Single-Cycle Collective Communication Over A Shared Network Fabric

Tushar Krishna
Intel Corporation, VSSAD
Hudson, MA 01749. USA
tushar.krishna@intel.com

Li-Shiuan Peh
Department of EECS. MIT
Cambridge, MA 02139, USA
peh@csail.mit.edu

***Abstract*—In the multicore era, on-chip network latency and throughput have a direct impact on system performance. A highly important class of communication flows traversing the network is collective, i.e., one-to-many and many-to-one. Scalable coherence protocols often leverage imprecise tracking to lower the overhead of directory storage, in turn leading to more collective communications on-chip. Routers with support for message forking/aggregation have been previously demonstrated, supporting such protocols. However, even with the fastest possible designs today (1-cycle routers), collective flows on a $k{\times}k$ mesh still incur delays proportional to k since all communication is across the entire chip. As k increases across technology generations, the latency of these flows will also go up.**

However, the pure wire delay to cross the chip is just 1-2 cycles today, and is expected to remain roughly invariant. The dependence of message delays on k arises due to the requirement to latch messages at every router. In this work, we remove this requirement. We design a network fabric that enables messages to (1) dynamically create virtual 1-to-Many (multicast) and Many-to-1 (reduction) tree routes over a physical mesh, (2) get forked/aggregated at nodes on the tree, and (3) traverse the tree - all within a single-cycle across each dimension. For synthetic 1-to-Many/Many-to-1 flows, we demonstrate 76/82% reduction in latency, and 1.6/2X improvement in throughput over a state-of-the-art NoC with 1-cycle routers and support for collective communication. Across a suite of SPLASH-2 and PARSEC benchmarks, full-system runtime and energy is reduced by 14% and 50% for a limited-directory protocol.

I. Introduction

In a multicore system, there occur scenarios when all cores need to participate to either service a request or signal the end of a transaction. This *collective communication* can be further classified into one-to-many (multicast) and many-to-one (reduction) flows. In the message passing domain, examples include routines like MPI_Bcast and $MPI\_Reduce/MPI\_Barrier$ respectively. In the shared memory domain, these occur in cache coherence protocols that use imprecise sharer information [1], [2] at the directory, for area/power/scalability, and instead resort to broadcasting requests and collecting ACKs to maintain coherence.

Without any network support to handle collective communication, a root sending/receiving broadcast/ACKs to/from M cores sends/receives M separate messages, which adds serialization delay and throttles throughput, by at least M, leading to system slowdown. This observation has spawned a lot of recent research in NoCs that support message forking [3], [4], [5], [6], [7], [8], [9] and aggregation [8], [9] at routers. We call this body of work *Baseline+Collective* in this paper. There has also been work in dedicated reduction networks for barriers [10], [11], [12]. While message forking and aggregation within the NoC lowers the bandwidth demands of these flows, latency is still a concern, since by design the root almost *always* has to communicate with the furthest cores on-chip[1]. This means that even if we can design routers with forking/aggregation support at *only* 1-cycle delay at every hop [8], the network delay will grow proportional to k in a k-node ring or a $k{\times}k$ mesh. As core count scaling increases k, this starts becoming a concern, potentially requiring expensive on-chip directory structures to minimize long distance communication.

Global wire delay is in fact not the problem when it comes to on-chip latencies. Repeated wires have been shown to transmit up to 13-16mm within 1 ns (i.e., $\sim$62 ps/mm)[2] [13], [14], [15], [16]. Given maximum chip sizes of $\sim$20mm$\times$20mm today, repeated wires can thus enable cross-chip communication within 1-2 cycles at 1 GHz. Absolute wire delay is not going down with technology scaling [13], [17]. But the trend of fairly constant clock frequencies (due to the power wall) and chip dimensions (due to yield) means that the delay in *cycles* to get from one end of the chip to the other is expected to remain 1-2. Creating a fully-connected topology is however not a feasible solution beyond a few cores, and we need routers to multiplex flows on a shared set of links. These routers lead to the dependence of latency on the number of hops traversed.

A highly promising approach to remove this dependence has been to create *virtual*, reconfigurable single-cycle multi-hop paths over a regular mesh [15], [16]. The idea is to replace the clocked drivers at every router by clockless repeaters, and drive signals across multiple hops within a cycle before they get latched at the destination router. The maximum number of hops (tile to tile distance) that can be traversed in a cycle, or HPC_{max}, depends on the underlying technology. At 45 nm, HPC_{max} is 16 for a pure repeated wire with 1mm tiles at 1GHz, and drops to 11 for a full data-path with a repeater and crossbar (mux) at every hop. Dynamically creating single-cycle multi-hop paths on demand has been demonstrated in SMART [15], [16] and found to be a better solution than both high-radix topology solutions with dedicated wires between a subset of nodes, and meshes with 1-cycle routers.

We compare the performance of Baseline+Collective - with 1-cycle-routers, and SMART, against a Baseline mesh (with 1-cycle routers and no collective communication support) and a Fully-connected NoC with 1-cycle dedicated links between all NICs (impractical due to area/power reasons). In Figure 1, we plot the average runtime of a 64-core system running a

The authors acknowledge the support of DARPA UHPC, SMART-LEES and MARCO C-FAR. This work was performed while Tushar Krishna was a PhD student at MIT.

[1]This is mitigated if the furthest cores are not in the destination set.
[2]Experimental Parameters: repeater spacing = 1mm, wire spacing = 3X the minimum to nullify the coupling capacitance.

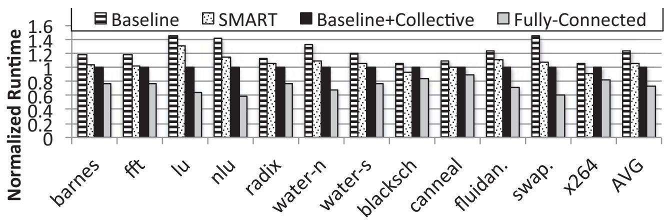


Fig. 1: **Impact of NoC designs on runtime for HT [1].**

limited-state directory protocol, modeled similar to AMD's HyperTransport [1] protocol[3] across a suite of SPLASH-2 and PARSEC benchmarks. Baseline+Collective and SMART give 18.7% and 14.0% performance improvement, respectively, over the Baseline. However, both these designs leave 26-30% performance on the table, compared to a Fully-Connected topology. The goal of this work is to bridge this gap, allowing collective communication flows to achieve the performance of a fully-connected topology on a shared mesh. The key challenge is that all the Baseline+Collective designs require flits to *stop* at every router, to get forked/aggregated. The SMART approach, on the other hand, is to allow flits to *bypass* routers completely and only get latched at the destination. These are conflicting targets since the former requires the use of intelligent logic within the router, while the latter tries to bypass all logic within the router. This work presents a technique to simultaneously achieve both goals.

