2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Push Multicast: A Speculative and Coherent Interconnect for Mitigating Manycore CPU Communication Bottleneck

Jiayi Huang[†], Yanhua Chen[†], Zhe Wang[‡], Christopher J. Hughes[‡], Yufei Ding[*], Yuan Xie[§]
[†]HKUST(GZ), [‡]Intel Labs, [*]UC San Diego, [§]HKUST
Emails: hjy@hkust-gz.edu.cn, ychen484@connect.hkust-gz.edu.cn
{zhe2.wang,christopher.j.hughes}@intel.com, yufeiding@ucsd.edu, yuanxie@ust.hk

***Abstract*—As CPUs scale up to many cores, the bandwidth of the network-on-chip (NoC) and cache can soon become the performance bottleneck. In modern processors, the cache hierarchy plays a reactive role to supply data upon request. In parallel programs, shared data accesses from different cores at different times can consume large cache and NoC bandwidth for the same data. These same-data accesses inherently have redundancy and lead to inefficient cache and NoC bandwidth utilization.**

In this work, we propose *Push Multicast*, a speculative and coherent interconnect. We transform the last-level cache into a proactive agent to *push* data to other sharers upon replying to the demand requester. Pushing enables *effective multicasting* to reduce LLC and NoC bandwidth consumption. A coherent in-network filter is proposed to prune the outstanding requests in the routers along the way of the pushed data delivery. Moreover, a dynamic mechanism is designed to pause and resume pushing adaptively. Compared with a system with an L1 Bingo data prefetcher and an L2 Stride prefetcher, Push Multicast achieves an average of 33% NoC bandwidth saving, a geomean of 1.02× and a maximum of 1.56× speedup in a 16-core system. In a 64-core system, it further achieves an average of 43% NoC bandwidth saving, along with a geomean of 1.11× and a maximum of 2.08× speedup.

I. Introduction

The insatiable need for compute in datacenters continues to drive CPU core counts upward. For example, Intel's Sierra Forest with 144 cores was recently announced [32], as was AMD's Bergamo with 128 cores [1]. The large set of cores is connected through scalable network-on-chip (NoC) fabrics. While manycore server CPUs are typically optimized for multi-programmed scenarios, most support cache coherent shared memory, and their NoCs also cater to multi-threaded workloads. For CPUs with shared, sliced caches, such as a globally shared last-level cache (LLC), as the number of cores grows, the pressure on the LLC and NoC grows. The bandwidth of both the LLC and the NoC can become a performance bottleneck. Furthermore, aggressive SIMD optimizations such as advanced vector and matrix extensions (e.g., Intel AVX512/AMX [30]) accelerate computation, which may lead to more frequent data accesses to worsen the problem.

In modern processors, the cache hierarchy plays a reactive role in supplying data: caches return data after receiving a request. In multi-threaded workloads on a manycore CPU with a shared LLC, shared data reads happen when different private caches send read requests to the LLC for the same line. The LLC controller subsequently processes each individual request and replies with identical data responses separately. These same-data accesses and deliveries contain high redundancy in both request and data response traffic. They not only consume request processing and response injection bandwidth of the LLC controller, but also lead to inefficient NoC utilization.

In this work, we focus on multi-threaded workloads with a considerable portion of read-shared data. We aim to mitigate the bandwidth bottleneck of read-shared data accesses caused by capacity misses in private caches. When the working set is too large to fit in a core's private cache, shared data that has been previously read is evicted before the next use. This forces each core to send a request to the LLC for each shared data read, pressing heavily on the NoC and LLC. An effective way to tackle this problem is multicasting. Prior work on multicasts can be mainly classified into two categories: request coalescing for multicasts and software-assisted multicasts.

Request coalescing proposals include request combining in the NYU Ultracomputer [24] and GPU packet coalescing [38]. The NYU Ultracomputer combines later requests with an earlier registered one in network switches and uses a response to serve all the registered requests. Similarly, GPU packet coalescing combines requests at the shared LLC and replies to them using a multicast with a single cache access. However, they are not hardware coherent and cannot be applied to many-core CPUs. Additionally, they are designed for communication between distinct compute and memory nodes and have limited support for the all-to-all communication pattern resulting from the sophisticated cache coherence and compute-LLC co-location in manycore CPUs. Moreover, threads running on different cores usually have great variations of execution speed in a mesh-based non-uniform cache architecture. Due to this, request coalescing at the LLC or routers can rarely happen, and so few multicasts can be initiated to save bandwidth.

