Towards the Ideal On-chip Fabric for 1-to-Many and Many-to-1 Communication

Tushar Krishna[†] Li-Shiuan Peh[†] Bradford M. Beckmann[‡] Steven K. Reinhardt[‡]

[†]Department of EECS, MIT
Cambridge, MA
{tushar, peh}@csail.mit.edu

[‡]AMD Research
Bellevue, WA
{Brad.Beckmann, Steve.Reinhardt}@amd.com

ABSTRACT

The prevalence of multicore architectures has accentuated the need for scalable cache coherence solutions. Many of the proposed designs use a mix of 1-to-1, 1-to-many (1-to-M), and many-to-1 (M-to-1) communication to maintain data coherence and consistency. The on-chip network is the communication backbone that needs to handle *all* these flows efficiently to allow these protocols to scale. However, most research in on-chip networks has focused on optimizing only 1-to-1 traffic. There has been some recent work addressing 1-to-M traffic by proposing the forking of multicast packets within the network at routers, but these techniques incur high packet delays and power penalties. There has been little research in addressing M-to-1 traffic.

We propose two in-network techniques, Flow Across Network Over Uncongested Trees (FANOUT) and Flow AggregatioN In-Network (FANIN), which perform efficient 1-to-M forking and M-to-1 aggregation, respectively, such that packets incur only single-cycle delays at most routers along their path, thus approaching an ideal network (one that incurs only wire delay/energy). Full-system simulations on a 64-core CMP with SPLASH-2 and PARSEC benchmarks show that FANOUT and FANIN together reduce runtime by 14.9% and network energy by 40.2%, on average, compared to state-of-the-art networks, operating at just 1% and 9.6% above the runtime and energy of an ideal network.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiprocessors—*Interconnection Architectures*; C.1.4 [**Parallel Architectures**]: Distributed Architectures

General Terms

Design, Performance

*The authors acknowledge the support of NSF CPA-0702341, and the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation entity.

MICRO'11 December 3-7, 2011, Port Alegre, Brazil

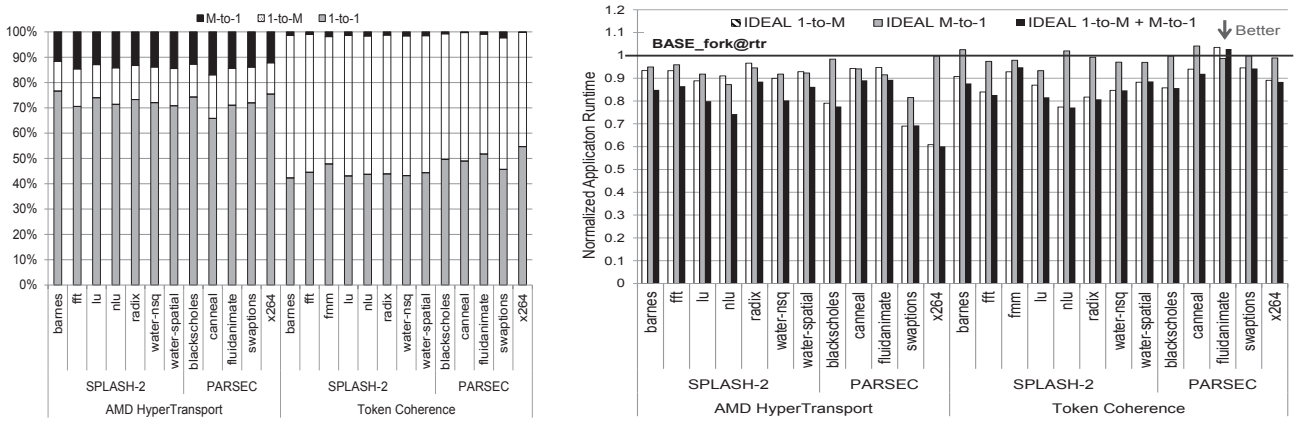
1. INTRODUCTION

As chip multiprocessors scale to higher core counts, designing a scalable on-chip cache subsystem has become a crucial component in achieving high performance. Together, the cache coherence protocol and on-chip network must achieve high throughput, low latency, and low energy.

There are several different types of cache coherence protocols, each of which puts different demands on the network connecting the cores. At one end of the design spectrum are broadcast-based protocols [6, 11, 29, 36]. These designs have the advantage of not requiring any directory storage, but have the limitation of increased network bandwidth demands because all requests and invalidates are broadcast. At the other end, full-bit directory protocols [27, 28] track all sharers, reducing network demand by replacing broadcasts with precise unicasts and multicasts. However, the required storage increases area and energy costs as core counts scale. More scalable directory protocols [1, 9, 12, 26, 33], including commercial designs like Intel's Quick Path Interconnect (QPI) protocols [1] and AMD's HyperTransport™Assist [12] incorporate partial directories to consume less storage than a full-bit directory, and rely on a combination of broadcasts, multicasts, and direct requests to maintain coherence.

For an N-core chip, we classify the communication patterns of protocols as 1-to-1, 1-to-M, and M-to-1 where M refers to multiple sources or destinations ($1 < M <= N$). 1-to-1 communication occurs in unicast requests/responses exchanged between cores. 1-to-M communication occurs in broadcasts and multicast requests [1, 6, 11, 29, 36]. M-to-1 communication occurs in acknowledgements [1, 11] or token collection [29, 33] in protocols to maintain ordering. In the on-chip domain, conventional wisdom seems to dictate that 1-to-M and M-to-1 traffic should be avoided, assuming the on-chip network will not be able to handle such high bandwidth. Instead, the coherence mechanism would involve serialized lookups through multiple caches and directories, adding latency to misses.

This work challenges this conventional wisdom, showing that a network specifically designed to handle 1-to-M and M-to-1 traffic can approach the performance of an *ideal* network, eliminating the need to avoid these patterns in the cache coherence protocol. We define this ideal 1-to-M/M-to-1 network as one in which each 1-to-M/M-to-1 packet incurs only wire delay from its source to destination (i.e., *hops* $\times$ 2 (one-cycle through the crossbar in the router, and another cycle for the link)), and no additional delay/energy because of buffering and waiting due to contention. We propose novel solutions that allow *both* 1-to-M and M-to-1 flows



(a) Message flows in HyperTransport and Token Coherence

(b) Full-system application runtime for ideal 1-to-M/M-to-1 networks normalized against state-of-the-art baseline

Figure 1: Motivation for optimizing on-chip networks to handle 1-to-M and M-to-1 communication. (Refer to Section 5.1 for the methodology and configuration.)

to approach ideal latency (wire delay), energy (wire energy), and throughput (high link utilization). In particular, this work makes a two-fold contribution: Flow Across Network Over Uncongested Trees (FANOUT) and Flow Aggregation In-Network (FANIN).

- FANOUT addresses inefficiencies in current state-of-the-art 1-to-M network designs, via a load-balanced routing algorithm (*Whirl*), a crossbar circuit (mXbar) that forks flits at the similar energy/delay as unicasts, and a flow-control technique for bypassing buffers; to realize single-cycle routers for 1-to-M flows.
- FANIN performs opportunistic aggregation of M-to-1 traffic in a distributed manner, with additional optimizations for synchronized routing (*rWhirl*) and a smart flow-control for waiting at routers, to realize single-cycle routers for M-to-1 flows.

We evaluate FANOUT and FANIN in a full-system 64-core environment with two broadcast-intensive coherence protocols, across SPLASH-2 [3] and PARSEC [8] applications. FANOUT reduces network latency of multicasts by 39.5%; FANIN aggregates 93.5% of acknowledgements. Together, FANOUT and FANIN reduce application runtime by 14.9%, and network energy by 40.2%, which are just 1% and 9.6% higher than the runtime and energy of an ideal network.

Section 2 of this paper motivates our work and discusses relevant related work. Section 3 and Section 4 describe FANOUT and FANIN. Section 5 presents our evaluations with full-system simulations, and Section 6 concludes.

2. MOTIVATION

To motivate our investigation, we first analyze message flows for several multi-threaded workloads. Fig. 1(a) shows a breakdown of the percentage of 1-to-1, 1-to-M and M-to-1 flows[1] in the network across SPLASH-2 [3] and PARSEC [8] benchmarks for Token Coherence [29] (a snoopy coherence protocol), and HyperTransport [11] (a stateless directory coherence protocol) in a 64-core CMP. For HyperTransport, 1-to-M requests and M-to-1 responses form 14.3% and 14.1% of injected messages on average, respectively, with M = 64 in both cases. Token Coherence reduces M-to-1 traffic to 2%, with M = 12 on average, at the cost of a higher percentage (52.4%) of 1-to-M traffic (M = 64). These observations point to the criticality of the network fabric connecting the cores to efficiently handle *all three* kinds of communication, and not become a bottleneck.

