined to $(2,1)$ in the same cycle 0. When the two flits arrive at $(1,1)$ in cycle 1, one wins arbitration for the link toward $(2,1)$, and the other is deflected.

the oldest packet arrives, another packet becomes oldest. Thus livelock freedom is guaranteed inductively. However, this age-based priority mechanism is expensive [36, 19] both in header information and in arbitration critical path. Alternatively, some bufferless routing proposals do not provide or explicitly show deterministic livelock-freedom guarantees [19, 16, 52]. This can lead to faster arbitration if it allows for simpler priority schemes. However, a provable guarantee of livelock freedom is necessary to show system correctness in all cases.

2.2.2 Injection

BLESS guarantees that all flits entering a router can leave it, because there are at least as many output links as input links. However, this does not guarantee that new traffic from the local node (e.g., core or shared cache) can always enter the network. A BLESS router can inject a flit whenever an input link is empty in a given cycle. In other words, BLESS requires a “free slot” in the network in order to insert new traffic. When a flit is injected, it contends for output ports with the other flits in that cycle. Note that the injection decision is purely local: that is, a router can decide whether to inject without coordinating with other routers.

2.2.3 Ejection and Packet Reassembly

A BLESS router can eject one flit per cycle when that flit reaches its destination. In any bufferless deflection network, flits can arrive in random order; therefore, a packet reassembly buffer is necessary. Once all flits in a packet arrive, the packet can be delivered to the local node. Importantly, this buffer must be managed so that it does not overflow, and in such a way that maintains correctness. The work in [38] does not consider this problem in detail. Instead, it assumes an infinite reassembly buffer, and reports maximum occupancies for the evaluated workloads. We will return to this point in § 3.3.

2.2.4 Microarchitecture

Baseline BLESS makes use of a two-cycle router pipeline, as described in [38]. The original proposal also describes how a 1-cycle router latency may be attained through lookahead traversal of flit headers.

However, as described, the pipeline combines route computation (determining productive directions) with arbitration (assigning output ports) in one cycle time. We find that this critical path is too long for a high-speed router.

For our BLESS hardware model used by our comparisons in § 5.7, we split route computation and arbitration into two separate cycles. We then make use of the lookahead link traversal technique described in [38] for flit headers to recover one cycle, yielding two cycles overall router latency. This pipeline arrangement is shown in Fig. 3.

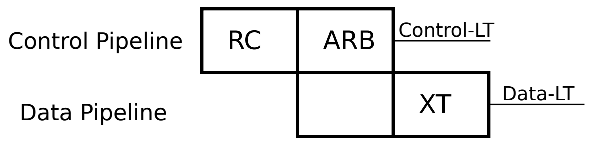


Figure 3: Pipeline structure for baseline BLESS [38]. In our hardware implementation for evaluation, we split routing logic into two stages, Route Compute (RC) and Arbitration (ARB), but make use of the flit-header lookahead traversal described in [38] to retain a two-cycle router latency.

While route computation is straightforward – a comparison of destination address to the current location, and a decision based on XY-routing [12] – the arbitration stage merits special attention. As depicted in Fig. 1, the FLIT-BLESS algorithm considers each flit in turn, from highest to lowest priority, to allocate output ports. Each flit then receives a port according to two rules: it obtains a productive port, if one is available. If not, it obtains any other port, in order to deflect.

Considering flits in priority order requires a sort function. This is best implemented in hardware as a sort network [5], in which each stage selectively performs a series of swaps based on pairwise comparisons. A three-stage network is sufficient to sort four flits by priority. Such a network is depicted in Fig. 4. This network implements bitonic sort: the arrow direction in each 2x2 module indicates the sort direction (increasing or decreasing). The module contains a comparator, and either passes or swaps its inputs to its outputs according to this sort direction.

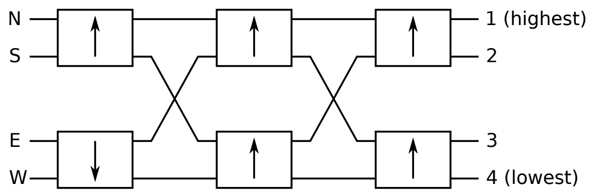


Figure 4: Bitonic sort network [5] for full priority-sort in a bufferless deflection arbiter.

Once flits are sorted, logic must allocate ports in priority order. The most straightforward implementation of this output port allocation consists of a series of allocator modules, one per flit. Each module receives the set of available (unallocated) ports from the previous stage as a bitvector, allocates one port for its associated flit, and then passes the remaining port information to the next stage. Such an arrangement is shown in Figure 5.

Note that it is possible to parallelize the sort network and port allocation logic by precomputing port allocations for all possible permutations of input flit priorities. The final port allocation decision is then determined by selecting the output of the appropriately permuted port allocator. However, this incurs significant expense in control logic area, because there are 24 (4!) possible permutations of flit priorities for the 4-input routing problem (in a 2D-mesh), requiring 24 separate port allocators. We thus do not consider it further in our baseline bufferless deflection routing model.

3 Deflection Routing Complexities

While bufferless deflection routing is conceptually and algorithmically simple, a straightforward hardware implementation leads to numerous complexities. In particular, two types of problem plague baseline bufferless deflection routing: high hardware cost, and unaddressed correctness issues. The hardware cost of a direct implementation of bufferless deflection routing is nontrivial, due to expensive control logic. Just as importantly, correctness issues arise in the reassembly buffers when they have practical (non-worst-case) sizes, and this fundamental problem is unaddressed by current work. Here, we describe the major difficulties: *output port allocation*, *expensive priority arbitration*, and *reassembly buffer cost and correctness*. Prior work cites these weaknesses [36, 19]. These problems directly lead to the three key insights in CHIPPER. We will describe in each in turn.

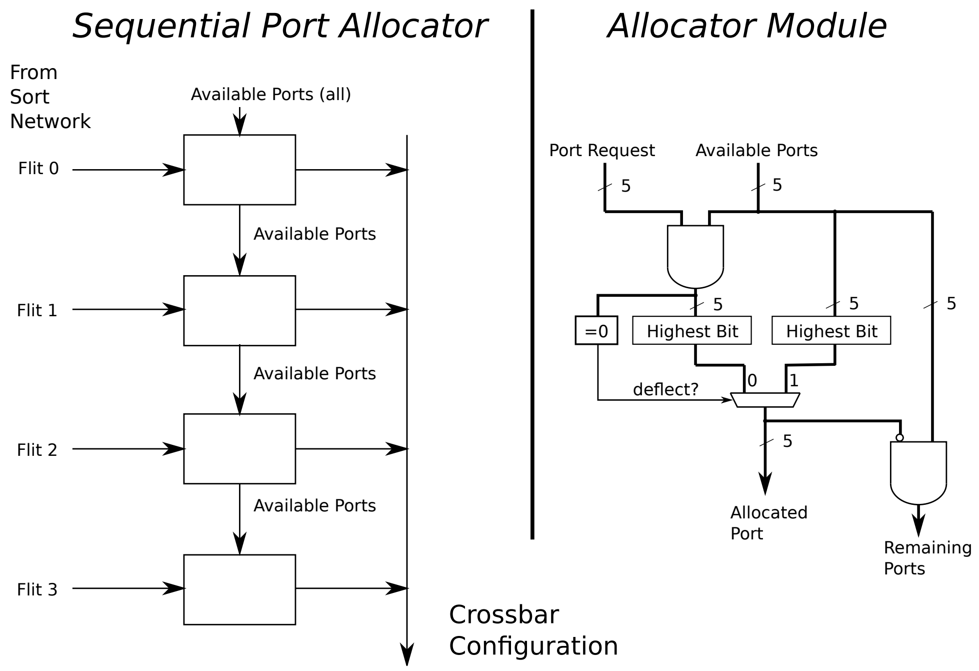
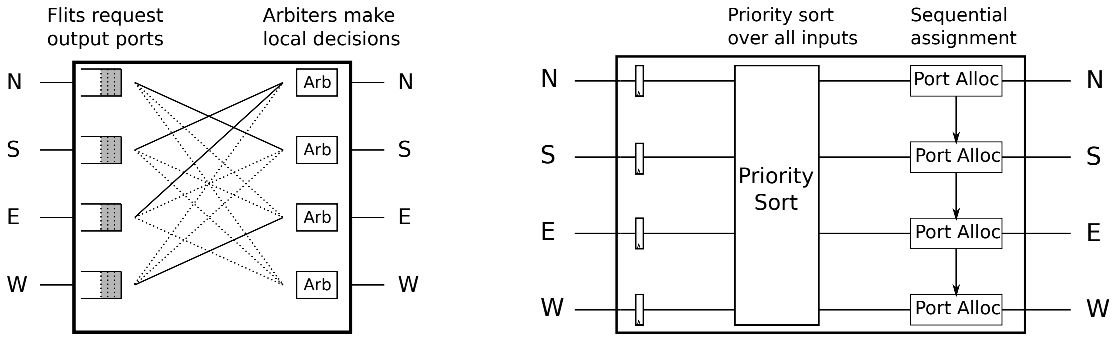


Figure 5: Bufferless deflection arbiter detail: sequential port allocation logic.



(a) Buffered router port allocator

(b) Bufferless deflection router port allocator

Figure 6: Port allocator structures: deflection routing requires more complex logic with a longer critical path.

3.1 Output Port Allocation

Deflection-routing consists of mapping a set of input flits, each with a preferred output and some priority, to outputs such that every flit obtains an output. This computation is fundamentally difficult for two reasons: *(i)* every non-ejected input flit must take some output, since flits are never buffered or dropped; and *(ii)* a higher-priority flit might take a lower-priority flit's preferred output, so the routing for a given flit involves an inherently sequential dependence on all higher-priority flits' routing decisions (as noted in [36] and [19]). In other words, the routing decision depends on the earlier port allocations; furthermore, the notion of "earlier" depends on the sorted ordering of the inputs. Thus, flits must be sorted by priority before port allocation. A carry-select-like parallel optimization [19] can reduce the critical path by precomputing deflections (e.g., for all possible permutations of flit priorities), but the sequential dependence for final port allocation remains a problem, and the area and power cost of the redundant parallel computations is very high with this scheme.