More recently, Wang *et al.* propose to have software represent memory access patterns as streams of accesses and offload each stream to the cache subsystem [58]. These streams can then proactively send data to cores and form a limited degree of multicast opportunistically. This is a general and viable technique, however, it is highly intrusive. It requires changes in most components of the system, including software,

DOI: 10.1109/HPCA61900.2025.00115

instruction set, core pipeline, load-store unit, cache hierarchy, and NoC. To avoid the complexity of software and system changes, in this work, we aim to design a lower-cost hardware-based solution for the LLC and NoC bandwidth problem.

What fundamentally limits opportunities for multicasts is the cache has no advance information of when and what shared data the cores need. If the cache can infer future read-shared data demands, it can enable effective multicasting. Previous work has explored coherence prediction to speculate on data sharers to reduce remote access latency, but has no support for multicasts to save bandwidth [36], [37], [41], [48]. They target producer-consumer and migratory sharing patterns caused by coherence misses and do not apply well to repeated read-shared accesses caused by capacity misses on large working sets of emerging workloads. Moreover, blindly applying multicasts to shared reads can overwhelm the network as concurrent same-line accesses from different cores can trigger massive redundant multicast traffic. Further, these prior techniques incur high hardware overhead with a dedicated predictor for each line.

In this work, we propose a hardware-based solution, Push Multicast, which is a proactive and coherent interconnect to mitigate the manycore bandwidth bottleneck through speculative multicasts. Specifically, we transform the LLC into a proactive agent to detect a shared data read from one core and *speculatively push* data to other sharers through a *multicast* to save LLC and NoC bandwidth. In addition, the router is augmented with a coherent filter to prune later-arriving requests to avoid explosive redundant multicasts; the pushed data carries the sharer information and acts as a shared cache line delegate floating in the network along the path of packet delivery. Moreover, the speculative push mechanism supplies data to the slowest sharers even before they request it, providing further bandwidth reduction and performance boost. Push Multicast also includes a dynamic feedback-based per-core push pausing and periodic resuming mechanism for adaptive multicasting. In our evaluation using Rodinia [14], OpenMP kernels [29], and PARSEC [9], results show that Push Multicast achieves a geomean of 1.02× speedup (up to 1.56×) with 33% NoC bandwidth savings on average in a 16-core system and a geomean of 1.11× speedup (up to 2.08×) in a 64-core system when compared to a system with an L1 data Bingo prefetcher [4] and an L2 Stride prefetcher.

In summary, the contributions of this paper are as follows.

- Detailed workload analyses reveal read-shared data access patterns in multi-threaded programs: large working sets lead to high private cache misses and the temporal sharer access locality reveals opportunities for shared data pushing and multicasting.
- Conceptualizing the ***push multicast*** principle with a speculative and coherent interconnect microarchitecture solution to demonstrate its potential for proactive cache design to enable effective read-shared data multicasting.
- A coherent in-network filter with a pushed data packet as a representative of a shared cache line to prune outstanding sharer requests along the way of pushed data delivery for traffic reduction.
- A dynamic feedback-based per-core push pausing technique and a periodic resuming mechanism for adaptive push multicasts.

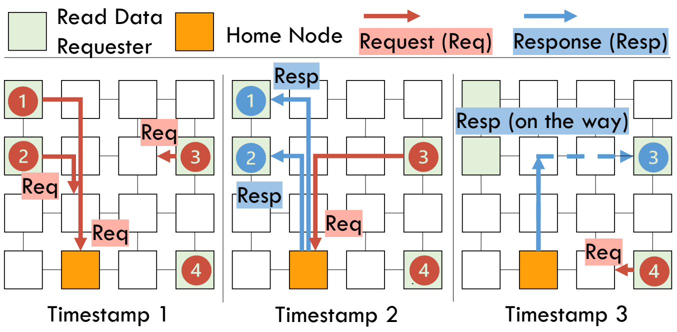


Fig. 1: Read-shared data accesses in a *reactive* conventional manycore CPU, where each core requests the same line and gets the response independently.

II. Background and Motivation

A. Manycore Architecture

A tiled manycore CPU comprises a grid of tiles connected to a network-on-chip (NoC) to facilitate communication between tiles. In general, its cache hierarchy employs several levels to store data on-chip and exploit data locality. Typically, in addition to a processor core, each tile also hosts private caches, such as L1 and L2 caches, and a slice of a shared last-level cache (LLC). In this work, we assume the LLC is shared by all cores. It is partitioned into slices and distributed across all the tiles; a given physical address maps to a single "home" slice according to an address hashing function.