Most research in on-chip networks has concentrated only on optimizing 1-to-1 flows. The other two flows end up being realized the naive way: 1-to-M results in M unicast packets being sent from the source network interface controller (NIC), and M-to-1 results in M unicast packets being received by the destination NIC. Both these approaches (1) create heavy congestion at the link from (to) the source (destination) NIC, creating hot-spots, and (2) flood the network due to the bursty nature of these messages, loading some links M times over their capacity of 1 flit[2] per cycle, leading to high contention and a dramatic rise in packet latency and corresponding penalties in throughput and energy.

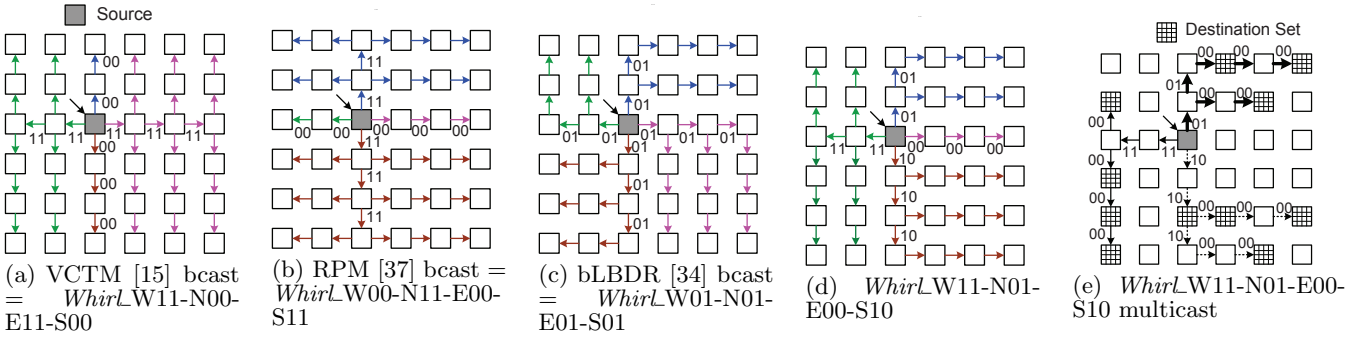
VCTM [15] identified this problem for 1-to-M traffic, and there have been recent works [15,16,34,35,37] with solutions to mitigate it. While these differ in terms of routing algorithms, target systems, and the scale of M, in essence they propose routers with the ability to fork flits (i.e., a single multicast packet enters the network, and multiple flits are replicated and sent out of each output port towards their destinations at intermediate routers). We collectively term these works as "fork@rtr".

The opposite problem with M-to-1 traffic has not been dealt with yet in the on-chip domain. In addition to coherence, M-to-1 flows occur in barrier synchronization [18], the reduce phase of MapReduce [14], and so on. Previous work has focused on specialized solutions for accelerating barrier messages [4, 10, 18, 22, 39] by relying on an ordered FAT-tree topology in the off-chip domain [18] and adding additional wires and registers to track barriers in the on-chip domain [4,10,22,39]. Unfortunately, none of these solutions are generic enough to be applied for other M-to-1 traffic across an unordered, distributed on-chip network topology such as a mesh that is commonly used in multicore chips [20, 40].

Current state-of-the-art fork@rtr designs [15,16,34,35,37], while better than a network with no multicast support by

[1]Every 1-to-M and M-to-1 flow translates to M messages in the network.

[2]Flits are subcomponents of a packet.



(a) VCTM [15] bcast = *Whirl*∟W11-N00-E11-S00
(b) RPM [37] bcast = *Whirl*∟W00-N11-E00-S11
(c) bLBDR [34] bcast = *Whirl*∟W01-N01-E01-S01
(d) *Whirl*∟W11-N01-E00-S10
(e) *Whirl*∟W11-N01-E00-S10 multicast

Figure 2: Possible broadcast trees. *Whirl's algorithm: Packets fork into all four directions at the source router. By default, every packet continues straight in its current direction. In addition, forks at intermediate routers are encoded by LeftTurnBit, RightTurnBit, where left and right are relative to the direction of traversal. These bits are reset to 0 once a turn completes (hence, 0 is implicit on all unmarked arrows).*

25-50% [15, 34, 37], are still far from the best that an on-chip network could achieve for 1-to-M and M-to-1 traffic. In Fig. 1(b) we compare the full-system application runtime for 64-threaded SPLASH-2 [3] and PARSEC [8] benchmarks using the ideal networks defined in Section 1 against the baseline fork@rtr designs. We observe that the fork@rtr design is 13% slower than ideal 1-to-M, 7% slower than ideal M-to-1, and 20% slower than ideal 1-to-M + M-to-1, on average, for HyperTransport. For Token Coherence, fork@rtr is 12-13% slower on average than the ideal networks[3].

In the rest of the paper, we introduce network optimizations to bridge these performance gaps.

3. FANOUT: FLOW ACROSS NETWORK OVER UNCONGESTED TREES

This section describes the FANOUT design. We start with a load-balanced 1-to-M routing algorithm we call *Whirl*. We then describe our crossbar circuit, mXbar, which forks multicast flits within a cycle, consuming energy similar to unicasts. Finally, we discuss flow-control optimizations to design a single-cycle multicast router.

3.1 *Whirl*: Load-balanced 1-to-M Routing

3.1.1 Background

Multicast packets are typically routed in a path-based or tree-based manner. In path-based routing, a multicast packet is forwarded sequentially from one destination to the next. For multicasts with many destinations, and for broadcasts, this leads to the packet traversing a logical ring embedded in the network, and forking out to the NIC at each destination router. While this places the minimum load of 1 flit per cycle on each link, it results in extremely high latency for the destinations at the end of the ring, and is thus not a scalable solution.

Tree-based routing creates virtual multicast trees in the network, and are used in most prior works [15, 16, 34, 35, 37]. However, a major limitation of all these schemes is that their various multicast trees reduce to *one* tree in the presence of broadcasts or multicasts with many destinations (i.e., any node that broadcasts ends up using the same tree structure for distributing the broadcast). The result is links are utilized in an unbalanced manner, lowering throughput. This is because as the broadcast moves through the network, it forks at intermediate routers based on *fixed* output port priorities to avoid duplicate reception of the same packet via alternate routes[4]. Fig. 2 shows the tree structures that all broadcast flits would use in some of these works, based on the output port priorities specified in their designs. For broadcasts from uniformly distributed sources in an 8x8 mesh, we observed that VCTM [15] uses X-links 11% and Y-links 89% of the time, while RPM [37] uses X-links 89% and Y-links 11% of the time. Increased load causes congestion, which adds delay and worsens throughput.

To the best of our knowledge, there has been no routing scheme that targets broadcasts/dense multicasts, and achieves ideal load balance.

3.1.2 Whirl

We propose *Whirl*: a tree-based routing scheme that (1) balances link loads for broadcasts and dense multicasts, (2) ensures non-duplicate packet reception, (3) is non-table-based, and (4) is deadlock-free.

Whirl parameterizes the entire space of possible broadcast trees, some of which are shown in Fig. 2, and randomly selects one tree on each broadcast/multicast to balance the link loads. The trees from VCTM [15], RPM [37], and bLBDR [34] form a subset of *Whirl*'s 16 broadcast trees. For multicasts with few destinations, our approach and previous approaches yield similar results, but as the destinations increase, our approach outperforms previous approaches due to more path diversity, and thus lower contention.

Whirl encodes the routing information for every broadcast/multicast packet in two bits: the LeftTurnBit (LTB) and the RightTurnBit (RTB). These tell the router whether the flit should turn[5] left or turn right *relative to* its current direction of motion. For instance, for a flit going West, left is South and right is North. The (LTB, RTB) pairs for each direction together create the global *Whirl* broadcast tree.

[3]The fork@rtr designs perform better than the ideal in some cases, since the faster ideal network could end up speeding up invalidations of shared locks, increasing cache misses.

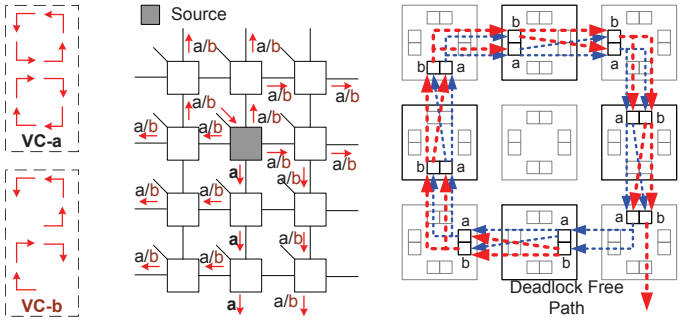
[4]For instance, in RPM [37], broadcasts are distributed via the tree structure shown in Fig. 2(b). For delivering the broadcast in the NE quadrant, the routers to the North of the source fork the broadcast flit to their East, while those on the East of the source do not fork the flit. This ensures that only one copy of the flit is delivered to all nodes.

[5]A turn here implicitly implies a fork in a broadcast scenario as the flit also continues straight.

LTB: LeftTurnBit
RTB: RightTurnBit

At every router:
(1) *continue* STRAIGHT
(2) *fork* into Network Interface
(3) **if:** LTB is high **then:** *fork* LEFT
(4) **if:** RTB is high **then:** *fork* RIGHT
(5) *clear (LTB,RTB)* in flits that forked left/right.