Fig. 6 shows a high-level comparison of the buffered and bufferless port allocation problems. In a traditional buffered router, each output port can make its arbitration decision independently: multiple flits request that output port, the port arbiter chooses one winner, and the remaining flits stay in their queues. In contrast, port allocation is inherently more difficult in a bufferless deflection router than in a buffered router, because the decision is global

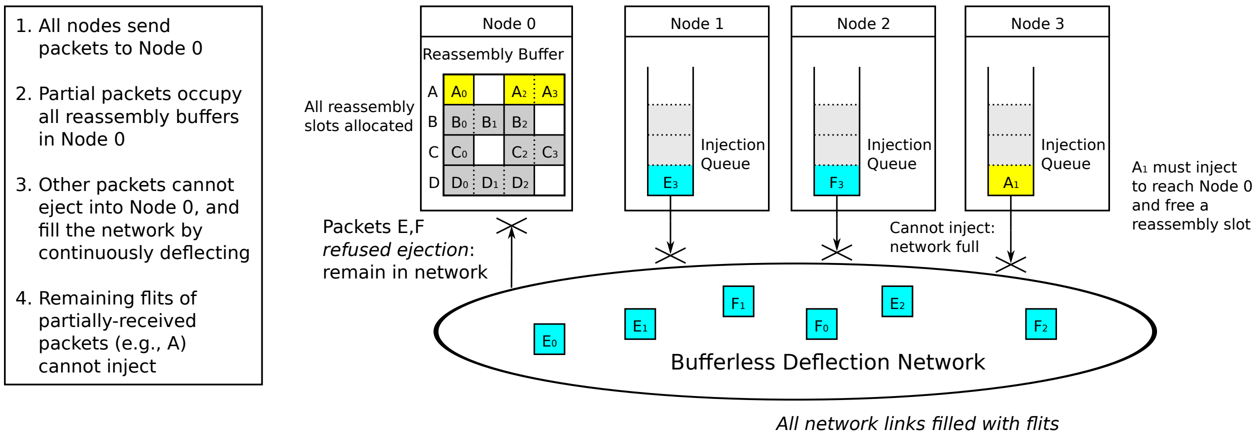

1. All nodes send packets to Node 0
2. Partial packets occupy all reassembly buffers in Node 0
3. Other packets cannot eject into Node 0, and fill the network by continuously deflecting
4. Remaining flits of partially-received packets (e.g., A) cannot inject
Node 0
Reassembly Buffer
All reassembly slots allocated
A
B
C
D
A0
A2
A3
B0
B1
B2
C0
C2
C3
D0
D1
D2
Node 1
Node 2
Node 3
Injection Queue
E3
F3
A1
A1 must inject to reach Node 0 and free a reassembly slot
Packets E,F refused ejection: remain in network
Cannot inject: network full
E0
E1
F1
F0
E2
F2
Bufferless Deflection Network
All network links filled with flits

Figure 7: Deadlock due to reassembly-buffer overflow.

over all outputs. The algorithm requires that we obey priority order, and so flits must pass through a sort network before allocating ports. Then, port allocation occurs sequentially for each flit in priority order. Because flits that lose arbitration deflect to other ports, lower-priority flits cannot claim outputs until the deflection is resolved. Thus, the port allocator for each flit must wait for the previous port allocator. The sequential dependence creates a long critical path; the worst case, in which all flits contend for one port, limits router speed. Finding a full permutation with deflections, in a bufferless router, has inherently less parallelism, and more computation, than port allocation in a buffered router.

3.2 Priority Arbitration

The priority arbitration problem – computing a priority order on incoming flits – also becomes more costly in a bufferless deflection network. In particular, the network must explicitly avoid livelock through careful design of its priority scheme. One option (used in [38]) is an Oldest-First priority scheme to guarantee flit delivery: the oldest flit will always make forward progress, and once it is delivered, another flit becomes the oldest. However, this scheme requires an age field in every packet header, and the field must be wide enough to cover the largest possible in-flight window. The arbiter then needs to sort flits by priorities in every cycle. A bitonic sort network [5] can achieve this sort in three stages for 4 flits. Unfortunately, this is a long critical path in a high-speed router, especially when combined with the port allocator; alternately, pipelining the computation yields a longer router latency, impacting performance significantly.

3.3 Reassembly Buffers

A bufferless deflection network must provide buffer space at network entry and exit: injection and reassembly buffers, respectively. Injection buffers are necessary because injection can only occur when there is an empty slot in the network, so new traffic must wait its turn; reassembly buffers are needed because deflection routing may fragment packets in transit. Injection buffers pose relatively little implementation difficulty: a node (e.g., a core or a shared cache) can stall when its injection FIFO fills, and can generate data on demand (e.g., from the cache, in the case of a cache-miss request). However, reassembly buffers lead to correctness issues that, without a more complex solution, yield large worst-case space requirements to avoid deadlock. Worst-case sizing is impractical for any reasonable design; therefore, this is a *fundamental* problem with bufferless deflection networks that must be solved at the algorithmic level.

Despite the fundamental nature of this problem in deflection routing, management of reassembly buffer space has not yet been considered in existing deflection-routed NoCs. BLESS [38] assumes infinite buffers, and then gives data for maximum reassembly buffer occupancy. In a real system, buffers are finite, and overflow must be handled. Michelogiannakis et al. [36] correctly note that in the worst case, a reassembly buffer must be sized to reassemble all packets in the system simultaneously.

To see why this is the case, observe the example in Fig. 7. Assume a simple reassembly-slot allocation algorithm that assigns space as flits arrive, and frees a packet's slot when reassembly is completed. The key observation is that a bufferless deflection network has no flow control: whereas in a buffered network, credits flow upstream to indicate free downstream buffer space (for both in-network buffers and final endpoint reception), nodes in a bufferless network are free to inject whenever there is a free outbound link at the local node. Thus, a reassembly buffer-full condition is not transmitted to potential senders, and it is possible that many packets are sent to one destination simultaneously. When all packets are sent to this single node (e.g., Node 0), the first several flits to arrive will allocate reassembly slots for their packets. Once all slots are taken, flits from other packets must remain in the network and deflect until the first packets are reassembled. Eventually, this deflecting traffic will fill the network, and block further injections. If some flits constituting the partially reassembled packets flits have not been injected yet (e.g., packet A), deadlock results. We have observed such deadlock for reasonable reassembly buffer sizes (up to 4 packet slots) in realistic workloads of network-intensive applications.

Fundamentally, this deadlock occurs because of a lack of *backpressure* (i.e., buffer credits) in the network. In other words, reassembly buffers have no way to communicate to senders that they are full, and so the only way to avoid oversubscription is worst-case provisioning. A bufferless network provides backpressure only in local injection decisions [23] – i.e., when the network is locally busy, a node cannot inject – which is not sufficient to prevent deadlock, as we just argued.

To build an effective bufferless deflection NoC, we *must* guarantee correctness with a reasonable reassembly buffer size. As argued above, the naïve locally-controlled buffer allocation leads to worst-case sizing, which can significantly reduce the area and energy benefits of bufferless routing. Because reassembly buffers are a necessary mechanism for deflection routing, and because the lack of flow control might allow deadlock unless buffers are unreasonably large, we consider the reassembly-buffer problem to be fundamentally critical to *correctness*, just as efficient port allocation and priority arbitration are fundamental to *practicality*. These three complexities directly motivate the key insights and mechanisms in our new router, *CHIPPER*.

4 CHIPPER: Mechanisms

As we have seen, bufferless deflection routing introduces several complexities that are not present in traditional buffered networks. Here, we introduce *CHIPPER* (Cheap-Interconnect Partially Permuting Router), a new router microarchitecture based on the key insight that these complexities are artifacts of a *particular formulation* of deflection routing, rather than fundamental limitations. By introducing a new architecture based on a *permutation network*, and two key algorithms, *Golden Packet* and *Retransmit-Once*, we provide a feasible implementation of bufferless deflection routing.

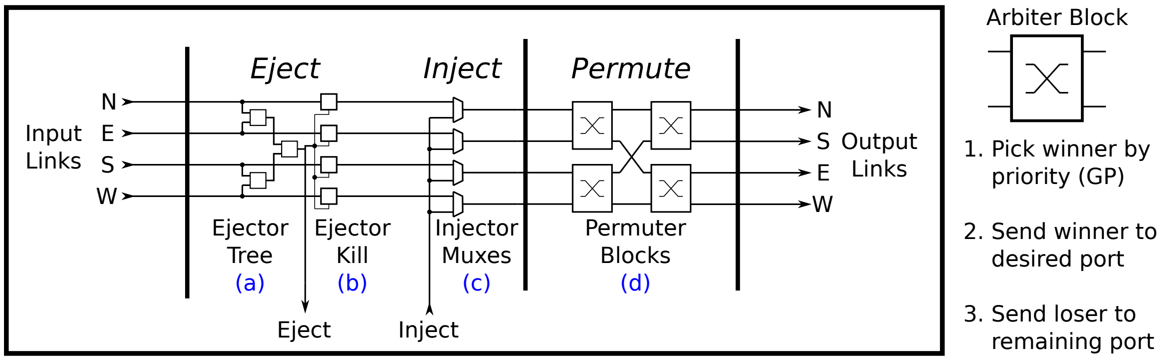


Figure 8: CHIPPER architecture: a permutation network replaces the traditional arbitration logic and crossbar.

4.1 Permutation Network Deflection Routing

Section 3.1 describes how deflection routing can lead to inefficient port allocation, because of the sequential dependence that deflection implies. We observe that sequential port allocation is not necessary for ensuring mutual exclusion on output ports. Instead, the deflection-routing problem can map to a permutation network. A network composed of 2x2 arbiter blocks that either pass or swap their arguments will implicitly give a 1-to-1 mapping of inputs to outputs.

In other words, if we assign the outputs of a permutation network to the output ports of a router, mutual exclusion naturally arises when flits contend for an output port, because at the final stage, only one flit can take the output. This idea leads to a completely new router organization.