We present two techniques, SMART-FanOut (SFO) and SMART-FanIn (SFI) that overlay broadcast and reduction trees, with dynamically changing roots (i.e., the source/destination of the broadcast/reduction) over a shared network fabric, such as a mesh, and traverse these trees within a cycle leveraging the single-cycle multi-hop capability of repeated wires. Across synthetic benchmarks, SFO/SFI demonstrate 76/82% reduction in latency, and 1.6/2X improvement in throughput over Baseline+Collective. Full-system simulations over a limited-directory protocol show a runtime reduction of 14% with SFO+SFI, which is only 12% away from the runtime of the Fully-connected topology.

The paper is organized as follows. Section II describes relevant background on SMART. Section III and IV present SFO and SFI. Section V presents the evaluations. Section VI contrasts against prior art and Section VII concludes.

II. Background: SMART NoC

Single-cycle Multi-hop Asynchronous Repeated Traversal (SMART) [15] enables traversals across multiple-hops within one cycle. The key idea is to replace clocked drivers within every router's crossbar by clock-less/asynchronous repeaters, thus dynamically creating a multi-hop repeated link which can drive signals across 13-16 mm within a GHz [13], [15], [16]. These repeaters are simultaneously setup a cycle in advance through a flow control mechanism to allow multiple flows to create *virtual* single-cycle multi-hop paths cycle-by-cycle.

In conventional NoCs, flits at every router arbitrate among themselves to gain access to the output ports during Switch Allocation *Local* (SA-L). The winner of SA-L traverses the crossbar and output link to the next router, stops, arbitrates for the next link, and so on. In a SMART NoC, flits arbitrate for *multiple* links and the buffer at the end point, all within the same cycle. Each output port winner from SA-L first broadcasts a SMART-hop setup request (SSR) up to HPC_{max}-hops from that output port. These SSRs - dedicated repeated wires that connect every router to a neighborhood of up to the HPC_{max} - help preset the intermediate routers for a multi-hop bypass path. SSRs are $log_2(1 + HPC_{max})$ bits wide, and carry the number of hops the flit wishes to traverse. Following the initial SSR broadcast, every router performs a second round of arbitration - Switch Allocation *Global* (SA-G) - to arbitrate among the SSRs they have received from the routers in their HPC_{max} neighborhood and setup three controls signals: BW_{ena}, BM_{sel} and XB_{sel}. BW_{ena} decides whether to latch the incoming flit or not; BM_{sel} at the input of the crossbar switch chooses between the incoming (*bypass*) flit on the link and a buffered (*local*) flit, and XB_{sel} connects an input port to an output port. In the next cycle, the flit performs a multi-hop switch and link traversal till it is stopped at a router with BW_{ena} = 1. Figure 2 provides an illustration. Router R0 creates a 3-hop path that can be traversed within a cycle, and the control signals at each router are shown.

Competing SSRs. The SA-G arbiters guarantee that only one flit will be allowed access to any particular input/output port of a router crossbar; any conflicting flits will be stopped by pulling BW_{ena} high. To decide which flit gets to go and which has to stop, every router prioritizes SSR requests according to a fixed priority based on flit distance. For e.g., the *Prio=Local* scheme gives highest priority to the local flit, followed by the flit from the neighboring router, followed by the flit from the router two hops away, and so on; If a router receives an SSR requesting a bypass, but also has its own flit to send out, it prioritizes the latter by raising BW_{ena} to 1, and setting BM_{sel} to $local$. Single-cycle multi-hop paths are thus opportunistic, not guaranteed. An alternate priority, *Prio=Bypass*, prioritizes flits from the furthest router over the flits from the nearer ones.

All routers need to collectively agree, in a distributed manner and during the same cycle, on which flit is performing a particular multi-hop traversal. This is required to make sure that the flit is latched at its correct destination, and not misrouted beyond the allowed HPC_{max} hops. This is guaranteed by enforcing the *same* relative priority between SSRs at each router - i.e., all routers need to enforce Prio=Local or Prio=Bypass. For bypassing routers at turns, a second-level priority based on direction is also required. Other details about the design (VC allocation, ordering etc) are not relevant to understand the rest of this work. In this work, we use the SMART_1D design, where flits can create single-cycle multi-hop paths along one dimension at a time, stopping at the turning routers.

III. SMART-FanOut: Single-cycle Broadcast

The goal of SMART-FanOut is to accomplish a broadcast within a single-cycle per-dimension. For e.g., in a 8×8 mesh,

[3]The distributed directory serves as an ordering point, but does not have any state. Instead it broadcasts all requests, and collects all ACKs.

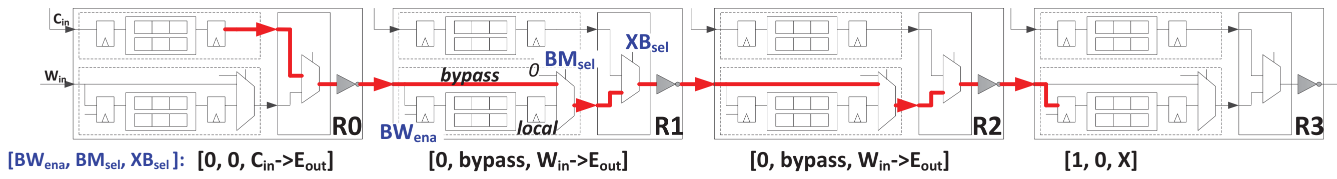


Fig. 2: **Example of a Single-cycle Multi-hop Path from R0 to R3 using SMART.** *The control signals shown are setup one cycle before the actual traversal using separate control wires called SSRs. If R2 has to send a flit out in the same cycle, it can set its BW_{ena} to 1 and BM_{sel} to local, prematurely stopping the flit from R0*. **[15]**

it should take 2 cycles to reach all of the 64 routers.