(a) *Whirl* pseudo-code.



(b) Deadlock avoidance by VC partitioning: VC-b implements a deadlock-free South-last turn model and acts as an escape VC.

Figure 3: Features of *Whirl*.

Choosing the *Whirl* broadcast tree. The global *Whirl* route for a packet is decided by the source NIC (sNIC, i.e., source routing). This is done not only to balance the load, but also to ensure non-duplicate and guaranteed reception of packets at all destinations' NICs; which is hard to support if the routers dynamically decide which route to take. The sNIC randomly chooses four bits: LTB_W, LTB_N, LTB_E, and LTB_S, which are the LeftTurnBits for each direction W, N, E, and S. The sNIC sends these four bits to the source router in the multicast packet.

Implementing forking using LTB and RTB. At the source router, the RTBs for each direction are computed from the four LTBs as follows: $RTB_S = \sim LTB_W$, $RTB_W = \sim LTB_N$, $RTB_N = \sim LTB_E$, $RTB_E = \sim LTB_S$. This rule enforces that duplicate copies of the same packet do not reach a node via different directions. For instance, in Fig. 2(c), $RTB_N = 1$ and $LTB_E = 0$ to ensure non-duplicate delivery in the NE quadrant. The flit is then forked out of all four output ports, with each copy carrying the corresponding (LTB, RTB). At all further routers, the routing algorithm that is followed is shown in Fig. 3(a). After the flit turns once, no further turns are allowed, hence the (LTB, RTB) are reset to 0. This is done for simplicity, and to implement deadlock freedom, which we discuss later in this section.

Throughput characterization. Packets traversing a combination of *Whirl*'s 16 broadcast trees use all possible links that lie along the minimal routing path. For broadcast-only traffic from uniformly distributed sources, simulations showed 50% utilization on both the X and Y links, demonstrating ideal load balance.

Deadlock avoidance. *Whirl* allows all turns except U-turns, and thus requires a deadlock avoidance mechanism. We do not wish to restrict any turns and take away the ideal load balancing benefits of *Whirl*'s throughput discussed earlier. We thus apply conventional Virtual Channel[6] (VC) management to avoid deadlock, as shown in Fig. 3(b). We partition the VCs into two sets, VC-a and VC-b. Packets are enforced to allocate only VC-a in the South direction, before they turn. They can allocate both VC-a and VC-b along the other directions, and after turning. Since all packets can only make one turn in *Whirl*, this restricts S-to-E and S-to-W turns within VC-b, implementing a deadlock-free South-last turn model [13]. Because the multicast tree can be decomposed into unicast paths, VC-b acts like an escape VC [13][7]. Fig. 3(b) shows an example scenario with all flits in VC-a in a circular dependency. However, VC-b will always have an escape path, and the flits in VC-a will eventually drain out via VC-b.

Another cause of deadlock in multicast networks is when two copies of the same flit take two alternate paths to reach the same destination. This can never occur in *Whirl* because of the LTB/RTB rules.

Point-to-Point Ordering. Multiple *Whirl* routes from the same sNIC can violate point-to-point ordering from source to destination. For coherence and other on-chip communication protocols that rely on this ordering, such as persistent requests in Token Coherence [29], sNICs statically assign only one of the *Whirl* trees, based on cache-block address, to all messages within an ordered virtual network/message class. Routers follow FIFO ordering for flits within an ordered virtual network, by using queueing arbiters for switch allocation, thereby guaranteeing point-to-point ordering.

Pruning the tree for multicasts. For multicasts (in which not all NICs are in the destination set), flits need to carry their destination set with them, and *Whirl*'s algorithm described in Fig. 3(a) can be modified such that flits do not continue/fork if no destination exists among the nodes reachable by that direction. As an example, Fig. 2(d) sketches a *Whirl* broadcast tree, and Fig. 2(e) shows its trimmed version for a multicast to 11 destinations. We do not show implementation details here due to space constraints.

3.2 mXbar: Router Microarchitecture for Forking

3.2.1 Background

Multicast routers fork flits out of multiple ports. This can be done either by (1) reading the same flit out of the buffer every cycle and sending it out of each output port one by one upon successful allocation, or by (2) reading the flit out of the buffer once and forking it within the crossbar.

The first approach adds serialization delay to multicast flits, increases buffer occupancy (thereby lowering throughput), and consumes high buffer read energy. However, it can use a simpler crossbar circuit that need not fork flits. VCTM [15] and MRR [16] use this technique[8].

[6]Input buffers at routers are typically divided into multiple virtual channels to avoid packets going out of one port getting blocked by packets going out of another port.

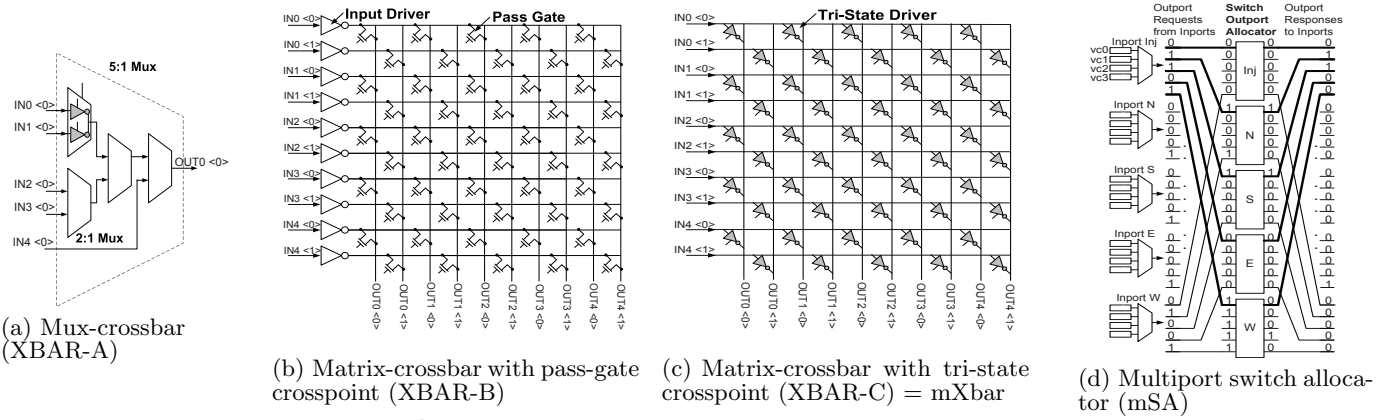
[7]The escape VC [13] concept proves that as long as packets are allowed to allocate any VC, the sufficient condition to break deadlocks is to have one VC enforce a deadlock-free route while all other VCs can permit all turns.

[8]In fact, MRR does not use a crossbar. It rotates flits across buffers at different output ports.

Table 1: Energy-Delay comparison for 5x5 128-bit crossbars, modeled using Orion 2.0 [21], at 45nm

Xbar	Type	Transistors	1-to-M Delay	1-to-M Energy	1-to-M Router Energy
A	Mux-based	240/bit	1	$221\times M$ fJ	$E_{wr} + E_{rd} + M\times E_{xb} = 117 + 221\times M$ fJ
B	Mat. + PassGate	25/bit	M	$48\times M$ fJ	$E_{wr} + M\times E_{rd} + M\times E_{xb} = 63 + 102\times M$ fJ
C	Mat. + TriState	150/bit	1	$65\times M$ fJ	$E_{wr} + E_{rd} + M\times E_{xb} = 117 + 65\times M$ fJ

$E_{wr/rd}$ = Energy for buffer write/read, E_{xb} = Energy for xbar 1-to-1 traversal



(a) Mux-crossbar (XBAR-A)

(b) Matrix-crossbar with pass-gate crosspoint (XBAR-B)

(c) Matrix-crossbar with tri-state crosspoint (XBAR-C) = mXbar

(d) Multiport switch allocator (mSA)

Figure 4: Crossbar switch circuits and allocator microarchitecture.

The second approach removes serialization delay, but requires a crossbar that performs forking. Samman et al. [35] and RPM [37] use mux-based crossbars to realize this. Mux-based PxP crossbars consist of a P:1 mux at each output port, as shown in Fig. 4(a), and are the default designs generated from RTL synthesis. We call this XBAR-A. Because each input fans out to each of these muxes, this design can inherently fork flits out of multiple output ports. In contrast, a conventional matrix-crossbar is laid out as a regular matrix, and uses pass-gates at crosspoints [38] to implement the switching action, as shown in Fig. 4(b). It relies on an input driver to drive (charge/discharge) *both* the horizontal and vertical wires of the crossbar. We call this XBAR-B. This design *cannot* efficiently fork flits because the driver cannot drive one horizontal *and* P vertical wires within a cycle[9]. The caveat, however, is energy. Crossbars are known to be one of the most power-consuming components of a router [20]. Each P:1 mux in XBAR-A is typically realized using a cascade of smaller 2:1 muxes, as shown in Fig. 4(a), which increases energy consumption tremendously because many more transistors are used than in XBAR-B, as shown in Table 1. In addition, matrix-crossbar XBAR-B can segment wires [38] to drive only the required portion of the wires, saving more power.