Fig. 8 shows the high-level CHIPPER router architecture. The pipeline contains two stages: eject/inject (parts *(a)*, *(b)*, *(c)*, described in § 4.1.1 below) and permute. As shown, the permutation network replaces the *control* and *data* paths in the router: there is no crossbar, as each flit's data payload travels with the header bits through the permutation network. This leads to a more area-efficient design.

A permutation network directs deflected flits to free ports in an efficiently parallelized way. However, obeying priority order in port allocation still requires a sequential allocator. To eliminate this problem, we relax the problem constraints: *we require only that the highest-priority flit obtains its request*. As we will argue below (in § 4.2), this constraint is sufficient for livelock freedom. This also allows the permutation network to have a simpler design (with fewer stages) that gives only partial permutability.[2] The design is fully connected: if there is only one input flit, it can route from any input to any output. However, the single crossover limits the possible turn configurations. Note that the assignments of input ports relative to output ports is "twisted": N and E are paired on input, while N and S are paired on output. This arrangement is due to the observation that straight-through router traversals ($N \iff S$, or $E \iff W$) are more common than turns.[3]

Our second insight is that the priority sort and port allocation steps can be combined in the permutation network. (However, note that the permutation network does not need to perform a full sort, because we only need to determine the highest-priority flit.) The key to this arrangement is in the steering function of each 2x2 arbiter block: first, a priority comparison determines a winning flit; then, the winning flit picks the output that leads to its desired port. The losing flit, if present, takes the other output. This design preserves priority enforcement at least for the highest-priority flit, since this flit will always be a winning flit. In the highest-contention case, when all four flits request the same output, the arbiter becomes a combining tree. In the case where every flit requests a different output, the number of correct assignments depends only on the permutability of the arbiter.

4.1.1 Injection and Ejection

We must now consider injection and ejection in this arbiter. So far, we have assumed four input ports and four output ports, without regard to the fifth, local, router port. We could extend the permutation network to a fifth input and output. However, this has two disadvantages: it is not a power-of-two size, and so is less efficient in hardware cost; and more importantly, the local port has slightly different behavior. Specifically, the ejection port can only accept a flit destined for the local node, and injection can only occur when there is a free slot (either because of an empty input link or because of an ejection).

We instead handle ejection and injection as a separate stage prior to the arbitration network, as shown in Fig. 8. We insert two units, the ejector and the injector, in the datapath. This allows the stages to insert and remove flits before the set of four input flits reaches the arbiter. The ejector recognizes locally-destined flits, and picks at most one through the ejector tree (part *(a)* in the figure). The ejector tree must respect the priority scheme, but as we will argue in the next section, our *Golden Packet* scheme is very cheap. When a flit is chosen for ejection, the tree directs it to the local ejection port, and the ejector kill logic (part *(b)*) removes it from the pipeline. Finally, when a flit is queued for injection, the injector finds an empty input link (picking one arbitrarily if multiple are available, not shown in the figure for simplicity) and directs the local injection port to this link via the injector muxes (part *(c)* in the figure). The resulting flits then progress down the pipeline into the permute stage. As we note in § 5.7, the router's critical path is through the permutation network; thus, the eject/inject stage does does not impact the critical path.

4.1.2 Why Partial Permutability?

Note that the partial permutability of the CHIPPER permutation network is an important feature of the design. As discussed above, this design decision limits the permutations (flit-turn combinations) that are possible. However, it also simplifies the router microarchitecture in two ways. First, and most obvious, it allows for a shorter critical path, relative to a three-stage fully permutable network. Second, and less obvious, a fully-permutable network cannot have *local steering functions*. To see why, refer to the three-stage bitonic sort network previously shown in Fig. 5. For

[2]While this increases deflection rate, we show in our evaluations in § 5 that the impact in the common case is minimal. The critical-path and simplicity savings thus outweigh this cost.

[3][25] also makes use of this observation to obtain a cheap microarchitecture in a different way.

a given input-output port pair, a flit has two possible paths: the arbiter blocks traversed in the first and third stages are fixed by the port choices, but in the second stage, either arbiter block offers a path. The steering function in the first stage must choose one path or the other (once the flit reaches the second stage, the steering function has only one productive choice). In this first stage, the steering function has only *local knowledge* – it knows the desired destinations of only two out of four input flits. Thus, it cannot direct the flit to avoid collisions in all cases. In other words, full permutability not only requires at least three stages (for four inputs), but also requires *global scheduling* at some stage of the permutation network. This may still yield a shorter critical path than a sequential port allocator. However, we do not evaluate this design option further, due to the disproportionate complexity involved in permutation network steering.

4.2 Golden Packet: Cheap Priorities

So far, we have addressed the port allocation problem. An efficient priority scheme forms the second half of a cheap router. In our design, each 2x2 arbiter block must take two flits and decide the winner. The Oldest-First priority scheme used by BLESS [38] decides this with an age comparison (breaking ties with other fields). However, this is expensive, because it requires a wide age field in the packet header, and large comparators in the arbiter. We wish to avoid this expense, even if it may sacrifice a little performance.

We start with the explicit goal of preserving livelock freedom, while stripping away anything unnecessary for that property. We observe that it is sufficient to pick a single packet, and prioritize that packet globally above all other packets for long enough that its delivery is ensured. If every packet in the system eventually receives this special status, then every packet will eventually be delivered. This constitutes livelock freedom. We call this scheme, which prioritizes a single packet in the system, *Golden Packet*.

We will introduce Golden Packet, or GP, in two pieces. First, GP provides a set of prioritization rules that assume the golden packet is already known. Then, GP defines an implicit function of time that rotates through all possible packets to define which is golden.

Ruleset 1 Golden Packet Prioritization Rules

Golden Tie: If two flits are golden, the lower-numbered flit (first in the golden packet) wins.
Golden Dominance: If one flit is golden, it wins over any non-golden flit.
Common Case: Contests between two non-golden flits are decided pseudo-randomly.

4.2.1 Prioritization Rules

The GP prioritization rules are given in Ruleset 1. These rules are designed to be very simple. If a flit header already contains a bit indicating golden status, then a GP arbiter requires only a comparator as wide as the flit sequence number within a packet – typically 2 or 3 bits – and some simple combinational logic to handle the two-golden-flit[4], one-golden-flit and the most common no-golden-flit cases. These rules guarantee delivery of the golden packet: the golden packet always wins against other traffic, and in the rare case when two flits of the golden packet contend, the Golden Tie rule prioritizes the earlier flit using its in-packet sequence number. However, since most packets are not golden, the Common Case (random winner) rule will be invoked most often. Thus, Golden Packet reduces critical path by requiring a smaller comparator, and reduces dynamic power by using that comparator only for the golden packet.

4.2.2 Golden Sequencing

We must specify which packet in the system is golden. All arbiters must have this knowledge, and must agree, for the delivery guarantee to work. This can be accomplished by global coordination, or by an implicit function of time computed at each router. We use the latter approach for simplicity.

We define a golden epoch to be L cycles long, where L is at least as large as the maximum latency for a golden packet, to ensure delivery. (This upper bound is precisely the maximum Manhattan distance times the hop latency for the first flit, and one more hop latency for each following flit, since the Golden Packet will never be misrouted and thus will take a minimal path.) Every router tracks golden epochs independently. In each golden epoch, either zero or one packet is golden.

[4]The two-golden-flit case is only possible when two flits from the single Golden Packet contend, which happens only if some flits in the packet had been deflected before becoming golden while in flight: once the packet is golden, no other traffic can cause its flits to deflect.

The golden packet ID rotates every epoch. We identify packets by (sender, transaction ID) tuples (in practice, the transaction ID might be a sender MSHR number). We assume that packets are uniquely identifiable by some such tuple. We rotate through all possible tuples in a static sequence known to all routers, regardless of packets actually in the network. This sequence nevertheless ensures that every packet is eventually golden, if it remains in the network long enough, thereby ensuring its delivery. The golden sequence is given in Algorithm 2 as a set of nested loops. In practice, if all loop counts are powers of two, a router can locally determine the currently-chosen golden packet by examining bitfields in a free-running internal counter.

In our design, routers determine the golden-status of a packet in parallel with route computation. This check is lightweight: it is only an equality test on packet ID. Note that packets must be checked at least once per epoch while in transit, because they may become golden after injection. However, the check can be done off the critical path, if necessary, by checking at one router and forwarding the result in a header bit to the next router.

Algorithm 2 Golden Epoch Sequencing (implicit algorithm at each router)

```
while true do
  for $t$ in $N_{txn\_ids}$ do
    for $n$ in $N_{nodes}$ do
      packet from transaction id $t$ sent from node $n$ is golden for $L$ cycles
    end for
  end for
end while
```

4.3 Retransmit-Once: Flow Control for In-MSHR Buffering

As we motivated in § 3, reassembly buffers pose significant problems for bufferless deflection networks. In particular, because there is no feedback (backpressure [23]) to senders, correct operation requires that the buffers are sized for the worst case, which is impractical. However, a separate mechanism that avoids buffer overflow can enable the use of a much smaller reassembly space. Along with this insight, we observe that existing memory systems already have buffer space that can be used for reassembly: the *cache miss buffers* (MSHRs [29] and shared cache request queues/buffers). In fact, the cache protocol must already allocate from a fixed pool of request buffers at shared cache nodes and handle the buffers-full case; thus, our flow control solution unifies this protocol mechanism with network-level packet reassembly.

4.3.1 Integration with Caches: Request Buffers

In a typical cache hierarchy, buffering exists already in order to support cache requests. At private (L1) caches, MSHRs [29] (miss-status handling registers) track request status and buffer data as it is read from or written to the cache data array. This data buffer is ordinarily accessible at the bus-width granularity in order to transfer data to and from the next level of hierarchy. Similarly, at shared (L2) cache banks, an array of buffers tracks in-progress requests. These buffers hold request state, and also contain buffering for the corresponding cache blocks, for the same reasons as above. Because both L1 and L2-level buffers are essentially the same for flow control purposes, we refer to both as "request buffers" in this discussion.