A. *Routing over Virtual Broadcast Trees*

Broadcasts are routed by creating virtual trees over the physical mesh, such as a XY-tree as shown in Figure 3(a). We call this a **Shared Virtual Tree (SVT)**. Each physical link of the mesh still needs to be arbitrated for, and the forks within the route (i.e., sending copies out of multiple ports) are performed by the router, within a cycle [6], [7], [8]. Broadcasts from multiple sources leads to the overlay of multiple SVTs arbitrating for the same set of links, resulting in either shorter bypass paths in SMART due to premature stopping (if Prio=Local is used) or longer wait times to get the longer bypass path (if Prio=Bypass is used). We present a novel broadcast tree called **Private Virtual Tree (PVT)** that eliminates contention between broadcasts completely, enhancing the chances of successfully arbitrating for all links along the dimension. PVTs originate from the four corner routers (CR), and use entirely different links to broadcast. All nodes that wish to broadcast first send the flit as a unicast to the closest CR, dynamically arbitrating for SMART paths like other unicasts. Since SMART is the underlying fabric, this redirection is relatively inexpensive[4]. Figure 3(b) shows an example. Suppose Router 22 wants to send a broadcast. In Step 0, it sends a unicast to CR 40. In Step 1, CR 40 broadcasts the message out of its North output port. We call this the *straight dimension*. In Step 2, all routers along the right edge (including Router 40) broadcast the message along the West direction. We call this the *turn dimension*. A copy of the message is retained at all routers as it proceeds along the multi-hop path. In Step 3, the broadcast flits are delivered to the NIC. Steps 0 and 3 are unicasts (to CR and NIC respectively) and use regular SMART arbitration to resolve contention. Steps 1 and 2 are broadcasts, and PVT ensures no contention for links among the broadcasts flits, as shown in Figures 3(c) and (d). CR 00 uses E-N links, 40 uses N-W links, 44 uses W-S links, and 04 uses S-E links. But there can be contention for links between a broadcast and unicasts, for which we explore two arbitration strategies, Complete and Greedy.

B. *SMART-FanOut Complete (SFO_Complete)*

In a *complete* fanout, a multicast flit at a router proceeds only if it has access to *all* links along the straight/turn dimension during the same cycle. This is ensured by presetting all routers along the straight and turn dimensions into a full bypass path (i.e., BM_{sel}=bypass in Figure 2), one cycle after the other. These are called Broadcast Slot Straight (BSS) and Broadcast Slot Turn (BST). BW_{ena} is also set to 1 during both the broadcast slots to retain a local copy of the bypassing flit, to send it to the NIC. The time interval between two BSSs is called Broadcast Interval (BI), and is a microarchitectural parameter. For instance, if BI equals 4, then cycle 0, 4, 8,... correspond to BSS, and cycles 1, 5, 9,... correspond to BST[5]. During BSS, W→E, S→N, E→W and N→S bypass paths are preset in routers along the bottom, right, top and left edges respectively, as shown in Figure 3(c). During BST, W→E, S→N, E→W and N→S bypass paths are preset at all routers, as shown in Figure 3(d). Steps 1 and 2 occur during BSS and BST respectively, guaranteeing a 2-cycle chip-wide broadcast. No unicast flits are allowed to use the reserved paths in these cycles. This is achieved by blocking SA-G for the appropriate ports one-cycle before BSS and BST. A subtle point to note is that arbitration for the NIC (i.e, Step 3) by buffered flits within the routers (unicast or broadcast) also needs to be blocked for the broadcast slots. This is because the input bandwidth of all crossbars at the participating routers is allocated to the bypassing flits on single-cycle multi-hop paths, so locally buffered flits at the same input ports cannot use this bandwidth during this cycle. If the mesh were to be designed exclusively for broadcasts, the theoretically minimum value of BI is 3: two cycles for BSS and BST, and one for Step 3 and/or 0.

1) *Deadlocks.*

Figure 3(b) shows that PVT allows X to Y, Y to X, and u-turns (e.g., the N port at CR 40). To avoid deadlocks, we divide the VCs in the broadcast virtual network into 3 classes: VC_{to_CR}, $\mathrm{VC}_{before\_turn}$, VC_{after_turn}. The first is used by "broadcast" flits to reach the CR via unicast. The flits then switch to $\mathrm{VC}_{before\_turn}$ and traverse the first dimension (X or Y depending on the particular CR). At the turn, they switch to VC_{after_turn}. There is no cyclic dependency.

2) *Buffer Management.*

The VC_{to_CR} is used by unicasts and sends a on-off signal only to its nearest neighbors [15][6]. The $\mathrm{VC}_{before\_turn}$ and VC_{after_turn} classes send one-bit on-off signals each up to HPC_{max} hops via repeated wires (which are inherently multi-drop). Step 1 from the CR occurs *only* if *all* routers along that dimension have a free buffer to hold the broadcast flit in the $\mathrm{VC}_{before\_turn}$ class. If not, it waits till its next time slot to

[4]Moreover, in many limited directory protocols, broadcasts only emanate from the corner routers, since these house memory controllers connected to the directory/LLC, removing the need for this redirection.

[5]If HPC_{max} is less than the number of nodes in a dimension, the leaves that lie within HPC_{max} to $2{\times}HPC_{max}$, receive the broadcast one cycle later by statically shifting BSS and BST across the routers, and so on.

[6]A flit is conservatively pre-emptively stopped at a router if its neighbor does not have free VCs, since bypass at the neighbor is not guaranteed.

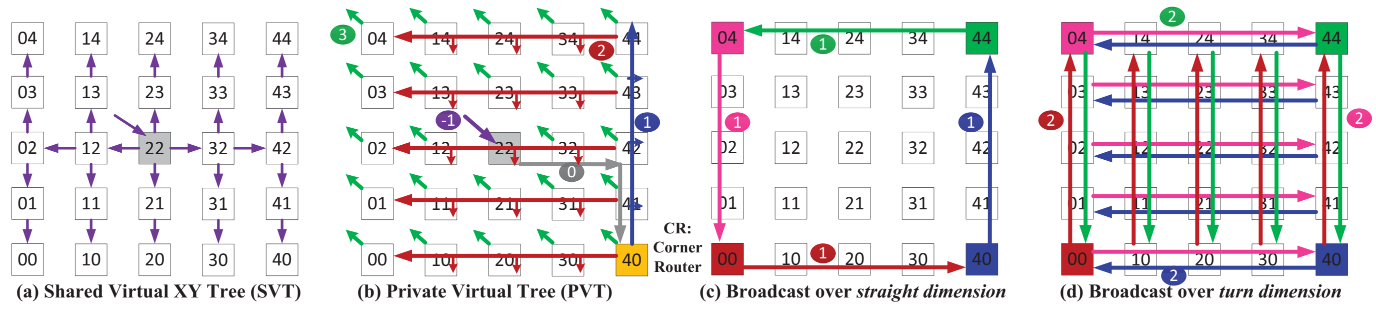

(a) Shared Virtual XY Tree (SVT)
(b) Private Virtual Tree (PVT)
CR: Corner Router
(c) Broadcast over straight dimension
(d) Broadcast over turn dimension

Fig. 3: **SMART-FanOut (SFO).**

send. Similarly for Step 2. SFO_Complete does not send flits to only a subset of nodes along the dimension.