We modeled these crossbars with five inputs/outputs in Orion 2.0 [21] at 45nm, targeting a 2GHz clock. Table 1 compares the delay/energy to perform a 1-to-M fork within a router. XBAR-A consumes 4.6X more energy than XBAR-B, and the corresponding router consumes 2.1X more energy even for a unicast (using $M = 1$ in the last column of Table 1), making it an impractical design to use. RTL-synthesized crossbars are not the only option in real designs; Intel's 80-core NoC [20] uses custom layouts for the crossbars to exploit regularity and reduce power.

[9]To fork flits, the input driver would have to be made about P times bigger, which would proportionally increase power consumption and become overkill for unicasts.

3.2.2 mXbar: Multicast Crossbar

We propose a crossbar circuit, XBAR-C, that uses a matrix-crossbar layout but has tri-state drivers at crosspoints instead of pass-gates, as shown in Fig. 4(c). The advantage of this design is that each output wire gets an independent driver, like XBAR-A, and can thus support forking of flits within a cycle. We thus trade the area advantage from XBAR-B by adding more transistors for higher drive-ability. The matrix-crossbar design still gives us the layout regularity and wire segmentation [38] advantages relative to the mux-based design. Table 1 shows that XBAR-C achieves a single-cycle delay like XBAR-A, while the router energy for forking flits with XBAR-C is 1.1X-0.8X the router energy of XBAR-B for 1-cast to 4-casts. XBAR-C is thus a practical design for forking flits, and we use it in the FANOUT router, referring to it as mXbar in the rest of the paper.

3.2.3 mSA: Multiport Switch Allocation

We show the design of our multiport switch allocator in Fig. 4(d), which enables an input port to gain access to multiple output ports of the mXbar in the same cycle. In this example, inport *Inj* requests outports *N*, *S*, and *W*, and is granted *N* and *S*. At the end of mSA, the winner of an output port is granted a free VC for the next router from a queue of free VCs [23].

3.3 Single-cycle FANOUT Router

3.3.1 Background

Single-cycle router pipelines have been proposed in the past [24, 25, 31] for unicast flits. In these designs, the router pipeline on the critical path reduces to just Switch Traversal (ST). The basic idea in these designs is to pre-allocate the crossbar switch before the actual flit arrives, to give it a direct access to the crossbar, thereby *bypassing* the buffering stage. We attempt to design such a pipeline for multicast flits. To the best of our knowledge, no prior research has attempted to extend buffer bypassing for multicasts, which

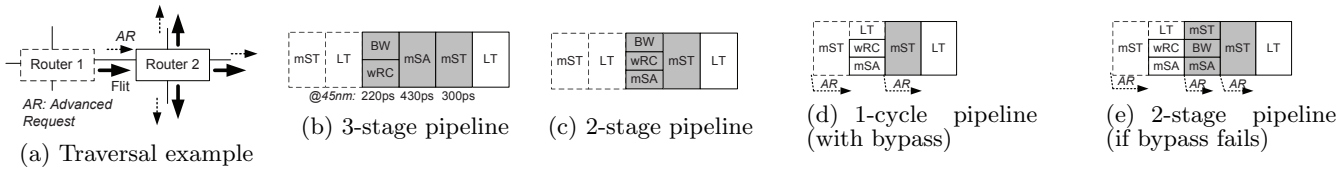


Figure 5: FANOUT pipeline optimizations to realize single-cycle multicast router. *The flit pipeline at Router 2 is shaded in gray and that of preceding Router 1 is outlined with dashed lines. The stages are BW: buffer write, wRC: whirl route computation, mSA: multiport switch+VC allocation, mST: mXbar switch traversal, LT: link traversal, AR: advanced request.*

is essential for meeting the delay/energy limits of an ideal broadcast network.

3.3.2 Pipeline Stages

We start by enumerating the various pipeline stages in the FANOUT router, and then start folding stages over each other to ultimately result in a single pipeline stage for multicast flits, as highlighted by Fig. 5. This FANOUT pipeline is for the request message class[10] of the network. Response messages use the FANIN pipeline (Section 4.5).

Step 1: Original pipeline (Fig. 5(b)). The unoptimized FANOUT router pipeline is the same as a baseline fork@rtr design, though the actual components (routing algorithm, switch allocation algorithm, and crossbar circuit) differ. At Router 1, the flit goes through the switch (mST) and link (LT) to arrive at Router 2. It gets buffered (BW), performs routing (wRC) in parallel, then places a request for multiple ports of the switch (mSA), forks through the switch (mST), and traverses the link (LT) to the next router. The critical path delays for each are shown in Fig. 5(b), obtained from RTL implementation of the FANOUT router in 45nm and synthesizing for a 2GHz clock.

Step 2: Lookahead routing (folding mSA and wRC) (Fig. 5(c)). The multiport switch allocation (mSA) cannot be performed until the output port requests for the flits are known, which are only available after *Whirl* route computation (wRC). We leverage lookahead routing [17] and perform wRC one hop in advance (i.e., the wRC at the current router determines the output ports at the *next* router, allowing an incoming flit to place a request for the switch as soon as it arrives). Thus BW, wRC, and mSA can all be done in parallel. However, performing wRC for the *next* router is not as trivial as in a unicast case because multicast flits could have *multiple* next routers if they are forking at the current router. To perform routing one hop in advance, the current router needs to perform wRC for *all* output ports out of which the flit will fork. Thus in a 5x5 router, every input port needs to maintain four wRC blocks, one for each output port assuming no u-turns. The power and area overhead for these 20 *Whirl* blocks was found to be less than 1% of that of the total router because it is very simple combinational logic (Fig. 3(a)). The output port request generated by the *Whirl* block for output port A is embedded in the corresponding flit going out of port A.

Step 3: Bypassing (wRC and mSA before flit arrival) (Fig. 5(d)). We can shrink the flit pipeline at the router to one cycle if we perform wRC and mSA before the flit arrives. To do so, we must examine what information is required by wRC and mSA. wRC needs to know the output ports out of which the flit will fork (5-bit vector) to activate the corresponding *Whirl* blocks at the input port. It also needs to know the 2-bit 'LTB,RTB' to determine the route. mSA needs to know only the output ports request. To realize this pipeline, the flit at Router 1 sends these 7 bits as an advanced request[11] (AR) to Router 2, while it traverses the mXbar (mST) at Router 1. This enables Router 2 to perform wRC and mSA while the flit performs link traversal. If mSA at Router 2 is successful in granting all output ports to the flit, it does not get buffered and uses the single-cycle pipeline in Fig. 5(d). This allows FANOUT to eliminate the buffer write and read energy, shown earlier in Table 1, from the router traversal. The mXbar is critical for achieving this one-cycle pipeline; otherwise a multicast flit will be forced to get buffered and spend multiple cycles to go out one by one. If mSA grants only a subset (or none) of the the ports, the flit gets buffered and starts mSA for the remaining ports, as shown in Fig. 5(e).

A flit that performs mST (either via bypassing or from the buffers) needs to send ARs out of all output ports that it will fork out from, as shown in Fig. 5(a). These advanced request bits are ready at the end of wRC and mSA.

3.4 Ideal 1-to-M Traversal

In summary, *Whirl* sets up load-balanced paths for multicast flits to lower congestion, while the mXbar and advanced requests together allow flits to perform single-cycle forking at routers without getting buffered, thereby incurring only wire (mST, LT) delay/energy from the source to all destinations and meeting our definition of an *ideal* traversal.

4. FANIN: FLOW AGGREGATION IN-NETWORK

In this section we present FANIN, our approach to push the energy-delay-throughput of aggregating M-to-1 messages in the network towards the ideal.

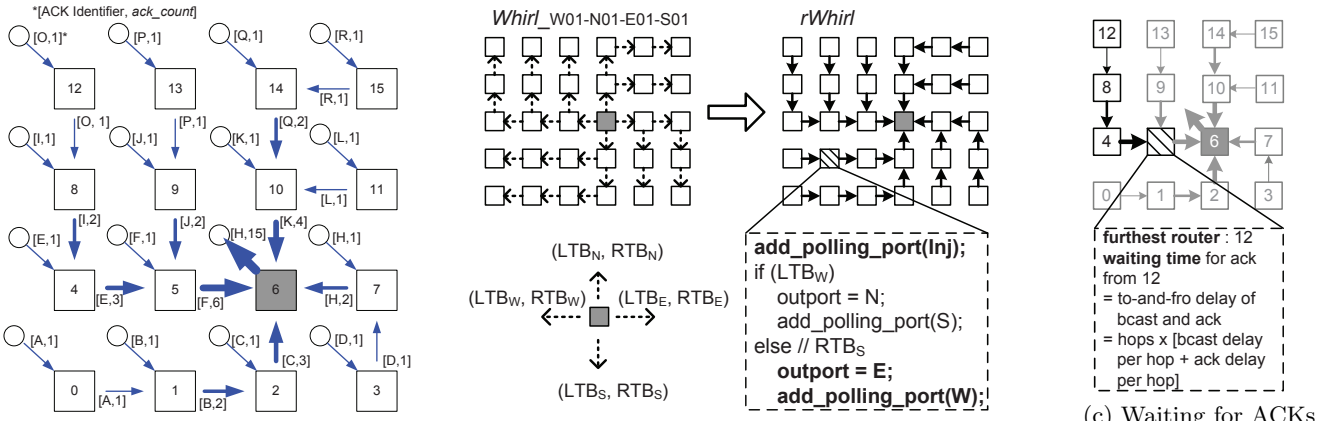
We will use the term "M:1" to refer to the communication flow in which M cores generate one response message each for the same destination core and memory address, acknowledging the preceding multicast. For convenience, we will use the term ACK for each individual flit in the M:1 flow, though in principle it is not restricted to just acknowledgements and can work for tokens, barriers, etc.