We observe that because these request buffers already exist and are accessible at single-flit granularity, they can be used for reassembly and injection buffering at the NoC level. By considering the cache hierarchy and NoC designs together, we can eliminate the redundancy inherent in separate NoC reassembly and injection buffers. In particular, an injection queue can be constructed simply by chaining MSHRs together in injection order. Similarly, a reassembly mechanism can be implemented by using existing data-steering logic in the MSHRs to write arriving flits to their corresponding locations, thereby reassembling packets (cache blocks) in-place. By implementing separate injection and reassembly buffers in this way, we can truly call the network *bufferless*.

4.3.2 Flow Control: Retransmit-Once

The lack of flow control in a bufferless deflection network can lead to deadlock in worst-case situations. We showed in § 3.3 that deadlock occurs when reassembly buffers (or request buffers) are all allocated and additional traffic requires more buffers. § 3.3 shows that a simple mechanism to handle overflow based on local router decisions can lead to deadlock. Therefore, an explicit flow control mechanism is the most straightforward solution to allow for non-worst-case buffering.

The design space for managing request buffers is characterized by two design extremes. First, a flow control scheme could require a sender to obtain a buffer reservation at a receiver before sending a packet that requires a request buffer. This scheme can be implemented cheaply as a set of counters that track reservation tokens. However, reservation requests are now on the critical path for every request.

Alternately, a flow control scheme could operate opportunistically: it could assume that a buffer will always be available without a reservation, and recover in the uncommon case when this assumption fails. For example, a receiver might be allowed to drop a request or data packet when it does not have an available buffer. The system can then recover either by implementing retransmit timeouts in senders or by sending a retransmit request from receiver to sender (either immediately or when the space becomes available). This scheme has no impact on the critical path when a request buffer is available. However, recovery imposes additional requirements. In particular, senders must buffer data for possible retransmission, and possibly wait for worst-case timeouts.

Instead, we propose a hybrid of these two schemes, shown in Fig. 9, called *Retransmit-Once*. The key idea is that the *opportunistic* approach can be used to establish a reservation on a request buffer, because the sender can usually regenerate the initial request packet easily from its own state. The remainder of the transaction then holds this reservation, removing the need to retransmit any data. This combination attains the main advantage of the opportunistic scheme – zero critical-path overhead in the common case – while also removing retransmit-buffering overhead in most cases. In other words, there is *no explicit retransmit buffer*, because only the initial request packet can be dropped and the contents of this packet is implicitly held by the sender.

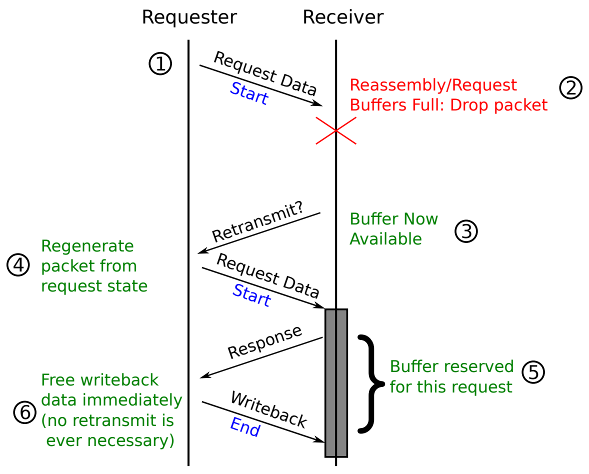

Requester
Receiver
①
Request Data
Start
Reassembly/Request Buffers Full: Drop packet
②
Retransmit?
Buffer Now Available
③
④
Regenerate packet from request state
Request Data
Start
Response
Buffer reserved for this request
⑤
⑥
Free writeback data immediately (no retransmit is ever necessary)
Writeback
End

Figure 9: Retransmit-Once flow control scheme.

We will examine the operation of Retransmit-Once in the context of a simple transaction between a private L1 cache (the requester) and a shared L2 cache slice (the receiver). The L1 requests a cache block from the L2; the L2 sends the data, and then, after performing a replacement, the L1 sends a dirty writeback to the L2 in another data packet. Request buffers are needed at both nodes. However, the requester (L1) initiates the request, and so it can implicitly allocate a request buffer at its own node (and stall if no buffer is available). Thus, we limit our discussion to the request buffer at the receiver (L2). Note that while we discuss only a simple three-packet transaction, any exchange that begins with a single-flit request packet can follow this scheme[5].

In the common case, a request buffer is available and the opportunistic assumptions hold, and Retransmit-Once has no protocol overhead. The scheme affects operation only when a buffer is unavailable. Such an example is shown in Fig. 9. The L1 (sender) initially sends a single-flit request packet to the L2 (receiver), at ①. The packet has a special flag, called the *start-bit*, set to indicate that the transaction requires a new request buffer (this can also be implicit in the packet type). In this example, the L2 has no request buffers available, and so is forced to drop the packet at ②. It records this drop in a *retransmit-request queue*: it must remember the sender's ID and the request ID (e.g., MSHR index at the sender) in order to initiate a retransmit of the proper request. (In practice, this queue can be a bitfield with one bit per private MSHR per sender, since ordering is not necessary for correctness.)

[5]For more complex protocols that may send a data packet to a third party (e.g., more complex cache mapping schemes where writebacks may go to different L2 banks/slices than where the replacement data came from), a separate control packet can make a reservation at the additional node(s) in parallel to the critical path of the request.

Some time later, a buffer becomes available, because another request completes at ③. The receiver (L2) then finds the next retransmit in its retransmit-request queue, and sends a packet to initiate the retransmit. It also *reserves* the available buffer resource for the sender (L1). This implies that only one retransmit is necessary, because the request buffer is now guaranteed to be reserved. The L1 retransmits its original request packet from its request state at ④. The L2's request buffer is reserved for the duration of the request at ⑤, and the transaction continues normally: the L2 processes the request and sends a data response packet. Finally, in this example, the L1 sends a dirty writeback packet. This last packet has a special *end-bit* set that releases the request buffer reservation. Importantly, during the remainder of the sequence, the L1 never needs to retransmit, because the L2 has reserved a request buffer with reassembly space. Thus, no retransmit buffering is necessary. Accordingly, when the L1 sends its dirty writeback, it can free all resources associated with the transaction at ⑥, because of this guarantee.

Algorithms 3 and 4 specify flow-control behavior at the receiver for the first and last packets in a transaction. A counter tracks available buffers.

4.3.3 Interaction with Golden Packet

Finally, we note that Retransmit-Once and Golden Packet coexist and mutually maintain correctness because they operate at different protocol levels. Golden Packet ensures correct flit delivery without livelock. Retransmit-Once takes flits that are delivered, and provides deadlock-free packet reassembly and request buffer management, regardless of how flit delivery operates and despite the lack of backpressure. In particular, Golden Packet always dictates priorities at the network router level: a packet that has a reserved buffer slot destination is no different from any other packet from the router's point of view. In fact, golden flits may contend with flits destined to reserved buffer spaces, and cause them to be deflected or to not be ejected in some cycle. However, correctness is not violated, because the deflected flits will eventually be delivered (as guaranteed by Golden Packet) and then reassembled (by Retransmit-Once).

Algorithm 3 Receiving a packet with the *start-bit*

```
if $slots > 0$ then
    $slots \Leftarrow slots - 1$
    allocate buffer slot and return
else
    set retransmit bit for sender $(node, transactionID)$
    drop packet
end if
```

Algorithm 4 Receiving a packet with the *end-bit*

```
if pending retransmits then
    send retransmit request indicated by next set retransmit bit
else
    $slots \Leftarrow slots + 1$
end if
```

4.4 Router Microarchitecture

4.4.1 Permutation Network

The arbiter block steering function is presented in Algorithm 5. The result of this function is a binary value: "swap" or "no swap". This signal drives a set of MUXes on the arbiter block outputs that either pass or swap the flit inputs.

Inputs	Stage 1	Stage 2	Outputs
N	N, S: output 0	N, E: output 0	N
E	E, W: output 1	S, W: output 1	S
S	N, S: output 0	E, N: output 0	E
W	E, W: output 1	W, S: output 1	W

Table 1: Steering functions for two-stage CHIPPER permuter

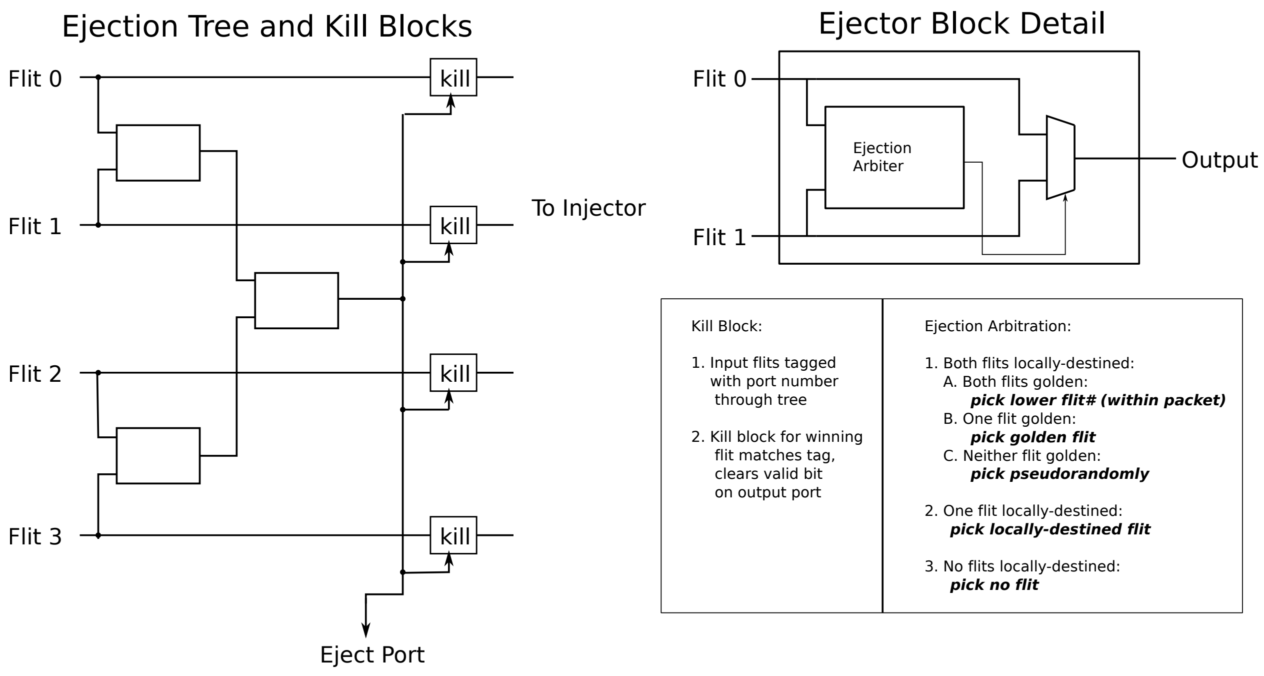
The desired-output-port bits d_0 and d_1 are computed according to a flit's desired router output according to the layout of the permutation network. In the two-stage partial permutation network used in CHIPPER, there is only one path to a given output port; the corresponding steering functions are given in Table 1. At the first stage, one output leads to each of the two second-stage arbiter blocks, and the d_i bits are computed according to the second-stage block that leads to the desired port. At the second stage, the output ports are attached directly, and so steering the winner is trivial. If neither of the output ports is productive (i.e., if a flit is deflected in the first stage and is choosing a port at the second stage), it can take either port.