These components are interconnected via a NoC. A request or data packet consists of one or more flow control units (flits), with the first and last flits designated as the head and tail flits, respectively. The head flit typically contains the accessed address and source/destination IDs. A read request is represented as a single-flit packet. One common flow control technique is virtual cut-through, which allows flits of the same packet to begin transmitting to the next hop without waiting for the remaining flits. Packets can be injected from an endpoint into the NoC or ejected from the NoC back to an endpoint, referred to as injection and ejection, respectively.

In modern manycore processors, read-shared data is typically served from the LLC. Fig. 1 shows an example where there are four read requests to a piece of shared data from different cores (❶, ❷, ❸, and ❹) at different times and the LLC slice replies to them one by one. Despite the cores wanting the same line, the number of responses is identical to the number of requests. Such shared data accesses have redundancy in both request and data response traffic and cause inefficiency in LLC and NoC bandwidth utilization, which can become a performance bottleneck.

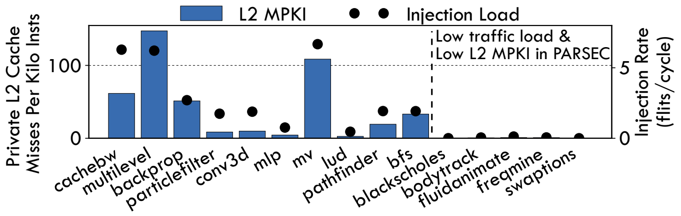


Fig. 2: Private L2 cache MPKI (bar, primary Y axis) and NoC injection load (dot, secondary Y axis).

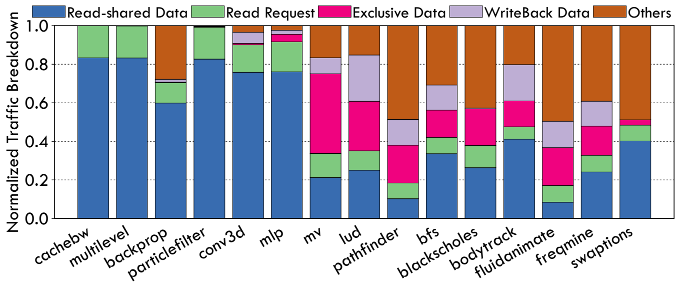


Fig. 3: Traffic breakdowns and shared data portion varies from 10% to 80%.

B. Motivation

We simulate multi-threaded workloads on a manycore CPU for cache traffic profiling to study the optimization opportunities for shared data accesses. The system configuration, simulation methodology, and workloads are described in §IV.

Large working sets can cause high private L2 cache misses and high network load. We measure the L2 misses per kilo-instruction (MPKI) for the benchmarks with large inputs, which lead to large working sets to stress the cache hierarchy and NoC. As shown in Fig. 2, the private L2 MPKI is more than 100 for some workloads. This leads to a large number of accesses to the shared LLC and thereby, a large volume of traffic in the NoC with moderate to high network load, except for the PARSEC benchmarks [9] with low traffic load.

Considerable portions of the LLC access traffic are read-shared data requests and responses. Similar to prior work [6], we further profile and classify the NoC traffic triggered by LLC accesses to understand which category plays a major role in the bottleneck. Fig. 3 shows the percentiles of the five traffic categories, including read-shared data, read request, exclusive data, write-back data, and others. Read-shared data packets are those generated from read requests to lines in the shared coherence state. *Read-shared data is from 10%–80% of the traffic, and the corresponding requests are also significant in all cases, showing the criticality of read-shared data accesses on bandwidth consumption.*

Read-shared data accesses from sharers to the LLC exhibit temporal locality. We simulate an OpenMP version of matrix-vector multiply (*mv*), a fundamental kernel in many deep learning [14] and HPC [35] workloads, and characterize its shared data access behavior. In this parallel kernel, each core reads the shared input vector and a private partition of the matrix; cores synchronize at the end of the parallel computation. Fig. 4 profiles the shared input vector access across all the cores. The violin plots show the distribution

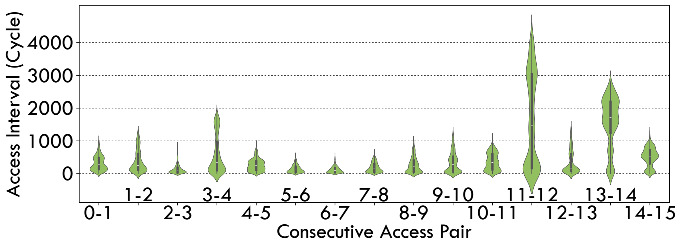


Fig. 4: Time interval distribution between consecutive shared data accesses, where 0-1 means time interval between accesses from sharers 0 and 1.