ACKs for an M:1 flow are aggregated at network routers, as they move towards their common destination. Each ACK flit carries a log(N) bit *ack_count*, where N is the maximum number of ACKs that could be received[12]. In addition, a single-bit *ack_bit* is set if the flit is an ACK. To aggregate

[10]The router buffers are partitioned into separate virtual message classes such as requests, responses, etc. to avoid protocol-level deadlocks.

[11]These bits sent in advance were called lookaheads in [24,25] and advanced bundles in [23].

[12]Response flits in certain protocols like Token Coherence [29] already carry such a counter, because cores can respond with multiple tokens at a time.



(a) ACK Aggregation Example (b) *rWhirl* with sample pseudo-code (c) Waiting for ACKs

Figure 6: Flow AggregatioN In-Network (FANIN)

ACKs, one of the ACK flits is dropped, and its *ack_count* added to the *ack_count* of the other flit.

4.1 Background

M-to-1 communication flow occurs in shared memory coherence protocols, in the form of acknowledgements [1, 11] or tokens [29, 33], and in message passing domains, such as barrier synchronization [4, 10, 18, 39]. There has been no on-chip solution to tackle the former, to the best of our knowledge. In the past, MIMD machines like IBM RP3 [32] and NYU Ultracomputer [19] used network switches to combine memory requests (loads/stores/fetch-and-add) to the same memory location and added extra buffers to track the responses. More recently, aggregation via the network fabric has been researched for implementing barrier synchronization [4, 10, 18, 22, 39]. These proposals essentially implement a wired OR, either by relying on an ordered FAT-tree topology in the off-chip domain, such as in IBM Blue Gene/L [18], or 1-to-M connectivity via on-chip global broadcast wires [4, 22], or all-to-all connectivity between special-purpose registers among a cluster of nodes [10, 39]. These works also add tables to track barriers, thus placing a limit on the number of active barriers at any point in time and adding area/power overheads. Our FANIN is much more general purpose because we target an unordered, distributed on-chip network with any kind of M-to-1 control flow (acknowledgements, tokens, barriers), without adding dedicated 1-to-M wires or extra storage structures.

4.2 Walk-through Example

In Fig. 6(a) we illustrate how ACK O injected from NIC 12 with an *ack_count* of 1 merges with ACK I (which is thus dropped), then merges with E, F, and finally H. H is sent up to the NIC with an *ack_count* of 15, instead of the NIC having to wait for individual ACKs from all the 15 senders.

Realizing this *ideal* aggregation of 15 flits into 1 is non-trivial. The reason is that ACKs are generated by the different cores at *different* times and take a *different* number of cycles to reach the next routers, during which time other ACKs for the same M:1 flow might have already left that router. The rest of this section describes how FANIN solves these issues to realize an ideal M-to-1 aggregation network.

4.3 *rWhirl*: Synchronized Routing

We first ensure that all ACKs for a particular M:1 flow follow a synchronized route. This enables ACKs at intermediate routers to know (1) which ports they need to poll for other ACKs, and (2) when all possible aggregations at that router complete. For instance, in Fig. 6(a), the ACK at the injection port of Router 5 only needs to poll input ports West and North; once it merges ACKs from both directions, it can move ahead. It does not need to wait for an ACK from the South port because ACK B from Router 1 will reach the destination via Router 2, and not Router 5, in this particular routing policy.

While a fixed route (like XY) by all ACKs serves this purpose, it would result in heavy congestion across the Y links leading to the destination because the destination would become a hot-spot node. Instead, we make all ACKs for an M:1 flow follow the *reverse* path of the 1-to-M *Whirl* route they were on, as shown in Fig. 6(b). We term this as *reverse Whirl* or *rWhirl*. This is realized by embedding the received *Whirl* route (4-bit $LTB_W, LTB_N, LTB_E, LTB_S$) from the broadcast into the response flit. Each router on the response path can now decode these bits to compute the 'LTB, RTB' for each direction for the original broadcast and estimate the output port for the ACK, as shown in Fig. 6(b)[13].

Deadlock avoidance. *rWhirl* allows all possible turns, and thus requires a deadlock avoidance mechanism. We avoid deadlocks by using the same VC partitioning technique as we did for *Whirl* (see Fig. 3(b)). All ACKs that start going South are forced to use VC-a, and not VC-b, until they turn, after which they can use any VC. ACKs going in other directions can use any VC. This implements a deadlock-free South-last turn model within VC-b, and guarantees deadlock freedom.

4.4 FANIN Flow Control/Protocol

4.4.1 *master ACKs*

Who does the aggregation? The first ACK that arrives at a router is responsible for aggregating ACKs for that M:1 flow at that router, and we call it the *master* ACK. In most cases, the first ACK to arrive at a router will be at the injection port. This is because any multicast that went through

[13]In certain protocols such as HyperTransport [11], the 1:M source and the M:1 receiver are not the same. This is not a problem because *rWhirl* essentially determines a load-balanced and synchronized route for the ACKs. They do not have to follow the exact reverse route of the broadcast.

this router would have delivered the multicast flit to the local NIC earlier than delivering it to the neighbor's NIC, and consequently the response ACKs would follow that order. Exceptions to this could occur due to congestion at cache controllers and cores. This master ACK gets buffered on arrival. In parallel, it determines which input ports it needs to poll based on the *rWhirl* route and the router's location relative to the destination, as shown in Fig. 6(b).

How is the aggregation done? The master ACK flit checks the incoming links at its polling ports every cycle. It does not poll flits already buffered in the router, which will be explained later. Whenever a new ACK arrives at the router (indicated by the *ack_bit*) at an input port, its *ack_id* (see Section 4.4.2 for details) is compared against all master ACKs (from different M:1 flows) polling this input port. On a match, the master ACK updates its *ack_count*, and the flit that arrived is simply dropped. Dropping the flit entails sending a credit back to the upstream router for the VC it was going to be buffered in.

What happens to polling ports after aggregation? Once the ACK flit aggregates a flit from an input port, it does not remove that port from its polling list to account for inefficient aggregation at upstream routers. For instance, in Fig. 6(a), if Router 8 failed to aggregate ACK O, both ACKs O and I would arrive at Router 4 from the North port, so Router 4 should continue polling this port.

Can there be multiple masters for the same M:1 flow at a router? If an ACK became a master, it means there is no other ACK for the same M:1 flow at this router, else there would have been an earlier master that would have aggregated and dropped it on arrival. Thus, master ACKs do not need to poll buffered flits within the router. There is, however, the corner case of two ACKs arriving in the same cycle from different input ports, in which case they would both become master ACKs if no other master ACK was looking out for them. To handle this special case, we (a) make each input port store the VCid of the ACK flit that arrived in the previous cycle, and (b) give ports an arbitrary static priority: $Inj > W > N > E > S$. In the first cycle after getting buffered, the master ACKs check the last arrival VCs at their polling ports (in addition to polling the links). If they find a match, the master ACK with higher priority aggregates the ACKs with lower priorities, and the latter VCs at the respective ports are made free.

How long should master ACKs wait at a router? Here, we describe a solution for scenarios in which every node (except the requester) responds with an ACK [1, 11]. For cases when this is not true, such as multicasts with few destinations or certain coherence protocols [26, 29], an opportunistic aggregation by master ACKs, with no explicit waiting is a better alternative.

The time of arrival of a particular ACK at an intermediate router depends on the location of this router relative to the source of the ACK. The master ACK at a router should wait to aggregate the ACK from the router furthest from it, before it proceeds. Fig. 6(c) shows the heuristic we use to compute this waiting time: the time for the ACK from Router 12 to arrive at Router 5 will be greater than or equal to the zero-load, to-and-fro delay of the preceding broadcast, and the current ACK (i.e., $hops \times (2 + 2)$), assuming two cycles per hop[14] for the broadcast and ACK flits.

[14]See Section 3.4 and Section 4.6.