C. SMART-FanOut Greedy (SFO_Greedy)

In a *greedy* fanout, the broadcast flit traverses its route (PVT or SVT) opportunistically, trying to reach as many of its leaf routers as possible within a cycle. While it has a higher chance of reserving all links along the dimension in a PVT than in a SVT, competition with unicast flits, and/or insufficient buffers can lead to shorter express paths, and re-arbitration during subsequent cycles for the rest of the path.

Broadcast flits arbitrate for multiple output ports (depending on their route). The winner at each output port sends out an SSR along that direction, requesting a bypass path till the end of the dimension, with an additional broadcast bit. During SA-G, in addition to the appropriate setup of the BM_{sel} and XB_{sel} signals, BW_{ena} is also made high to retain a copy of the incoming flit at this router. In the next cycle, the flit performs a multi-hop switch and link traversal along the dimension. The retained copies arbitrate for the ejection port, and the turn dimension.

A broadcast's bypass could get terminated (BM_{sel} = 0 or local) at a router R before traversing the entire dimension for the following reasons: (a) it loses SA-G to some locally buffered flit at R if Prio=Local were used, (b) the neighbor of R does not have free buffers[7].

D. Multicasts

Multicasts are delivered exactly like broadcasts in both schemes, with copies being retained at every router. This copy is dropped if the current NIC is not part of the destination set[8].

IV. SMART-FanIn: Single-cycle Reduction

We target the scenario where M nodes send an "ACK" each to the same destination, where it is ultimately combined via some reduction operator (ADD, OR, MIN, MAX etc). Examples include acknowledgments in coherence protocols, barrier synchronization, MPI_reduce routine in MPI, and so on. M separate ACKs cause latency, throughput and energy overheads. The goal of SMART-FanIn is to perform a distributed reduction within the network on a single-cycle (per-dimension) multi-hop path, so that the destination receives only *one* ACK with the result of the reduction - a count of M if the operator is ADD. We assume an ADD operator, though any associative and commutative operator works as well.

[7]We do not allow a non-continuous subset of routers to receive the broadcast to avoid tracking which routers the broadcast was not delivered to.

[8]Our design is optimized for broadcasts and dense multicasts. If the destination set is very sparse, it will be more efficient to send it as separate unicasts with SMART paths rather than create the broadcast tree.

A. SMART-FanIn Complete (SFI_Complete)

In a *complete* fanin, the destination NIC receives *exactly one* ACK representing an *aggregate* of all ACKs. The waiting cache/directory controller can proceed as soon as this ACK is received. Conceptually, all injected ACKs wait indefinitely at routers, before their ACK_count is added into the last ACK that is injected into the system. But we show an optimized implementation where most ACKs can be dropped immediately upon injection, and no explicit addition is required.

1) Microarchitecture.

We add 2 extra fields to the $SSRs$: 1-bit is_ACK, and k-bit ACK_id. ACK_id is used to identify ACKs from the same flow. We add a central table called *ACK Reduction Table (ART)* with 2^k entries, one for each ACK_id. Each entry is only 3-bits: 1-bit $reserved$, and 2-bit num_dir - to hold the count of the number of directions from where to aggregate ACKs. An ART entry is reserved by the preceding broadcast at its root/source, and used by the response ACKs. For each ACK_id, exactly num_dir separate ACKs enter each router. Till num_dir is greater than 1, incoming ACKs are stopped (BW_{ena} = 1) and *dropped*; num_dir is decremented by 1. If num_dir equals 1, the incoming ACK is the last ACK this router is waiting for, and is allowed to bypass or get buffered, decrementing num_dir to 0. The buffered ACK arbitrates for the switch to proceed further.

2) Walk-through Example.

We present a walk-through example of how SFI_Complete works, using Figure 4. The example is a simple one where there is only one M-to-1 (M=24) flow in the system, with all nodes sending ACKs to the root NIC at router 40.

Step #-1 (Broadcast Slot): Suppose the broadcast is sent out from Router 40 in a YX manner, as shown earlier in Figure 3(b). The broadcast sets up the control circuitry for the ACK. At the root CR 40, $ART[0]$ is reserved for the ACKs by setting $reserved$ to 1, and ACK_id 0 is embedded into the broadcast flit, and will be sent out with the ACKs. The num_dir count is appropriately set at all routers, for the XY routing by ACKs in this example. At Router 40, it is set to 2 to account for ACKs coming in from W and N[9]. At Router 00, it is set to 1 as an ACK from only the NIC will enter this router. At Router 41, it is set to 3 to account for ACKs from the W, N and NIC[10].

[9]There is no ACK from NIC 40 as it is the source of the broadcast.

[10]If a multicast, not a full broadcast, was sent out, the num_dir would not count ACKs from those NICs that are not in the destination set.