Algorithm 5 Arbiter Block Steering Algorithm

```
**Given**: $GP_0$, $GP_1$: golden-bit for flits 0, 1
**Given**: $n_0$, $n_1$: in-packet sequence number for flits 0, 1
**Given**: $d_0$, $d_1$: desired output port (0 or 1) for flits 0, 1
$winner \Leftarrow 0$
if $GP_0 \wedge GP_1$ then
    $winner \Leftarrow (n_1 > n_0)$ ? 1 : 0
else if $GP_0$ then
    $winner \Leftarrow 0$
else if $GP_1$ then
    $winner \Leftarrow 1$
else
    $winner \Leftarrow pseudorandom()$
end if
$swap \Leftarrow (winner = 0 \wedge d_0 = 1) \vee (winner = 1 \wedge d_1 = 0)$
```

4.4.2 Ejection

The ejection logic was summarized previously in Fig. 8. Here, we discuss the particulars of the ejector. Fig. 10 shows the ejection tree and a single ejector block in detail. As noted previously, ejection must honor the priority scheme as the permuter stage does, or else the livelock guarantee will be broken. When using the Golden Packet scheme in particular, however, this functionality is very cheap. The ejector tree determines a single flit that wins the ejector port in a given cycle among multiple potential contenders. The arbitration rules given in the ejector block detail are exactly identical to the permuter arbiter block algorithm given in Algorithm 5 with the additional stipulation that only locally-destined flits are considered. Finally, the ejector kill blocks remove ejected flits from the pipeline before they advance to the injector and permuter units. This is implemented by carrying a two-bit tag with each flit through the ejector tree to indicate input port; each ejector block watches for a match on the ejection bus, and clears the valid-bit on the corresponding output port on a match.



Kill Block:

1. Input flits tagged with port number through tree
2. Kill block for winning flit matches tag, clears valid bit on output port

Ejection Arbitration:

1. Both flits locally-destined:
   A. Both flits golden:
      pick lower flit# (within packet)
   B. One flit golden:
      pick golden flit
   C. Neither flit golden:
      pick pseudorandomly
2. One flit locally-destined:
   pick locally-destined flit
3. No flits locally-destined:
   pick no flit

Figure 10: Ejector detail.

4.4.3 Injection

The injector's job is simple: when a flit is waiting for injection, the injector control logic looks for any open slot (cleared valid bit) on the four flit paths as they pass into the permutation network. It then directs the injection MUXes accordingly. Conceptually, the control logic can implement a priority-encoder among empty inputs in order to select where to place injected flits. (An alternate design could keep separate injection queues per input, but we found that this was not necessary for adequate performance.) A detail of the injector logic is shown in Fig. 11.

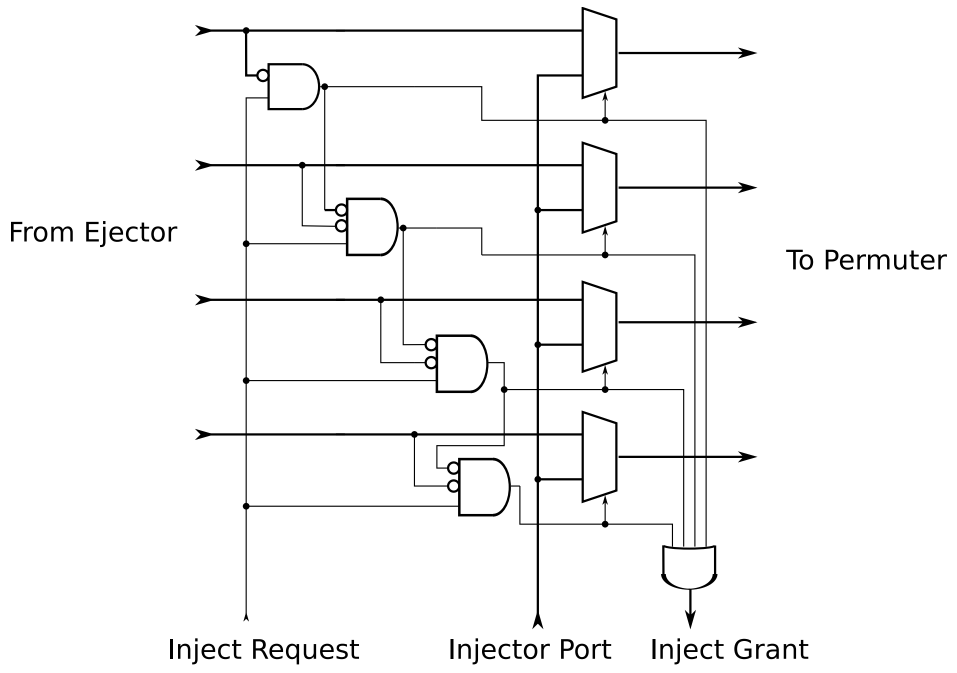


Figure 11: Injector detail. Thin wires represent single valid-bits.

4.4.4 Request Buffers (MSHRs)

As we described in § 4.3.1, cache systems employ *request buffers* to track in-flight requests. In a distributed, NoC-based architecture, these structures are placed at each node (private cache at a core, or shared cache slice). Fig. 12 illustrates these buffers in more detail. First proposed in [29] as *MSHRs* (miss-status handling registers), the buffers must track the current state of the request (this will be specific to coherence and memory-system implementation), must track the address or cache block of the request, and must contain buffer space to receive the data as it returns from the memory system.

4.4.5 Request Buffers for Injection and Reassembly

Fig. 13 shows how the existing MSHR/request buffer structures in a shared-cache memory system are used as injection and reassembly buffer space. We can consider the injection and reassembly structures separately. First, an injection queue is maintained by adding a linked-list pointer (MSHR index) to each MSHR, and a queue-head pointer. These fields create a singly-linked list of MSHRs, and the data to be injected is stored in each MSHR (or implied by its control state, if a control packet is queued). The injection datapath requires a separate read port on the MSHR file, and uses this read port to stream at most one flit per cycle.

Reassembly is only slightly more involved. First, the destination MSHR must be determined. An incoming flit may either be tagged with an MSHR index if its corresponding transaction has been reserved an MSHR, or may not be tagged. If it is tagged, its MSHR (reassembly buffer) is reserved, and the flit proceeds to that buffer. If it is not tagged, allocation logic attempts to allocate a free MSHR. If none exists, the flit is dropped, according to the Retransmit-Once scheme (§ 4.3).

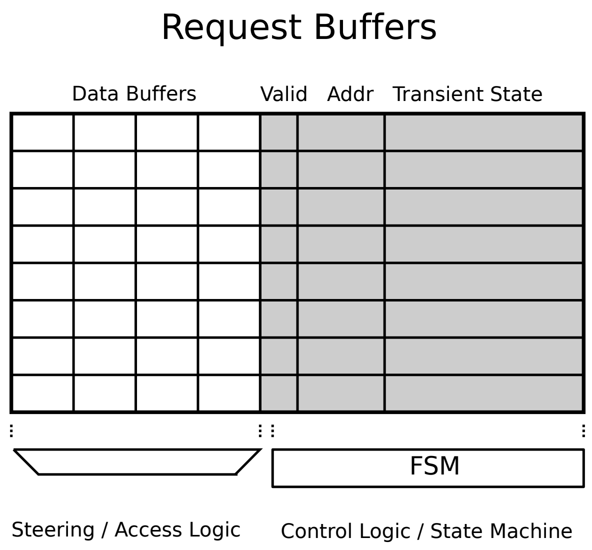


Figure 12: Request buffer detail: arrays of request buffers exist in cores' private caches, and also in shared cache slices, in order to track requests in flight.

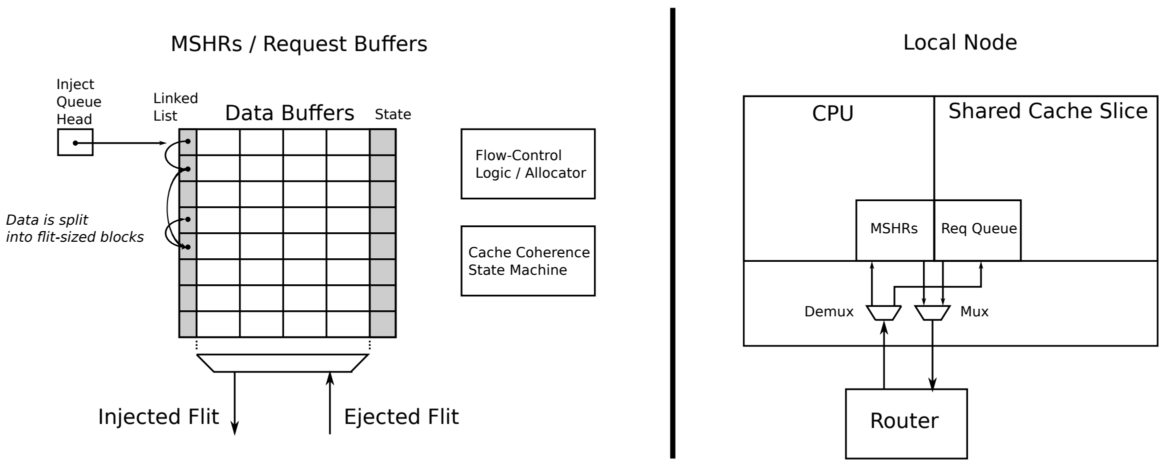


Figure 13: Integration of the existing MSHR / request buffer structures as injection and reassembly buffer space for CHIPPER.