The ACK from the local NIC should be the master ACK for the waiting time heuristic to hold. If an ACK arrives from some other port and becomes the master ACK, this means the NIC ACK has not yet arrived or has already left after waiting. This breaks the heuristic's assumptions, so there is no point for this new ACK to wait. The master ACK at the injection port, on arrival, computes $furthest\_hops$, the number of hops between the current router and the furthest router reachable via its polling ports via minimal hops (which is Router 12 in Fig. 6(c)). The furthest router would be at the corner, and is thus easy to compute using the current coordinates and polling ports. It then waits for $furthest\_hops \times 4$ before starting switch allocation. The master ACKs at all other ports do not wait, and instead perform opportunistic aggregation by polling input links every cycle until they get to use the switch.

The policy of making ACKs wait at routers, while increasing the chances of aggregating other ACKs (thereby reducing traffic), offers a trade-off because waiting ACKs occupy router buffers, throttling new flits from entering the router. Moreover, inefficient waiting could lead to a delayed completion of the preceding request. We evaluate our heuristic in Section 5.2.

4.4.2 Comparison Logic for Aggregation

Two ACKs belonging to the same M:1 flow are identified by identical destinations (6 bits in 8x8 mesh) and memory addresses (32-40 bits). Comparing 38-46 bits at each router at multiple ports is an overkill in terms of area and power, and not very scalable. Hashing the address to fewer bits adds the risk of conflicts during aggregation.

We solve this by leveraging the fact that at any point in time, the number of unique M:1 flows is limited by the number of outstanding multicasts, which in turn is limited by the size of the MSHR at each multicasting cache/directory controller. In our design, each multicasting controller[15] maintains a pool of multicast ids called *m_id*s. Every time a new multicast is sent, it is assigned a unique *m_id* and the controller marks this *m_id* as busy. The number of busy *m_id*s at the controller represents the number of multicast requests for which responses have not yet been received.

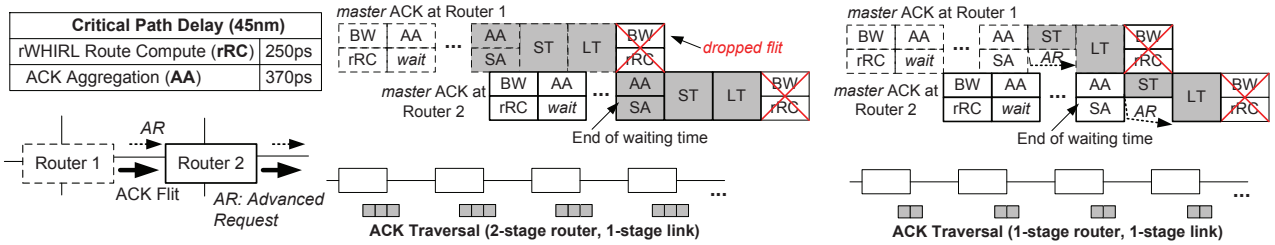
On receiving the multicast, the responding controllers embed the same *m_id* in the ACKs. This ensures that all ACKs belonging to an M:1 communication will have the same $ack\_id = [dest\_id, m\_id]$ and thus this field can be compared for aggregation instead of addresses.

When the multicasting controller receives all expected ACKs, or an unblock[16], it frees the corresponding *m_id*, and can reissue it to future multicasts. Thus at any point in time the *ack_id* is unique to a particular M-to-1 communication flow.

The maximum number of unique *m_id*s required at each multicasting controller is equal to the number of MSHR entries at the controller. In a 64-core CMP, the *dest_id* is 6 bits, while the typical number of MSHR entries is less than 32 [1, 11], giving an *m_id* of 5 bits. This results in an *ack_id* of 11 bits. We can also choose to have fewer *m_id*s than the number of MSHR entries (to reduce the *ack_id* bits further). In this case, if all *m_id*s are busy, and the multicasting controller needs to send out a new multicast, it assigns it an

[15]This could be the requester [1,29] or the home node [11,26].

[16]In protocols like HyperTransport [11], all ACKs go to the requester, which then sends an unblock message to its home node, which is the multicasting controller.



(a) Traversal example (b) Regular FANIN pipeline (c) Optimized FANIN pipeline

Figure 7: FANIN pipeline optimizations to realize single-cycle aggregation router. *The critical cycles (adding to overall traversal delay) of the pipelines are shaded in gray. The stages are: BW: buffer write, rRC: rWhirl route computation, AA: ACK aggregation, SA: switch allocation, ST: switch traversal, LT: link traversal, AR: advanced request.*

m_id of -1. ACKs with *m_id* of -1 are not aggregated in the network, thus maintaining correctness[17].

4.5 Single-cycle FANIN Router

FANIN's regular pipeline is shown in Fig. 7(b), along with the synthesized critical path delays for the *rWhirl* (rRC) and ACK aggregation (AA) steps. We discuss Router 2 without any loss of generality. We define critical stages as the pipeline stages that add to the number of cycles per hop for the aggregated ACKs. The master ACK at Router 2 always needs to get buffered because it performs the aggregation and needs to wait for incoming ACKs. However, this part of the pipeline is overlapped by the flit arrival from Router 1 for which this master ACK is waiting, and is thus non-critical. Once the flit from Router 1 arrives, it gets aggregated and dropped, and the master ACK at Router 2 starts switch allocation (SA). It then performs switch traversal (ST). These two router pipeline stages are critical. We leverage bypassing, sending an advanced request (AR) ahead of the regular flit, similar to Section 3.3.2, to shrink the number of critical stages in the router to one, as shown in Fig. 7(c). The AR in this scenario needs to carry the 11-bit *ack_id*, 1-bit *ack_bit*, and 6-bit *ack_count*. This AR can perform the aggregation, while the actual flit traverses the link (and is then dropped the next cycle on aggregation). This optimized ACK traversal is shown in Fig. 7(c) and presents two critical stages per hop for ACKs (one cycle in router, one in link). This FANIN pipeline is used for the response message class of the network. The request message class will follow the FANOUT pipeline described earlier in Section 3.3.2.

4.6 Ideal M-to-1 Traversal

In summary, FANIN enables ACKs from the four quadrants of the chip to reach the destination router in a synchronized manner (using *rWhirl*) as four aggregated ACKs that incurred 1-cycle router and 1-cycle link critical delays at all intermediate hops (using intelligent waiting and advanced requests). At the destination, these ACKs get opportunistically aggregated into one ACK and proceed to the NIC, thus reducing network load from M to a single ACK and pushing energy-delay-throughput to the ideal.

5. EVALUATION RESULTS

For all our evaluations, we perform full-system simulations using Wind River Simics [2]. We model FANOUT and FANIN in detail in the Garnet [5] on-chip network simulator within the GEMS [30] infrastructure. This simulation framework provides a cycle-accurate timing model. We use Orion 2.0 [21] for estimating the power consumed by the components of the network.

5.1 Target System and Configuration

We model a 64-core tiled CMP with the parameters shown in Table 2. Our network and technology parameters are shown in Table 3 and Table 4. The number of buffers/VCs in all configurations is set by the buffer turnaround time within the request/response message classes in the baseline. For optimizing just FANOUT+FANIN relative to the ideal, we achieve similar results with 2/3rd the buffer/VC space.

We evaluate the parallel sections of the SPLASH-2 [3] and PARSEC [8] benchmarks for all configurations. Each run consists of 64 threads of the application running on our CMP. We run multiple times with small random perturbations to capture variability in parallel workloads [7], and average the results.

5.1.1 Coherence Protocols

We run two broadcast-based coherence protocols, one derived from HyperTransport (HT) [11], and one from Token Coherence (TC) [29]. In HT, all cores send requests as unicasts to a stateless directory home node (ordering point), which forwards it to all other cores via a broadcast. The requester collects all acknowledgements, and then unblocks the home node via a unicast. We enhance the protocol with an optimization that merges multiple read requests to the same cache line at the home node when those requests are competing for the same unblock message. This optimization reduces the additive queueing delay incurred by these waiting requests, and also avoids broadcasting each of them, lowering the application runtime of the baseline network by 39.6% on average. In TC, all cores broadcast their requests, and only cores with *tokens* respond. Ordering races are resolved by broadcasting *persistent* requests. Because not all nodes send back ACKs in TC, master ACKs in FANIN do not wait at routers, and instead perform opportunistic aggregation of the tokens. The proportion of 1-to-M and M-to-1 flows in both these protocols, shown earlier in Fig. 1(a), is important to keep in mind when understanding the results.

5.1.2 Baseline Network

We model a baseline network with hardware multicast support, similar to VCTM [15], bLBDR [34], MRR [16], and RPM [37], with routers forking flits (BASE_fork@rtr) as they move towards their destination. The baseline broad-

[17]GETS requests in Token Coherence [29] are also assigned an *m_id* of -1 since MSHR entries can become free before all tokens are received in some cases.