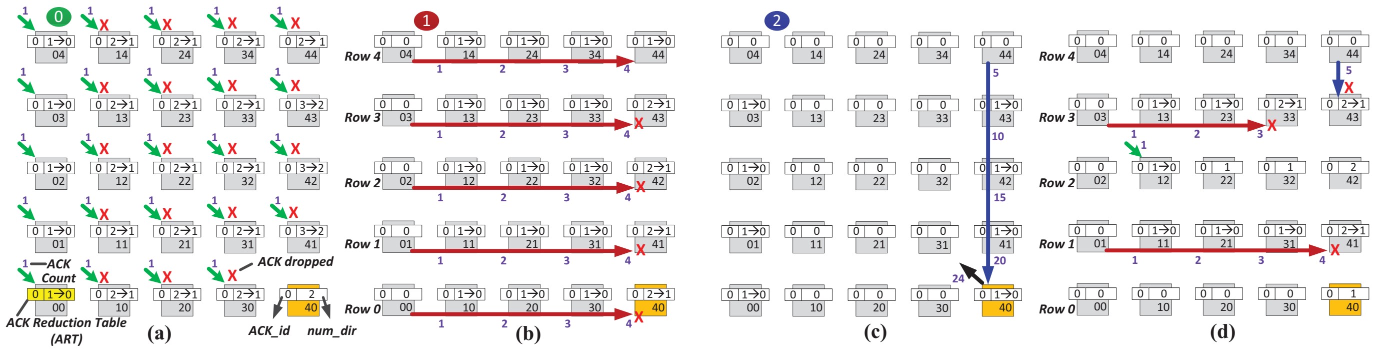


Fig. 4: **SMART-FanIn (SFI).**

Step #0 (ACK injection): All NICs inject ACKs into their routers. All NICs might not (and probably will not if there is contention at the NIC/cache) inject in the same cycle. We assume the same cycle just for the sake of this example. This is shown in Figure 4(a). The ACKs at Routers 00, 01, 02, 03 and 04 have $ART[0].num\_dir$ = 1, and get buffered to arbitrate for the output port (i.e., SA-L) competing with other buffered flits. All other ACKs have $num\_dir > 1$ and are *dropped*. $ART[0].num\_dir$ is decremented by one.

Step #1 (ACK X traversal): The ACKs that win SA-L (assumed to be all ready ACKs - 00, 01, 02, 03 and 04), send out SSRs along the X (East) direction, with is_ACK set to 1, and ACK_id set to 0. Let us look at the Row 0 in Figure 4(b). Routers 10, 20, and 30 have $ART[0].num\_dir$ as 1, and hence setup a single-cycle multi-hop bypass path for Router 00 till Router 40. In the next cycle, Router 00 sends the ACK in a cycle to 40, and frees $ART[0]$. At 10, 20, and 30, $ART[0].num\_dir$ is decremented to 0, and freed. The bypassing ACK has effectively *aggregated* the ACKs at the routers it bypassed. The same holds for other rows. At Routers 43, 42, 41, and 40, $ART[0].num\_dir$ is 2, and so these incoming ACKs from West are dropped, decrementing num_dir to 1. At Router 44, $ART[0].num\_dir$ is 1, and the incoming ACK from West is buffered, becoming the only active ACK for this M-to-1 flow.

Step #2 (ACK Y Traversal): This aggregating traversal is repeated along the Y direction. Figure 4(c) shows that Router 44 has $ART[0].num\_dir$ = 0, and tries to setup a single-cycle path to Router 40. Routers 43, 42, 41 and 40 setup a multi-hop bypass path since num_dir is 1. The ACK from Router 44 bypasses all these routers, leaving one ACK representing a count of 24 at Router 40. This ACK is sent up to the NIC. We have achieved perfect aggregation.

3) Handling arbitrary delays.

If num_dir is not 1, BW_{ena} is pulled high during SA-G, and the incoming ACK next cycle is latched. We allow only the final ACK to bypass through a router to aggregate all other ACKs. Figure 4(d) demonstrates a snapshot of the network at a certain cycle, demonstrating some scenarios that introduce arbitrary delays across the ACKs. **Row 0:** We can infer from num_dir=0 at 00, 10, 20 and 30 that their ACKs have already been delivered to 40 as an aggregated ACK, and then dropped since num_dir at 40 is 1. **Row 1:** The ACK from 01 is aggregating ACKs at 11, 21, 31 and 41. **Row 2:** Router 02 has num_dir equal to 0, so its ACK has been sent to 12 and dropped. The NIC at 12 is injecting an ACK this cycle that will decrement num_dir to zero and get buffered. It will then be able to perform a multi-hop aggregating traversal to Router 42. **Row 3:** The ACK from Router 03 aggregates ACKs at 13 and 23, but stops at 33, since num_dir was 2 as the ACK from the NIC has not been received yet. **Row 4:** The aggregated ACK from 44 tries to perform a multi-hop traversal to 40, but is dropped at 43, as num_dir is 2. Once the ACK from NIC 33 is injected, it can reach 43, making num_dir 0. This ACK would then be able to go to 40, bypassing (i.e., aggregating) 42, 41, and 40. We are able to achieve complete aggregation in this scenario as well, though not all ACKs were able to create long SMART paths, increasing the latency of the final received ACK.

4) Route.

The destination of the ACKs (the requester) could be different from the source of the broadcast (a CR - if PVT is used - or a memory controller). The $requester\_id$ field from the broadcast flit is used to compute num_dir for the ACKs for a pre-decided XY or YX routing.

5) ART entries.

All unique broadcast roots/sources need to choose different ART entries, thus making sure that ACKs from two separate active flows do not get assigned the same ACK_id. The minimum number of entries is 4 for PVT and N (number of cores) for SVT. We use a 64-entry ART (i.e., one per requesting core), derived empirically. Using an ART reduces ACK traffic, allowing us to remove a 128-bit buffer from the response VCs at each input port and use those to build the ART, thus adding no area or energy overhead [18]. A root is allowed to reassign an ART entry only after its ACKs have been received at the destination. If the ACK destination is the same as the root, this is easy to enforce. But if the ACK destination is different from the root, the root should not free the ART entry even after it reaches num_dir = 0 (the other routers should). The destination upon receiving the aggregated ACK sends a separate message to the root to free the entry. We piggy back this message on the regular coherence unblock messages to the directory at the CR, thus adding no overhead.

If there are no free ART entries for a broadcast to reserve, it marks its ACK_id as invalid. The corresponding ACKs do not try and aggregate, and are instead delivered as separate unicasts, which is functionally correct.

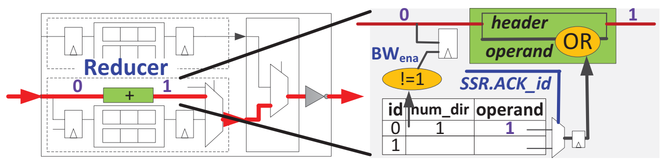


Fig. 5: **Reducer (OR) on datapath.**

6) Deadlocks.

ART entries conceptually represent indefinitely waiting ACKs. We avoid deadlocks since each ACK flow has a reserved ART entry at every router, and does not block ACKs from another flow.

7) Implementing other Reduce operations.