Note that this mechanism is guaranteed to work: Retransmit-Once ensures that all packets are either *(i)* control packets that begin a transaction, are a single flit in length, and thus require no reassembly buffering, or *(ii)* data packets that have a buffer reserved (i.e., have a valid destination-MSHR tag). Thus, the control logic need only handle these two cases, and does not require any associative logic.

Once the flit's destination buffer is determined, the flit is written into the corresponding MSHR. This requires a second write port, sustaining at most one flit per cycle of bandwidth. Because the associative match described above may take some time, the ejection path may be pipelined.

The total cost of this integration (above a baseline MSHR structure) is thus: linked-list fields and head pointer ($\log_2 N$ bits per MSHR for N MSHRs); one additional read port, and one write port, on the MSHR file; the control logic to implement Retransmit-Once, most likely implemented with the protocol finite-state machine; and finally, the minimal storage space required for the retransmit-request bitfield, as described in § 4.3.

Parameter	Setting
System topology	8x8 mesh, dense configuration (core + shared cache at every node); 4x4 for multithreaded
Core model	Out-of-order x86, 128-entry instruction window, 16 MSHRs
Private L1 cache	64 KB, 4-way associative, 64-byte block size
Shared L2 cache	perfect (always hits), distributed (S-NUCA [24]), 16 request buffers (reassembly/inject buffers) per slice
Coherence protocol	Simple directory-based, based on SGI Origin [30], perfect directory
Interconnect Links	1-cycle latency, 128-bit flit width (4 flits per cache block)
Baseline buffered router	2-cycle latency, 4 VCs/channel, 8 flits/VC
Baseline BLESS router	2-cycle latency, FLIT-BLESS [38]

Table 2: System parameters used in our evaluation.

5 Evaluation

Our goal is to build a cheap, simple bufferless deflection router while minimally impacting performance for our target, low-to-medium-load applications. We evaluate two basic metrics: performance (application-level, network-level, and operational frequency), and hardware cost (network power and area). We compare CHIPPER to a traditional buffered NoC, as well as a baseline bufferless NoC, BLESS [38]. We will show performance results from simulation, and hardware cost results (including per-workload power) from RTL synthesis of BLESS and CHIPPER models, as well as ORION [55].

5.1 Methodology

We evaluate our proposed design using an in-house cycle-accurate simulator that runs both multiprogrammed and multithreaded workloads. For multiprogrammed runs, we collect instruction traces from SPEC CPU2006 [49] applications, as well as several real-world desktop and server applications (including two commercial web-search traces). We use PinPoints [43] to select representative phases from each application, and then collect instruction traces using a custom Pin-tool [32]. For multithreaded workloads, we collect instruction traces from SPLASH-2 [57], annotated with lock and barrier information to retain proper thread synchronization. Power, area and timing cost results come from hardware models, described in § 5.7. Power results in this section are based on cycle-accurate statistics from workload simulations and represent total network power, including links.

Each multiprogrammed simulation includes a 40M cycle warmup, and then runs until every core has retired 10M instructions. Applications freeze statistics after 10M instructions but continue to run to exert pressure on the system. We found that warmup counters on caches indicate that caches are completely warm after 40M cycles for our workloads, and 10M instructions is long enough for the interconnect to reach a steady-state. Each multithreaded simulation is run until a certain number of barriers (e.g., main loop iterations).

5.2 System Design and Parameters

We model an 8x8-mesh CMP for our multiprogrammed evaluations and a 4x4-mesh CMP for our multithreaded evaluations. Detailed cache, core and network parameters are given in Table 2. The system is a shared-cache hierarchy with a distributed shared cache. Each node contains a compute core, a private cache, and a slice of shared cache. Addresses are mapped to cache slices with the S-NUCA scheme [24]: the lowest-order bits of the cache block number determine the home node. The system uses a directory-based coherence protocol based on the SGI Origin [30]. We also evaluate sensitivity to cache mapping with a locality-aware scheme.

Importantly, we model a *perfect shared cache* in order to stress the interconnect: every access to a shared cache slice is a hit, so that no requests go to memory. This isolates the interconnect to provide an upper bound for our performance degradation – in other words, to report conservative results.

5.3 Workloads

Multiprogrammed: We run 49 multiprogrammed workloads, each consisting of 64 independent programs. 39 of these workloads are homogeneous, consisting of 64 copies of one application. The remaining 10 are randomly-chosen mixes from our set of 39 applications. Our application set consists of 26 SPEC CPU2006 benchmarks (including two traces of mcf), three SPEC CPU2000 benchmarks (vpr, art, crafty), and 10 other server and desktop traces: health

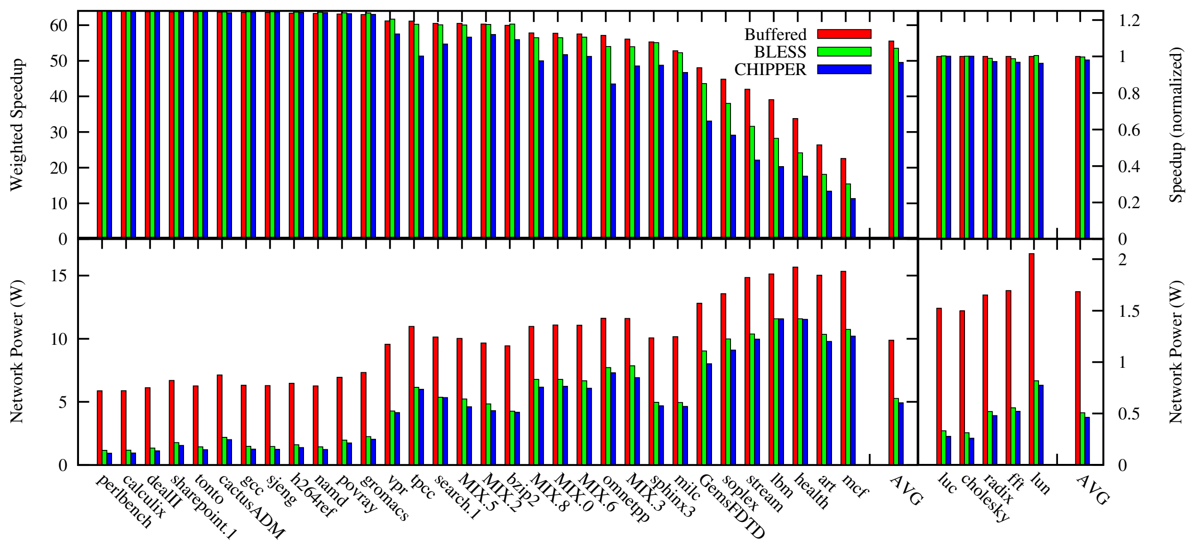


Figure 14: Application performance and network power comparisons.

(from the Olden benchmarks [45]), matlab [33], sharepoint.1, sharepoint.2 [37], stream [34], tpcc [1], xml (an XML-parsing application), search-1, search-2 (web-search traces from a commercial search engine).
Multithreaded: We run five applications from the SPLASH-2 [57] suite: `fft`, `luc`, `lun`, `radix` and `cholesky`. As described in § 5.1, we delineate run lengths by barrier counts: in particular, we run `cholesky` for 3 barriers, `fft` for 5, `luc` for 20, `lun` for 10, and `radix` for 10 barriers.

5.4 Application-Level Performance

In multiprogrammed workloads, we measure application-level performance using the weighted speedup metric [48]:

$$WS = \sum_{i=1}^{N} \frac{IPC_{shared}}{IPC_{alone}} \qquad (1)$$

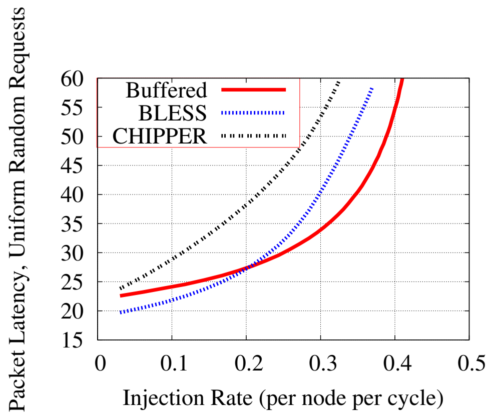
We compute weighted speedup in all workloads using a buffered-network system as the baseline (IPC_{alone} values). This allows direct comparison of the networks. In multithreaded workloads, we compare execution times directly by normalizing runtimes to the buffered-network baseline.
Overall results: For our set of multiprogrammed workloads, CHIPPER degrades weighted speedup by 13.6% on average (49.8% max in one workload) from the buffered network, and 9.6% on average (29.9% max) from BLESS. For our set of multithreaded workloads, CHIPPER degrades performance (increases execution time) by 1.8% on average (3.7% max). As described above, these results are pessimistic, obtained with perfect shared cache in order to stress the interconnect. Additionally, for the least intensive third of multiprogrammed workloads, and for the multithreaded workloads we evaluate, performance impact is completely negligible.
Per-workload results: However, average performance degradation does not tell the whole story. Examining degradation by workload intensity yields more insight. Fig. 14 shows weighted speedup (for multiprogrammed) and normalized runtime (for multithreaded), as well as network power, for a representative subset of all multiprogrammed workloads (for space reasons) and all multithreaded workloads. Behavior can be classified into two general trends. First, for workloads that are not network-intensive, CHIPPER experiences very little degradation relative to both buffered and BLESS networks. This is the best case for a cheap interconnect, because the application load is low, requiring much less than the peak capacity of the baseline buffered network. As workloads begin to become more network-intensive, moving to the right in Fig. 14, both bufferless networks (BLESS and CHIPPER) generally degrade relative to the buffered baseline. We note in particular that the SPLASH-2 multithreaded workloads experience very little degradation because of low network traffic. As described in [38], bufferless routing is a compelling option for low-to-medium load cases. We conclude that at low load, CHIPPER is effective at preserving performance while significantly reducing NoC power.