Table 2: CPU and Memory

Processors	64 in-order SPARC
L1 Caches	Private 32 kB I&D
L2 Caches	Private 1MB per core
Coherence Protocol	(1) HyperTransport [11] (2) Token Coherence [29]
DRAM Latency	70ns

Table 3: On-chip Network

Topology	8x8 mesh
Router Ports	5
Ctrl VCs/port	12, 1-flit deep
Data VCs/port	3, 3-flit deep
Flit size	128 bits
Link length	1mm

Table 4: Process

Technology	45 nm
V_{dd}	1.0 V
Frequency	2.0 GHz

cast follows an XY-tree routing algorithm, as explained in Section 3.1. The pass-gate matrix-crossbar from Fig. 4(b) is used in the baseline for its low power and area.

5.2 Evaluation Results

Runtime with FANOUT and FANIN. We start by comparing the performance benefits of FANOUT and FANIN, with all their routing, flow-control, and microarchitecture optimizations included, against the IDEAL, which was shown in Fig. 1(b). Fig. 8(a) shows the normalized full-system application runtime with FANOUT and FANIN for both protocols. With FANOUT, HT shows 10% improvement, while TC shows 9.7% improvement, on average. With the addition of FANIN, we see 18.4% runtime reduction on average for HT, and 11.4% for TC. This can be understood by the message count breakdowns in Fig. 1(a). HT has 14.1% M-to-1 injections, each of which translates to 63 flits, leading to bursty congestion. FANIN thus has a higher impact on performance. In contrast, TC has less than 2% M-to-1 traffic, which is why FANIN adds only 1-5% speed-up relative to FANOUT across the benchmarks.

These results show that the progress of applications is limited more by ACKs in HT, and by broadcasts in TC. FANIN and FANOUT respectively provide network solutions to remove these bottlenecks and allow the application to approach its runtime on the ideal networks. Compared to IDEAL, FANOUT+FANIN is off by less than 1% for both HT and TC.

Network latency with FANOUT and FANIN. The full-system runtime behavior seen earlier can be understood by looking at the network latency impact in Fig. 8(b). The FANOUT design reduces average network latency by about 14% for HT and 50% for TC. This is again due to 45-55% broadcasts in TC, as opposed to 11-17% broadcasts in HT. The average latency of broadcast packets was observed to go down by 40% on average in both HT and TC with FANOUT. However, the dominance of bursty ACKs in HT increases its average latency across all packets. The addition of FANIN enables HT to observe a 50% reduction in network latency. FANOUT+FANIN reduce the average network latency to about 20 cycles for both HT and TC, which roughly translates to 2.5 cycles per hop[18]. FANOUT+FANIN thus enables each network packet to incur just 0.5-cycle more latency per hop on average than the minimum 2-cycle per hop datapath.

Network energy with FANOUT and FANIN. Fig. 9 shows the breakdown of network energy with FANOUT and FANIN for HT and TC. The energy consumed by the AA logic (comparators and adders) in FANIN is also accounted for. For HT, the overall network energy actually goes up by 4.3% with FANOUT. The reason for this increase can be seen from the relative breakdown of buffer and crossbar energy. Buffer reads do go down with FANOUT (due to the single-cycle pipeline enabled by the mXbar), but only by 3.9% because the buffer accesses are dominated by the ACKs in HT. The crossbar energy, meanwhile, goes up by 10.1% because the mXbar consumes more energy than the baseline crossbar for unicasts, as explained earlier in Table 1.

When FANIN is added, however, the network energy goes down by 60.2% on average. The advantage of FANIN is not only the reduction of traffic, leading to fewer buffer, crossbar, and link traversals, but also the reduction of contention, thereby complementing FANOUT's optimizations.

For TC, the story is different. The dominance of broadcasts means that FANOUT gives a 19.7% reduction in energy, primarily due to a reduction in buffer reads (due to the mXbar) and buffer writes (due to multicast bypassing). FANIN lowers the energy further by 2%.

Compared to the IDEAL in energy[19], FANOUT+FANIN is off by 9.6% on average for both HT and TC, as opposed to 32.1% in the baseline. If we reduce the number of VCs by 2/3rd, to satisfy turnaround in FANOUT+FANIN rather than baseline, FANOUT+FANIN is 7.9% off IDEAL energy, while still less than 1% off IDEAL runtime.

Impact of components of FANOUT. In Fig. 10(a), we show the effect on full-system runtime of stand-alone components of FANOUT (*Whirl*, mXbar, bypass). *Whirl* shows 4% runtime savings for *nlu* and *fluidanimate* in HT and 6% for *lu* in TC. These are applications that have 1.3-2X higher injection rates than the others, and thus benefit from a load-balanced network. For the other applications, the XY-tree of the baseline does as well as *Whirl*. mXbar shows about 5-10% runtime reduction in TC for most benchmarks. In HT, *nlu, canneal* and *swaptions* benefit from mXbar but other benchmarks do not because the performance starts getting limited by the ACKs. Buffer bypassing by itself shows 3-4% runtime improvement in average in both HT and TC. Without the mXbar, flits need to get buffered to fork out of the router, and buffers are bypassed only at those routers where no forks need to occur.

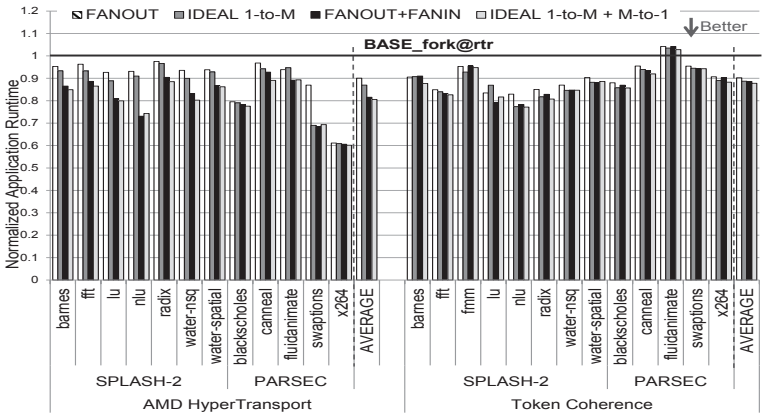
When all three components of FANOUT come together, flits follow load-balanced paths, fork through the mXbar, and avoid getting buffered, resulting in a 10% improvement in runtime on average, which is higher than what any one technique provides.

Impact of components of FANIN. We discuss the impact of FANIN on HT in Fig. 10(b). We first evaluate the performance of FANIN with opportunistic aggregation (i.e., with no waiting). This results in a 6.9% runtime improvement. The ratio of received-to-injected ACKs goes down to 0.31 on average, which means one-third of the ACKs for an M:1 flow get aggregated.

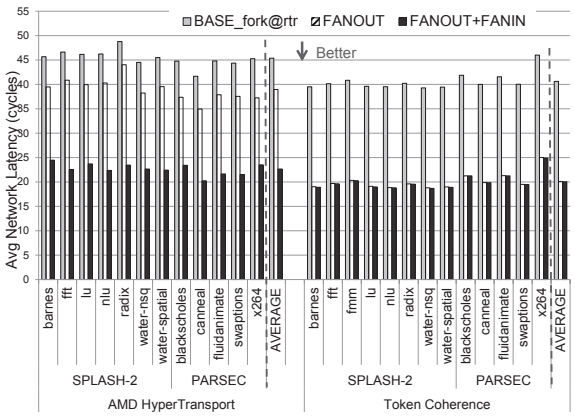
Next, we add our heuristic of waiting for $furthest\_hops \times 4$ cycles. The ideal ratio of received-to injected ACKs should be 1/63 (all cores except the requester send an ACK) =

[18]Assuming an average hop count of eight in an 8x8 mesh.

[19]The energy consumed in the ideal networks is just wire/datapath energy, i.e. ST and LT.

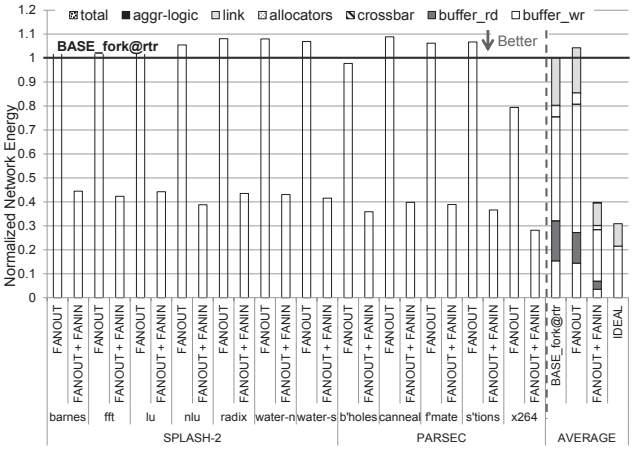


(a) Full-system application runtime (normalized against BASE_fork@rtr)

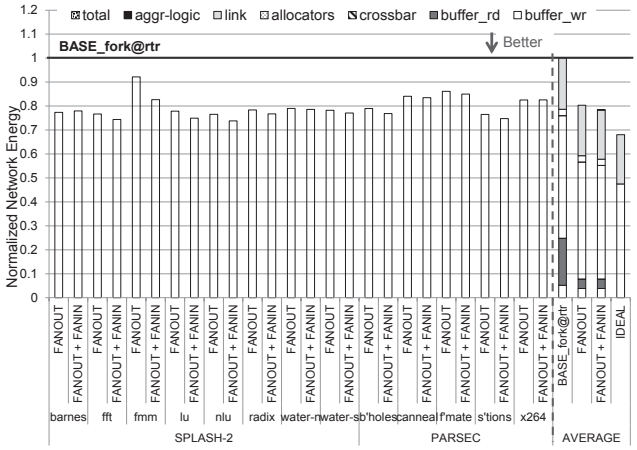


(b) Average network latency

Figure 8: Impact of FANOUT and FANIN on performance.