SFI can implement other Reduce operations such as OR, MAX, MIN, etc, by adding a *Reducer* block on the datapath, as shown in Figure 5. The protocol remains exactly the same, except that the ART has an extra field containing the operand value of the dropped ACK. The Reducer operates on the operands from the bypassing ACK and the one from the ART, and the result is sent out.

8) Timing.

On the control path, we added a check to set BW_{ena} to 1 if $ART[ACK\_id].num\_dir$ is not 1. This lookup did not affect the critical path (RTL implementation of SA-G) which was dominated by setting up the crossbar signals. On the datapath, if the *Reducer* is used, its gate delay could affect HPC_{max} which dropped from 11 to 9 for a 2-input OR (Spice simulations at 45nm).

B. SMART-FanIn Greedy (SFI_Greedy)

In a *greedy* fanin, ACKs are opportunistically aggregated with no explicit waiting. This means that the root could receive one or more partially aggregated ACKs. The ART is not used. The unique ACK_id of each flow is identified as ($mshr\_idx$, $dest\_id$) where $mshr\_idx$ is the index of the MSHR at the root node from where the broadcast was injected. Buffered ACKs poll incoming $SSRs$, while arbitrating for the switch, and aggregate any ACKs with the same ACK_id stopping at the router. For Prio=Bypass, we add an adder as the Reducer in Figure 5 to aggregate ACKs bypassing this router, if an ACK_id match was found during the SSR stage.

V. Evaluation

We implement SFO and SFI in the cycle-accurate NoC simulator Garnet [19], available within the GEMS [20] infrastructure.We model a 8×8 mesh for all our runs. Baseline+Collective, described in Section I, uses the SVT. For SFO and SFI, we assume HPC_{max}=8 [15].[11]

A. Synthetic 1-to-Many Traffic

We start by evaluating the flavors of SFO with synthetic broadcast traffic. The metric of performance is *1-to-M latency*, which we define as the time taken to deliver the multicast to *all* its destinations. We also plot the *Ideal* in each graph, assuming 1-cycle contention-less traversal, and theoretical throughput for that particular traffic over a mesh.

Broadcast from CRs. Figure 6a plots the 1-to-M latency as a function of injection rate, when only the 4 CRs inject. SFO lowers broadcast delivery time by 73-86% over the baseline. We first analyze SFO_Complete. In this design, the complete broadcast takes exactly two cycles. The best low-load latency is 3.8 cycles, offered by Broadcast Interval (BI) = 4. With BI=3, there is only one non-blocked time slot for the flits going up to the NIC (Step 3 in Figure 3(b)), adding penalties to flits that just missed the time slot, increasing low-load latency to 5.4 cycles. For BI=6, 8 and 10, the wait cycles for the broadcast slots increases low-load latency to 5-7.5 cycles. The best throughput is offered by BI=6, which is 1.36-2.14X higher than other BIs. This is because 6 cycles is *exactly* enough to cover the 2 broadcast slots, and 4 cycles to send broadcast flits from each CR serially up to the NIC. A BI lower than this throttles throughput, while a BI higher than this adds wait times. The greedy scheme offers 83% reduction in low-load latency, and similar throughput as baseline with Prio=Local[12].

Broadcast from all nodes. When all nodes inject, as shown in Figure 6b, the role of PVT and SVT come into play. SFO_Complete with BI=6 over the PVT has 62% lower latency and similar throughput as the baseline which is over a SVT. SFO_Greedy over SVT (with no redirection to the CR) provides the best latency of 5.6 cycles, and a throughput that is 1.22X better than baseline and 30% away from the ideal. SFO_Greedy with Prio=Local over the PVT shows 1.55X higher throughput than the baseline, and is only 10% away from the theoretical ideal on a mesh, demonstrating the throughput benefits of the PVT. The 3-4 cycle redirection latency to get to the nearest CR increases its low-load latency. This points to the interesting design space tradeoffs enabled by SMART where a low-latency redirection enabled a large throughput enhancement.

Multicast from all nodes. Figure 6c shows the performance for multicast traffic, with every node injecting to a randomly chosen set of destinations. Here SFO_Complete with BI=4 gives better throughput than BI=6 since fewer flits need to go the NIC. SFO_Greedy provides 75% reduction in low-load latency, and 11% higher throughput than the baseline. The SFO PVT is optimized for broadcasts, and does not offer significant throughput advantages to multicasts.

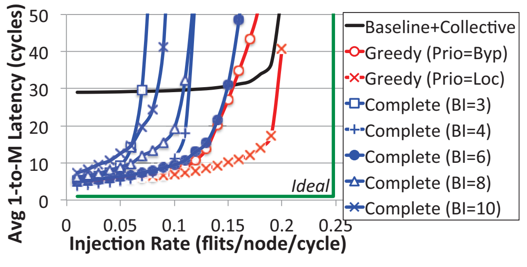
In summary, of all flavors of SFO, the greedy scheme over SVT provides the lowest latency, and the greedy scheme over PVT provides the highest throughput.

B. Synthetic Many-to-1 Traffic

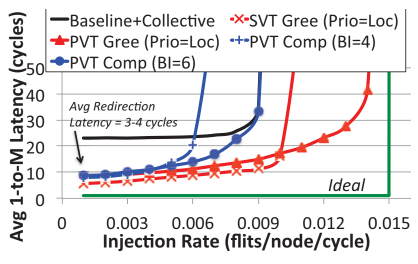
Synthetic many-to-1 traffic represents ACK traffic in coherence protocols or the MPI_reduce primitive in MPI. In this pattern, all nodes (except the destination, so 63 in this case) inject "ACKs" - with the same ACK_id - directed to a randomly chosen destination, at a specified injection rate. We do not inject a broadcast to reserve ART entries. Instead, we reserve them magically prior to the ACK injection, and free them according to the SFI scheme described earlier. In

[11]A smaller HPC_{max}, say 4, which we do not present in the interest of space, increases per-dimension latency to 2 cycles.

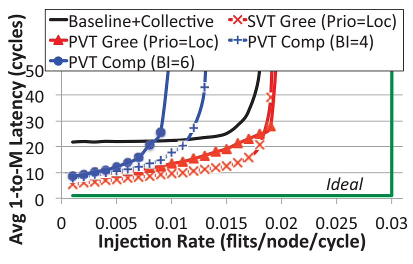
[12]Prio=Local offers better throughput than Prio=Bypass [15]. This is because Prio=Bypass can reserve output links for bypass flits, at the cost of locally buffered flits, yet no flit may actually show up due to some SSR interaction at the previous routers it is unaware about. Prio=Local would have sent the local flits on the output links instead.