5.5 Power and Energy Efficiency

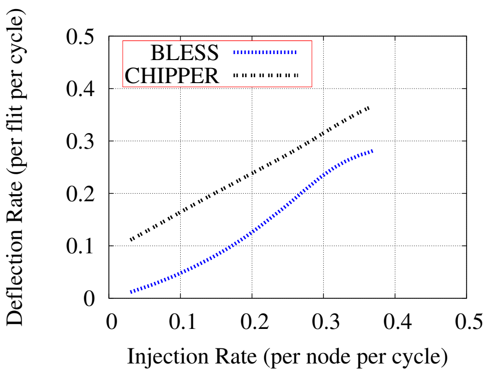
Power: Figure 14 shows average network power for each evaluated workload. These results demonstrate the advantage of bufferless routing at low-to-medium load. Both CHIPPER and BLESS have a lower power ceiling than the buffered router, due to the lack of buffers. Thus, in every case, these router designs consume less power than a buffered router. In multiprogrammed workloads, CHIPPER consumes 54.9% less power on average than buffered and 8.8% less than BLESS; with multithreaded workloads, CHIPPER consumes 73.4% less than buffered and 10.6% less than BLESS.
System energy efficiency: The discussion above evaluates efficiency only within the context of network power. We note that when full-system power is considered, slowdowns due to interconnect bottlenecks can have significant negative effects on total energy. A full evaluation of this tradeoff is outside the scope of this work. However, the optimal point depends entirely on the fraction of total system power consumed by the NoC. If this fraction is sufficiently large, the energy tradeoffs shown here apply. For low-to-medium intensity loads, minimal performance loss coupled with significant router power, area and complexity reduction make CHIPPER a favorable design tradeoff regardless of the fraction of system power consumed by the NoC.

5.6 Network-Level Performance

We present latency and deflection as functions of injection rate for uniform random traffic in Figures 15a and 15b respectively. We show in Fig. 15a that CHIPPER saturates more quickly than BLESS, which in turn saturates more quickly than a buffered interconnect. Furthermore, CHIPPER clearly has a higher deflection rate for a given network load, which follows from the less exhaustive port allocator.



(a) Total latency



(b) Deflection rate

Figure 15: Network-level evaluations: latency and deflection with synthetic traffic.

Sensitivity: We evaluate sensitivity to two network parameters: golden epoch length, and reassembly buffer size. For the former, we observe that as epoch length sweeps from 8 (less than the minimum required value for a livelock freedom guarantee) to 8192, and synthetic injection rate sweeps from 0 to network saturation, *IPC* varies by 0.89% maximum. This small effect is expected because golden flits comprise 0.37% on average (0.41% max) of router traversals over these sweeps. The epoch length is thus unimportant for throughput.

The reassembly buffer size can have a significant effect if it is too small for the presented load. When reassembly buffers are too small, they become the interconnect bottleneck: the opportunistic assumption of available receiver space fails, and most requests require retransmits. With a 25 MPKI synthetic workload and only one buffer per node, the retransmit rate is 72.4%, and IPC drops by 56.7% from the infinite-buffer case. However, IPC reaches its ideal peak with 8 buffers per node at this workload intensity, and is flat beyond that; for a less-intensive 10 MPKI synthetic workload, performance reaches ideal at 5 buffers per node. In the application workloads, with 16 buffers per node, the retransmit rate is 0.0016% on average (0.021% max). Thus, we conclude that when buffers are sized realistically (16 buffers per node) the overhead of Retransmit-Once is negligible.
Effect of Locality-Aware Data Mapping: Finally, we evaluate the effects of data locality on network load, and thus, the opportunity for a cheaper interconnect design. We approximate a locality-aware cache mapping scheme by splitting the 8x8-mesh into sixteen 2x2 neighborhoods or four 4x4 neighborhoods: for each node, its cache blocks are

	Buffered	BLESS	CHIPPER	% Δ Buffered → CHIPPER	% Δ BLESS → CHIPPER
Area	480174 μm^2	311059 μm^2	306165 μm^2	36.2% reduction	1.6% reduction
Timing (crit path)	1.88ns	2.68 ns	1.90 ns	1.1% increase	29.1% reduction

Table 3: Hardware cost comparisons for a single router in a 65nm process.

striped statically across only its neighborhood. This is a midpoint between one extreme, in which every node has its entire working set in its local shared-cache slice (and thus has zero network traffic) and the other extreme, S-NUCA over the whole mesh, implemented in our evaluations above.

We find that for the set of 10 random-mix workloads on an 8x8-mesh (11.7% weighted speedup degradation from buffered to CHIPPER), modifying cache mapping to use 4x4 neighborhoods reduces weighted speedup degradation to 6.8%, and using 2x2 neighborhoods reduces degradation to 1.1%. This result indicates that mechanisms that increase locality in NoC traffic can significantly reduce network load, and provide further motivation for cheap interconnects such as CHIPPER.

5.7 Hardware Complexity

In order to obtain area and timing results, and provide power estimates for workload evaluations, we use RTL (Verilog) models of CHIPPER and BLESS, synthesized with the Synopsys toolchain using a commercial 65nm process. We model the buffered baseline timing with a publicly available buffered NoC model from Stanford [50]. However, because of an inadequate wire model, we were not able to obtain adequate area/power estimates for the flit datapath; for this reason, we used ORION [55] to obtain estimates for the buffered baseline area/power. For both bufferless routers, we synthesized control logic, and then added crossbar area and power estimates from ORION. CHIPPER is also conservatively modeled by including crossbar area/power, and shrinking the permutation network to only the control-path width; further gains should be possible with a realistic layout that routes the datapath through the permutation network. For both bufferless models, we synthesize a single router, with parameters set for an 8x8-mesh. Finally, for all three networks, we model link power with ORION assuming 2.5mm links (likely conservative for an 8x8-mesh). We do not model reassembly buffers, since we use MSHRs for this purpose.

Table 3 shows area and timing results for CHIPPER, BLESS and traditional buffered routers. The reduction in area from buffered to either of the bufferless designs is significant; this gap is dominated by buffers (35.3% of the buffered router's area). The additional reduction from BLESS to CHIPPER is due to simpler control logic. Altogether, CHIPPER has 36.2% less area than the buffered baseline. Additionally, the critical path delays are comparable for both designs: CHIPPER's critical path, which is through the sort network, is only 1.1% longer than the critical path in the buffered model. We conclude that CHIPPER can attain nearly the same operating frequency as a buffered router while reducing area, power (as shown in § 5.5) and complexity significantly.

6 Related Work

Deflection routing: Deflection routing was first introduced as hot-potato routing in [3]. It has found use in optical networks [8], where deflection is cheaper than buffering. Recently, bufferless routing has received renewed interest in interconnect networks. BLESS [38] motivates bufferless deflection routing in on-chip interconnect for cost reasons. However, it does not consider arbitration hardware costs, and it does not solve the reassembly-buffer problem. The Chaos router [28] is an earlier example of deflection routing. The router is not bufferless; rather, it uses a separate deflection queue to handle contention. The HEP multiprocessor [47] and the Connection Machine [20] used deflection networks. Finally, [31, 52] evaluate deflection routing in several NoC topologies and with several deflection priority schemes. However, [31] does not evaluate application-level performance or model hardware complexity, while [52] does not show livelock freedom nor does it consider hardware cost of the deflection router control logic. Neither work examines the reassembly-buffer problem that we solve.

Drop-based bufferless routing: BLESS [38] is bufferless as well as deflection-based. However, several networks eliminate buffers without deflection. BPS [16] proposes bufferless routing that drops packets under contention. SCARAB [19] builds on BPS by adding a dedicated circuit-switched NACK network to trigger retransmits. This work evaluates hardware cost with detailed Verilog models. However, neither BPS nor SCARAB rigorously prove livelock freedom. Furthermore, the separate NACK network increases link width and requires a separate circuit-switching crossbar.

Other bufferless alternatives: Ring-based interconnects [46, 44] are particularly well-suited for bufferless operation, because no routing is required once a flit enters the ring: it simply travels until it reaches its destination. Rings have low complexity and cost, but scale worse than meshes, tori and other topologies beyond tens of nodes. Hierarchical bus topologies [53] offer another alternative, especially compelling when traffic exhibits locality. Both of these non-mesh topologies are outside the scope of this work, however.

Reducing cost and complexity in buffered routers: Elastic Buffer Flow Control [35] makes use of the buffer space inherent in pipelined channels to reduce buffer cost. The iDEAL router [26] reduces buffering by using dual-function links that can act as buffer space when necessary. The ViChaR router [39] dynamically sizes VCs to make more efficient use of a buffer budget, allowing reduced buffer space for equivalent performance. In all these cases, the cost of VC buffers is reduced, but buffers are not completely eliminated as in bufferless deflection routing. Going further, Kim [25] eliminates VC buffers while still requiring intermediate buffers (for injection and for turning). The work shares our goal of simple microarchitecture. Routing logic is simpler in [25] than in our design, because of buffering; however, [25] does not use adaptive routing, and requires flow control on a finer grain than Retransmit-Once to control injection fairness. [56] proposes buffer bypassing to reduce dynamic power and latency in a buffered router, and [36] evaluates such a router against BLESS. The paper's evaluation shows that with a custom buffer layout (described in [2]), an aggressive buffered design can have slightly less area and power cost than a bufferless deflection router, due to the overhead of BLESS arbitration and port allocation. However, our goal is specifically to reduce these very costs in bufferless deflection routing; we believe that by addressing these problems, we show bufferless deflection routing to be a practical alternative.

Improving performance and efficiency of bufferless NoCs: Several works improve on a baseline bufferless design for better performance, energy efficiency, or both. Jafri et al. in [23] propose a hybrid NoC that switches between bufferless deflection routing and buffered operation depending on load. Nychis et al. in [40] investigate congestion control for bufferless NoCs that improves performance under heavy load. Both mechanisms are orthogonal to our work, and CHIPPER could be combined with either or both techniques to improve performance under heavy load.