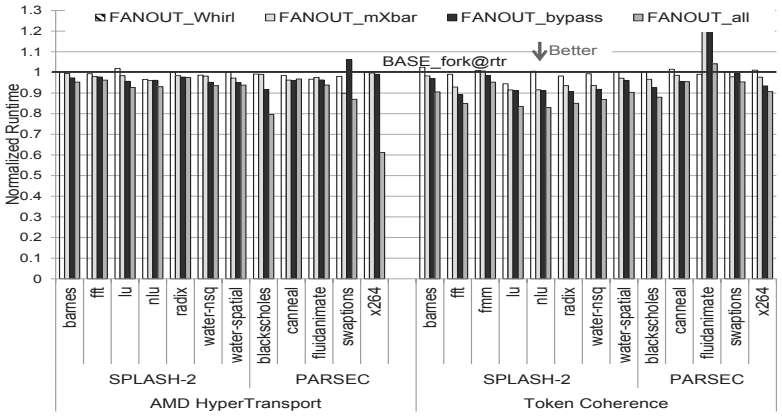


(a) Network energy for HyperTransport (normalized against BASE_fork@rtr)

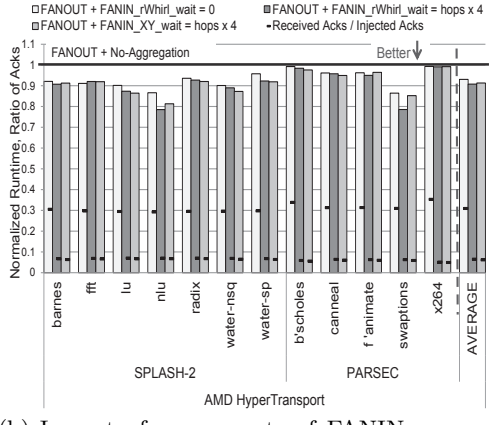


(b) Network energy for Token Coherence (normalized against BASE_fork@rtr)

Figure 9: Impact of FANOUT and FANIN on network energy.



(a) Impact of components of FANOUT on application runtime (normalized against BASE_fork@rtr)



(b) Impact of components of FANIN on runtime and received ACKs (normalized against FANOUT)

Figure 10: Impact of routing and flow-control components of FANOUT and FANIN.

0.015. However, because we do not perform any waiting at the destination router, the destination NIC should receive four ACKs, instead of 63, making the best achievable ratio = 4/64 = 0.0625. We observe that the received-to-injected ACKs goes down from 0.31 to 0.065 with our heuristic, which is only 4% higher than the best FANIN can achieve. The

runtime goes down by another 3%. While the waiting heuristic is needed to approach the ideal network's runtime, higher wait times degraded runtime because waiting ACKs reduce available router buffers, and inefficient waiting could delay request completion.

We also study the impact of *rWhirl* versus XY routing. Interestingly, XY performs comparably to, and sometimes slightly better than, *rWhirl* for most benchmarks, except *nlu* and *swaptions*. The reason is that XY forces all ACKs to travel X first, which results in all routers in the North/South of the destination receiving ACKs from three directions (while routers in East/West of the destination receive ACKs from only one direction). If the wait-time is perfect, this should not matter. However, because it is only a heuristic, the waiting master ACK at the North/South routers in XY ends up performing 3% more aggregations than in the *rWhirl* case. This result highlights that, for ACKs, the waiting and aggregation ratio has a higher impact on runtime than the load-balancing across network paths.

6. CONCLUSIONS

We present FANOUT and FANIN which approach ideal energy-delay-throughput for 1-to-M and M-to-1 on-chip traffic, respectively. FANOUT is an in-network forking methodology for 1-to-M flows, comprising of a load-balanced routing algorithm for multicasts, a crossbar circuit that forks flits at the similar delay/energy as unicasts, and a single-cycle router. FANIN is an in-network aggregation methodology for M-to-1 flows, comprising of a synchronized routing algorithm for ACKs, an intelligent waiting heuristic for efficiency, and a single-cycle router. We demonstrate that FANOUT and FANIN can provide network scalablility to shared memory coherence protocols that frequently use broadcasts and wide multicasts. The optimizations we propose, however, are much more general, and can also be ported to enhance user-level messaging systems that use 1-to-M and M-to-1 communication.

7. REFERENCES

[1] Intel Nehalem. `http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719`.
[2] Simics Full-system Simulator. `http://www.windriver.com/products/simics`.
[3] SPLASH-2. `http://www-flash.stanford.edu/apps/SPLASH/`.
[4] J. L. Abellán *et al.* Efficient and scalable barrier synchronization for many-core CMPs. In *Proc. 7th ACM International Conference on Computing Frontiers*, 2010.
[5] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *ISPASS*, Apr. 2009.
[6] N. Agarwal, L.-S. Peh, and N. K. Jha. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. In *HPCA*, Feb. 2009.
[7] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.
[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, Oct. 2008.
[9] E. E. Bilir *et al.* Multicast snooping: A new coherence method using a multicast address network. In *ISCA*, 1999.
[10] J. G. Castanos *et al.* Evaluation of a multithreaded architecture for cellular computing. In *ISCA*, 2002.
[11] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27:10–21, Mar. 2007.
[12] P. Conway *et al.* Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30:16–29, 2010.
[13] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Pub., 2003.
[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, Dec. 2008.
[15] N. Enright Jerger, L.-S. Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *ISCA*, Jun. 2008.
[16] P. A. Fidalgo, V. Puente, and J.-Á. Gregorio. MRR: Enabling fully adaptive multicast routing for CMP interconnection networks. In *HPCA*, 2009.
[17] M. Galles. Scalable pipelined interconnect for distributed endpoint routing: The SGI SPIDER chip. In *Hot Interconnects 4*, Aug. 1996.
[18] A. Gara *et al.* Overview of the Blue Gene/L system architecture. *IBM J. Res. Dev.*, 49:195–212, Mar. 2005.
[19] A. Gottlieb *et al.* The NYU Ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Trans. on Computers*, 32:175–189, 1983.
[20] Y. Hoskote *et al.* A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, Sept. 2007.
[21] A. B. Kahng *et al.* ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. *DATE*, Feb. 2009.
[22] V. Krishnan and J. Torrellas. The need for fast communication in hardware-based speculative chip multiprocessors. *Int. J. Parallel Program.*, 29:3–33, Feb. 2001.
[23] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha. A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS. In *ICCD*, Oct. 2007.
[24] A. Kumar, L.-S. Peh, and N. K. Jha. Token flow control. In *MICRO*, Nov. 2008.
[25] A. Kumar *et al.* Express virtual channels: Towards the ideal interconnection fabric. In *ISCA*, Jun. 2007.
[26] G. Kurian *et al.* ATAC: a 1000-core cache-coherent processor with on-chip optical network. In *PACT*, 2010.
[27] J. Laudon and D. Lenoski. The SGI origin: a ccNUMA highly scalable server. In *ISCA*, Jun. 1997.
[28] D. Lenoski *et al.* The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA*, Jun. 1990.
[29] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *ISCA*, Jun. 2003.
[30] M. M. K. Martin *et al.* Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *CAN*, Sep. 2005.
[31] H. Matsutani *et al.* Prediction router: Yet another low latency on-chip router architecture. In *MICRO*, Feb. 2009.
[32] G. F. Pfister *et al.* The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. In *ICPP*, pages 764–771, 1985.
[33] A. Raghavan *et al.* Token tenure: PATCHing token counting using directory-based cache coherence. In *MICRO*, Nov. 2008.
[34] S. Rodrigo *et al.* Efficient unicast and multicast support for CMPs. In *MICRO*, Sep. 2008.
[35] A. F. Samman *et al.* Multicast parallel pipeline router architecture for network-on-chip. In *DATE*, 2008.
[36] K. Strauss *et al.* Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *MICRO*, 2007.
[37] L. Wang, Y. Jin, H. Kim, and E. J. Kim. Recursive partitioning multicast: A bandwidth-efficient routing for networks-on-chip. In *NOCS*, 2009.
[38] H.-S. Wang *et al.* Power-driven design of router microarchitectures in on-chip networks. In *MICRO*, 2003.
[39] M. A. Watkins *et al.* ReMAP: A reconfigurable heterogeneous multicore architecture. In *MICRO*, 2010.
[40] D. Wentzlaff *et al.* On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, Sept. 2007.