(a) Broadcasts from 4 corner routers.

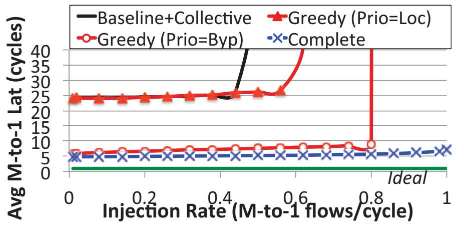


(b) Broadcast from all nodes.

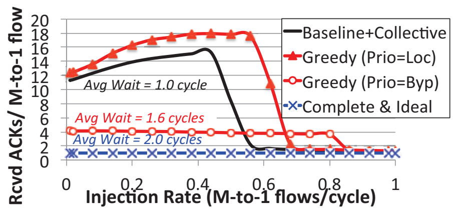


(c) Multicast (random # dest) from all nodes.

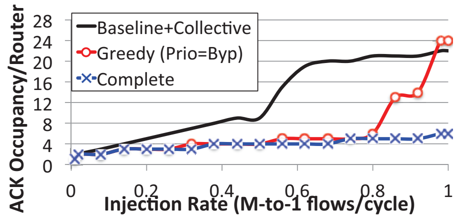
Fig. 6: **SMART-FanOut (SFO) with synthetic 1-to-many traffic.**



(a) Latency for receiving all 63 ACKs.

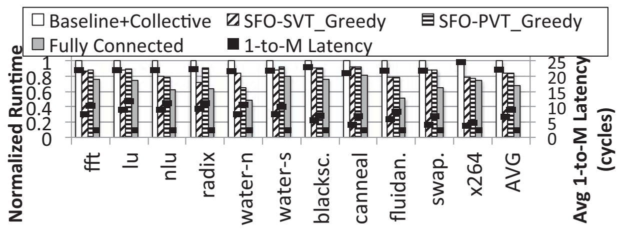


(b) ACKs received / M-to-1 Flows Injected.

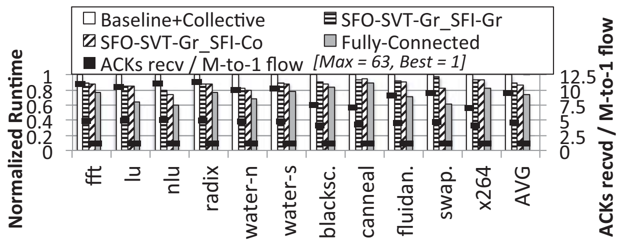


(c) Avg ART entries occupied.

Fig. 7: **SMART-FanIn (SFI) with synthetic many-to-1 traffic**



(a) Token Coherence (TC) Protocol Performance.



(b) HyperTransport (HT) Protocol Performance.

Fig. 8: **Full-system impact of SFO and SFI for a 8×8 CMP.**

Figure 7a, the x-axis plots the injection rate for each 63-to-1 flow, and the y-axis plots the *M-to-1 latency* - cycles it took to receive one (or more) ACKs with a combined count of 63. For the baseline, the average M-to-1 latency is ∼25 cycles, till the network saturates at an injection rate of 0.44 M-to-1 flows/cycle. SFI_Greedy (Prio=Local) works almost identical to the baseline since all ACKs are forced to stop at every router, to prioritize its locally buffered ACKs over the bypassing ones to use the links. With SFI_Greedy (Prio=Bypass), the very same locally buffered ACKs now get aggregated into the higher-priority bypassing ACK, as explained earlier in Section IV-B, reducing the latency to 5.7-8.9 cycles, before the network saturates at an injection rate of 0.8. With SFI_Complete, the latency goes down to 4.7-7.1 cycles, and the network does not saturate even at an injection of 1 M-to-1 flow every cycle. SFI_Complete reduces the effective injection rate by M enhancing throughput.

Figure 7b plots the *aggregation ratio*, i.e., number of ACKs received per M-to-1 flow. The SFI_Complete meets the ideal by delivering only *one* ACK with a count of 63 for each flow. SFI_Greedy (Prio=Bypass) delivers 4 ACKs on average - this number goes up with arbitrary delays between ACK injection which will be explored with full-system traffic later in Section V-C. The baseline and SFI_Greedy (Prio=Local) wait for 1 cycle on average at every router (during which time they opportunistically aggregate), and deliver 12-18 ACKs per flow before saturation. Once they saturate, ACKs are forced to wait for 4.5 cycles on average, increasing aggregation ratio to the ideal 1 ACK per flow.

Figure 7c plots the number of distinct ACKs[13] at each router, on average. This represents the total number of active M-to-1 flows in the system at any time. For the baseline, the number of ACKs goes up to 9 before saturation (at which point it shoots up to 20). For SFI, the number of ACKs (i.e., ART occupancy) is about 4.

In summary, the complete SFI scheme provides the lowest latency, and highest throughput for reduction traffic.

C. Full-system Performance

We run the parallel sections of SPLASH-2 [21] and PARSEC [22] through Wind River Simics [23] with 64 in-order SPARC cores, connected to the GEMS [20] timing model. We model 32kB Private I&D L1, and 1M Private L2 caches per tile. We run two coherence protocols: (1) Token Coherence (TC) [2], which has 52% broadcast flows, and (2) HyperTransport (HT) [1], which has 14% broadcast and 14% ACK flows, on average, across these workloads.

SFO on TC. Figure 8a plots the runtime and 1-to-M latency of SFO_Greedy with SVT and PVT, compared to the baseline and fully-connected NoCs, all running TC. SFO provides 16% runtime reduction over the baseline. There is no significant difference between the runtime in SVT and PVT implementations since most workloads are not network bandwidth limited.

[13]Two ACKs from the same flow cannot be simultaneously buffered at a router in both Base+Coll and SFI, as one would be aggregated into the other.

The exception is *water-nsq* where SVT_Greedy provides 16% runtime improvement while PVT_Greedy provides 35%. The average 1-to-M latency over PVT is 3 cycles more than over SVT due to the redirection.