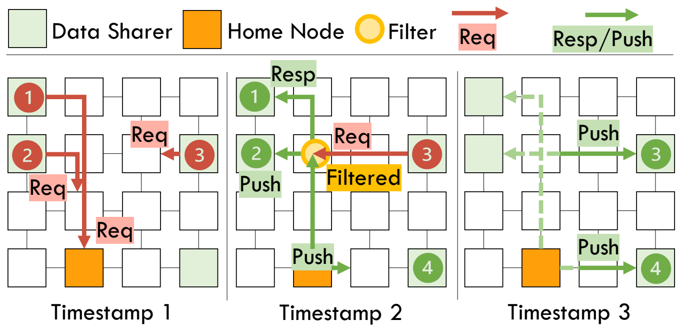


Fig. 5: Shared data accesses in the proposed *proactive* and coherent Push Multicast manycore architecture, where a demand request can trigger a multicast to speculatively push data to other sharers.

of time intervals between consecutive accesses from different cores to the shared lines. It reveals the time difference between consecutive accesses to the same data typically spans one thousand cycles, and the cumulative time interval from the first to the last access among all sharers can extend over several thousand cycles. This is due to thread variation caused by out-of-order execution and NUCA effect. *This implies the time gap between two consecutive same-line requests is much larger than LLC lookup time. Thus, packet coalescing at the LLC to initiate multicasts would be ineffective.*

This characterization shows the importance of read-shared data to the LLC and NoC and calls for optimizations specifically for this class of data. The observations of temporal locality across sharers motivate Push Multicast to reduce both LLC and NoC traffic. In particular, we see the potential of shared data multicasting to not only the core that requests it but also other sharers. Moreover, the time spread of requests inspires the push multicast concept of speculatively multicasting shared data to cores prior to their requests.

III. The Push Multicast Approach

A. Overview

Push Multicast is a proactive and coherent interconnect to address the LLC and NoC bandwidth bottleneck caused by read-shared data accesses. Fig. 5 presents an example to demonstrate the key concepts in Push Multicast and its potential. At timestamp 1, three read-shared requests are triggered by three sharers due to private L2 misses for the same line to the home node (aka LLC slice), and the first request (❶) arrives earliest. At timestamp 2, when the home node processes the first request, it generates a push multicast packet to reply to the demand requester (❶); this packet also pushes data to other sharers (❷, ❸, and ❹ in this example). The pushed data acts as a representative of the shared cache line along its way to the sharers. In this role, it can filter other outstanding read requests to the same line from its destined sharers. For instance, requests from ❷ and ❸ get filtered in a router when they meet the pushed data, and the push becomes a demand response for the pruned requests. Finally, at timestamp 3, the pushed data arrives at sharer ❹ before the core even requests the data. This turns what would be a miss into a hit, in contrast to the case in timestamp 3 in Fig. 1.

B. *Cache Push Mechanism*

Push Triggering Mechanism in LLC. The LLC needs to determine *when* to initiate a push and to *whom*. We separate the push mechanism into two phases: *sharer establishing phase* and *push activated phase*, where the sharer establishing phase prepares the sharer information for the push activated phase. In the sharer establishing phase, the set of sharers for a cache line has not reached steady state, i.e., the cache line is accumulating sharers. In the push activated phase, the program has reached a steady state for this line and triggers re-references due to private cache misses. Then, the LLC triggers pushes to the sharers, speculating on their future use of the line.

Specifically, the LLC checks if the read request is from a new sharer. If so, it replies with a unicast message and adds the requester ID to the sharer list. Otherwise, it initiates a push transaction to all sharers rather than a unicast response only to the demand requester. This inference method is simple as it is intended to target parallel workloads where threads perform similar tasks. This design introduces negligible hardware overhead as it leverages the existing directory (or snoop filter), which is used to keep track of the sharers of a cache line for coherence maintenance[1].

Push Handling Procedure in Private Caches. Conventionally, private caches initiate requests to the LLC, and expect a response to finish the transaction. This guarantees the processing and consumption of each response and no blocking happens. In Push Multicast, the LLC can also be an initiator that speculatively pushes a shared line to private caches that do not request it. A private cache should properly handle a push to ensure it does not block other incoming responses indefinitely and cause a deadlock. For example, it can happen that unexpected pushed data cannot be accepted into the cache because all the lines in the matched set are in blocking transient states and waiting for acknowledgments (Acks). If the pushed data blocks these Acks, a deadlock happens.