Permutation network: Our permutation network is a 2-ary 2-fly Butterfly network [12]. The ability of indirect networks to perform permutations is well-studied: [54] shows a lower bound on the number of cells required to configure any permutation. (For our 4-input problem, this bound is 5, thus our design is only partially permutable.) Rather, the new contribution of the CHIPPER deflection-routing permutation network is the realization that the deflection-routing problem maps naturally to an indirect network, with the *key difference* that contention is resolved at each 2x2 cell by misrouting rather than blocking. CHIPPER embeds these permutation networks within each node of the overall mesh network. To our knowledge, no other deflection router has made this design choice.

Livelock: Livelock freedom guarantees can be classified into two categories: probabilistic and deterministic. BLESS [38] proposes Oldest-First (as discussed in § 2.2.1), which yields an inductive argument for deterministic livelock freedom. Busch et al. [7] offer a routing algorithm with a probabilistic livelock guarantee, in which packets transition between a small set of priorities with certain probabilities. [28] also provides a probabilistic guarantee. Golden Packet provides a deterministic guarantee, but its key difference from [7, 28] is its end goal: it is designed to be as simple as possible, with hardware overhead in mind.

Deadlock: Deadlock in buffered networks is well-known [11] and usually solved by using virtual channels [10]. However, our reassembly-buffer deadlock is a distinct issue. Hansson et al. [18] observe a related problem due to inter-packet (request-response) dependencies in which deadlock can occur even when the interconnect itself is deadlock-free. Like the reassembly-buffer deadlock problem described in this paper, this occurs due to ejection backpressure: responses to previous requests cannot be injected, and so new requests cannot be ejected. However, our problem differs because it exists independently of inter-packet dependencies (i.e., could happen with only one packet class), and happens at a lower level (packet reassembly). [18] proposes end-to-end flow control with token passing as a solution to message-dependent deadlock, but assumes a window-based buffering scheme. Our flow-control scheme is distinguished by its opportunistic common-case, lack of explicit token passing, and lack of an explicit retransmit window due to integration into MSHRs.

7 Other Applications and Future Work

While *CHIPPER*'s design point is appealing for its simplicity, there is a large design space that spans the gap between large, traditional buffered routers and simple deflection routers. Several directions are possible for future work. First, the mechanisms that comprise CHIPPER are not limited to the specific design shown here, nor are they mutually dependent, and extensions of these techniques to other networks might allow for hardware cost reduction at other design points. Golden Packet can be extended to any non-minimal adaptive interconnect in order to provide livelock freedom. Likewise, the basic permutation-network structure can be used with other priority schemes, such as Oldest-First or an application-aware scheme [14, 13], by modifying the comparators in each arbiter block. Finally, Retransmit-Once offers deadlock freedom in any deflection network that requires reassembly buffers. In fact, it can also be extended to provide flow control for other purposes, such as congestion control; in general, it allows receivers to throttle senders when necessary, in a way that is integrated with the basic functionality of the network. Additionally, we have shown only one permutation network topology. A more detailed study of the effect of partial permutability on network-level and application-level performance would allow for optimizations that take advantage of properties of the presented traffic load. In particular, heterogeneity in the permutation network with regard to the more likely flit permutations (at center, edge and corner routers) might increase efficiency.

8 Conclusions

We presented *CHIPPER*, a router design for bufferless deflection networks that drastically reduces network power and hardware cost with minimal performance degradation for systems with low-to-medium network load. CHIPPER *(i)* replaces the router core with a partial permutation network; *(ii)* employs *Golden Packet*, an implicit token-passing scheme for cheap livelock freedom; and *(iii)* introduces *Retransmit-Once*, a flow-control scheme that solves the reassembly-buffer backpressure problem and allows use of MSHRs for packet reassembly, making the network truly bufferless. Our techniques reduce router area by 36.2% from a traditional buffered design and reduce network power by 54.9% (73.4%) on average, in exchange for 13.6% (1.9%) slowdown, with multiprogrammed (multithreaded) workloads. In particular, slowdown is minimal and savings are significant at low-to-medium load. We thus present a cheap and practical design for a bufferless interconnect – an appealing design point for vastly reduced cost. It is our hope that this will inspire more ideas and further work on cheap interconnect design.

Acknowledgments

We thank the anonymous reviewers of our HPCA-17 paper [15] for their feedback. We gratefully acknowledge members of the SAFARI research group, CALCM, and Thomas Moscibroda at Microsoft Research for many insightful discussions on this and related work. Chris Fallin was supported by a PhD fellowship from SRC, and subsequently NSF, while conducting this work. We acknowledge the support of Intel, AMD, and Gigascale Systems Research Center. This research was partially supported by an NSF CAREER Award CCF-0953246.

References

[1] TPC-C. `http://www.tpc.org/tpcc/`.

[2] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. *ICS*, 2006.

[3] P. Baran. On distributed communications networks. *IEEE Trans. on Comm.*, 1964.

[4] L. A. Barroso et al. Piranha: a scalable architecture based on single-chip multiprocessing. *ISCA-27*, 2000.

[5] K. Batcher. Sorting networks and their applications. *AFIPS Spring Joint Comp. Conf.*, 32:307–314, 1968.

[6] S. Borkar. Thousand core chips: a technology perspective. *DAC-44*, 2007.

[7] C. Busch, M. Herlihy, and R. Wattenhofer. Hard-potato routing. *STOC*, 2000.

[8] T. Chich, P. Fraigniaud, and J. Cohen. Unslotted deflection routing: a practical and efficient protocol for multihop optical networks. *IEEE/ACM Transactions on Networking*, 2001.

[9] D. E. Culler et al. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.

[10] W. Dally. Virtual-channel flow control. *IEEE Par. and Dist. Sys.*, 1992.

[11] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Comp.*, 1987.

[12] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[13] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. *MICRO-42*, 2009.

[14] R. Das et al. Aérgia: exploiting packet latency slack in on-chip networks. *ISCA-37*, 2010.

[15] C. Fallin, C. Craik, and O. Mutlu. CHIPPER: A low-complexity bufferless deflection router. *HPCA-17*, 2011.

[16] C. Gómez et al. Reducing packet dropping in a bufferless noc. *Euro-Par-14*, 2008.

[17] P. Gratz, C. Kim, R. McDonald, and S. Keckler. Implementation and evaluation of on-chip network architectures. *ICCD*, 2006.

[18] A. Hansson, K. Goossens, and A. Radulescu. Avoiding message-dependent deadlock in network-based systems-on-chip. *VLSI Design*, 2007.

[19] M. Hayenga, N. Jerger, and M. Lipasti. Scarab: A single cycle adaptive routing and bufferless network. *MICRO-42*, 2009.

[20] W. Hillis. *The Connection Machine*. MIT Press, 1989.

[21] Y. Hoskote et al. A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro*, 2007.

[22] W. W. Hwu et al. Implicitly parallel programming models for thousand-core microprocessors. *DAC-44*, 2007.

[23] S. A. R. Jafri et al. Adaptive flow control for robust performance and energy. *MICRO-43*, 2010.

[24] C. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-dominated on-chip caches. *ASPLOS-10*, 2002.

[25] J. Kim. Low-cost router microarchitecture for on-chip networks. *MICRO-42*, 2009.

[26] A. Kodi, A. Sarathy, and A. Louri. iDEAL: Inter-router dual-function energy and area-efficient links for network-on-chip (NoC) architectures. *ISCA-35*, 2008.

[27] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[28] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. *ISCA-18*, 1991.

[29] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. *ISCA-8*, 1981.

[30] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. *ISCA-24*, 1997.

[31] Z. Lu, M. Zhong, and A. Jantsch. Evaluation of on-chip networks using deflection routing. *GLSVLSI-16*, 2006.

[32] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI*, 2005.

[33] MathWorks. MATLAB. `http://www.mathworks.com/products/matlab/`.

[34] J. McCalpin. STREAM: sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/`.

[35] G. Michelogiannakis et al. Elastic-buffer flow control for on-chip networks. *HPCA-15*, 2009.

[36] G. Michelogiannakis et al. Evaluating bufferless flow-control for on-chip networks. *NOCS*, 2010.

[37] Microsoft Corporation. Microsoft SharePoint. `http://sharepoint.microsoft.com/en-us/Pages/default.aspx`.

[38] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. *ISCA-36*, 2009.

[39] C. Nicopoulos et al. ViChaR: A dynamic virtual channel regulator for on-chip networks. *MICRO-39*, 2006.

[40] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. Next generation on-chip networks: What kind of congestion control do we need? *Hotnets-IX*, 2010.

[41] K. Olukotun et al. The case for a single-chip multiprocessor. *ASPLOS*, 1996.

[42] J. Owens et al. Research challenges for on-chip interconnection networks. *IEEE Micro*, 2007.

[43] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. *MICRO-37*, 2004.

[44] D. Pham et al. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *J. Solid-State Circuits*, 41(1):179–196, Jan 2006.

[45] A. Rogers et al. Supporting dynamic data structures on distributed shared memory machines. *ACM Trans. Prog. Lang. and Sys.*, 17(2):233–263, Mar 1995.

[46] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *SIGGRAPH*, 2008.

[47] B. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 1981.

[48] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-9*, 2000.

[49] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006`.

[50] Stanford CVA Group. Network-on-Chip project router model. `http://nocs.stanford.edu/`.

[51] Tilera Corporation. Tilera announces the world's first 100-core processor with the new TILE-Gx family. `http://www.tilera.com/news_&_events/press_release_091026.php`.

[52] S. Tota et al. Implementation analysis of NoC: a MPSoC trace-driven approach. *GLSVLSI-16*, 2006.

[53] A. Udipi et al. Towards scalable, energy-efficient, bus-based on-chip networks. *HPCA-16*, 2010.

[54] A. Waksman. A permutation network. *JACM*, 15:159–163, Jan 1968.

[55] H. Wang et al. Orion: a power-performance simulator for interconnection networks. *MICRO-35*, 2002.

[56] H. Wang, L. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. *MICRO-36*, 2003.

[57] S. Woo et al. The SPLASH-2 programs: characterization and methodological considerations. *ISCA-22*, 1995.