SFO+SFI on HT. Figure 8b plots the runtime and ACKs received per M-to-1 flow for the SFI_Greedy (Prio=Bypass) and Complete schemes, with the SFO_Greedy scheme, compared to the baseline and fully-connected, all running HT. SFI_Greedy provides 11% runtime reduction on average, with SFI_Complete reducing it by a further 3% on average. The baseline, which has an optimized aggregator at each router, delivers 8-12 ACKs per M-to-1 flow. SFI_Greedy reduces this to 5-6, and SFI_Complete reduces it to exactly 1. While SFI_Greedy and SFI_Complete have similar performance, the latter consumes 50% less energy, and the former 29%, than the baseline, due to the fewer buffer writes/reads and link traversals. This is despite the crossbar energy per access in SFI being 1.58 times higher than in the baseline to account for the bypass muxes and repeaters driving wires multi-hop [15]. One ART entry per core suffices for all this performance gain and energy savings, and the average ART occupancy was 20 for SFI_Complete.

VI. Related Work

Networks with Multicast Support. On-chip routers with message forking support have been proposed recently in works like VCTM [3], bLBDR [5], MRR [4], RPM [6], FANOUT [8]. Our baseline is derived from these, and assumes a highly-optimized 1-cycle forking delay [6], [8]. Tree-based multicast routing algorithms like Whirl [8] and BAM [9] (optimized over RPM [6]) try to utilize the network bandwidth and buffers more efficiently. Our PVT has a similar goal, and succeeds in completely eliminating contention between multicasts, at the cost of redirection to a CR. SFO is the first work to demonstrate single-cycle traversals across multiple nodes of a multicast tree.

Networks with Reduction Support. In the off-chip domain, dedicated reduction networks for barrier synchronization have been used in supercomputers like NYU Ultracomputer [24] and IBM Blue Gene/L [25]. Aggregation of memory requests was done in IBM RP3 [26] and NYU Ultracomputer [24]. In the on-chip domain, barrier synchronization is performed over dedicated global broadcast wires [11], [12], [27] or transmission lines [10]. Com [9] and FANIN [8] perform in-network reduction without adding a dedicated reduction network, and form our baseline. In FANIN the first ACK for a flow to enter a router opportunistically aggregates other ACKs of the same flow for a heuristically defined wait time, before proceeding further. Com uses a central CAM at every router to perform an addition of ACK counts, and requires ACKs to return to the broadcast root. Our ART avoids a CAM lookup by ACK_id indexing, can handle ACKs flowing to a different node than the root, and can also handle any reduction operation. SFI is the first work to demonstrate reduction across multiple routers within the same cycle.

VII. Conclusion

In this work, we present SMART-FanOut and SMART-FanIn that enable forking and reduction respectively across multiple nodes in a dimension within a single-cycle, leveraging clockless repeated wires on the datapath. We explore greedy and complete forking/aggregation approaches, and conclude that a greedy forking and complete aggregation strategy provides the best performance at the lowest energy. Going forward, single-cycle collective communication enabled by SFO and SFI, without the overhead of dedicated networks, can provide scalability to limited-directory protocols and MPI as we scale core counts.

References

[1] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture," *IEEE Micro*, vol. 27, pp. 10–21, Mar. 2007.

[2] M. M. K. Martin *et al.*, "Token Coherence: Decoupling Performance and Correctness," in *ISCA*, Jun. 2003.

[3] N. Jerger *et al.*, "Virtual Circuit Tree Multicasting: A Case for On-chip Hardware Multicast Support," in *ISCA*, 2008.

[4] P. A. Fidalgo *et al.*, "MRR: Enabling Fully Adaptive Multicast Routing for CMP Interconnection Networks," in *HPCA*, 2009.

[5] S. Rodrigo *et al.*, "Efficient Unicast and Multicast Support for CMPs," in *MICRO*, 2008, pp. 364–375.

[6] L. Wang *et al.*, "Recursive Partitioning Multicast: A Bandwidth-Efficient Routing for Networks-on-Chip," in *NOCS*, 2009.

[7] F. A. Samman *et al.*, "Multicast Parallel Pipeline Router Architecture for Network-on-Chip," in *DATE*, 2008, pp. 1396–1401.

[8] T. Krishna *et al.*, "Towards the Ideal On-Chip Fabric for 1-to-Many and Many-to-1 Communication," in *MICRO*, 2011.

[9] S. Ma *et al.*, "Supporting efficient collective communication in NoCs," in *HPCA*, 2012, pp. 165–176.

[10] J. Oh *et al.*, "TLSync: support for multiple fast barriers using on-chip transmission lines," in *ISCA*, 2011, pp. 105–116.

[11] J. L. Abellán *et al.*, "Efficient and Scalable Barrier Synchronization for Many-Core CMPs," in *ICCF*, 2010, pp. 73–74.

[12] V. Krishnan *et al.*, "The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors," *Int. J. Parallel Program.*, vol. 29, pp. 3–33, Feb. 2001.

[13] R. Ho, "On-Chip Wires: Scaling and Efficiency," Ph.D. dissertation, Stanford University, Aug 2003.

[14] B. Kim and V. Stojanović, "Equalized Interconnects for On-Chip Networks: Modeling and Optimization Framework," in *ICCAD*, 2007.

[15] T. Krishna *et al.*, "Breaking the On-Chip Latency Barrier Using SMART," in *HPCA*, 2013, pp. 378–389.

[16] C.-H. O. Chen *et al.*, "SMART: A Single-Cycle Reconfigurable NoC for SoC Applications," in *DATE*, 2013, pp. 338–343.

[17] http://www.itrs.net.

[18] C. Sun *et al.*, "DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *NOCS*, 2012, pp. 201–210.

[19] N. Agarwal *et al.*, "GARNET: A Detailed On-chip Network Model inside a Full-system Simulator," in *ISPASS*, 2009, pp. 33–42.

[20] M. M. K. Martin *et al.*, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *CAN*, 2005.

[21] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995, pp. 24–36.

[22] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.

[23] http://www.windriver.com/products/simics.

[24] A. Gottlieb *et al.*, "The NYU Ultracomputer - Designing a MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers*, vol. 32, pp. 175–189, 1983.

[25] A. Gara *et al.*, "Overview of the Blue Gene/L system architecture," *IBM J. Res. Dev.*, vol. 49, pp. 195–212, Mar. 2005.

[26] G. F. Pfister *et al.*, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," in *ICPP*, 1985.

[27] M. A. Watkins *et al.*, "ReMAP: A reconfigurable heterogeneous multicore architecture," in *MICRO*, 2010.