This can be resolved by separating the push and response packets into different virtual networks. This requires extra buffers in all the cache controllers and network routers, costing a large area overhead. In this work, we use data response

[1]The current design applies to coherence protocols with silent eviction from Shared state, which is the only state in which we trigger pushes. Extension for protocols with non-silent eviction is discussed in §VI.

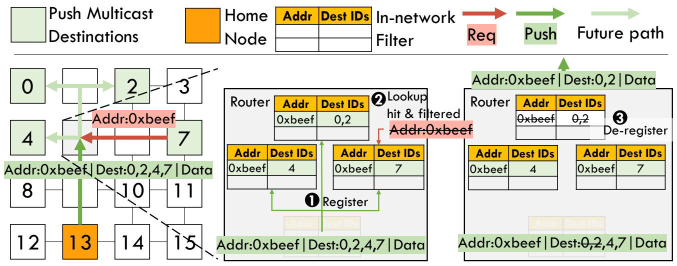


Fig. 6: An in-network filtering example where the push multicast packet destined for tiles 0,2,4,7 meets the read request from tile 7 in the router.

virtual channels for pushes and employ a simple dropping technique to avoid deadlocks. If a pushed line attempts to evict a blocked line, the pushed line is dropped to avoid possible deadlocks. Otherwise, it is accepted into the cache.

Note that no livelock can happen. A response, including a push triggered by a request from the local or a peer cache, is guaranteed to be accepted for an outstanding miss to the same line. This is typical in cache designs to avoid deadlock and livelock.

C. *Coherent In-Network Filter*

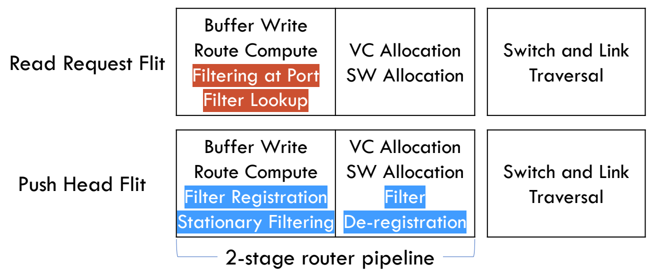
We propose a coherent in-network filtering mechanism in the routers to filter read requests that have their responses embedded in an outstanding push transaction, which acts as a representative of the shared line. When a push is on the way to its sharers' destination tiles in the network, these sharers could still trigger read requests to the same line prior to receiving the push. These requests are redundant and can cause unnecessary response traffic if they get served by the LLC. So, it's important to prune them away to save bandwidth.

The mechanism comprises three stages: filter registration, read request filtering, and filter de-registration. Fig. 6 shows an example to demonstrate the procedure of in-network filtering. In the example, a push transaction (with address `0xbeef` and destination bit vector for tiles {`0,2,4,7`}) is triggered by the LLC. Meanwhile, a read request is initiated from tile 7 to the same cache line as the push. The request and the push meet each other in tile 5's router and the filtering procedure happens as follows:

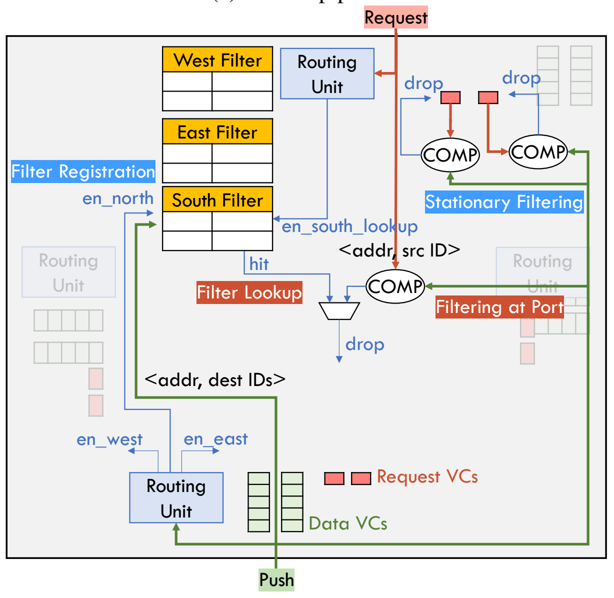
❶ **Filter Registration.** At timestamp 1, the push arrives at the router input port and its output ports are computed based on the routing algorithm and the destinations. Then, the address and the destinations associated with an output port are registered in the output ports' filters.

❷ **Read Request Filtering.** At timestamp 2, a read request to the same line from tile 7 arrives at the router. It looks up its input port's associated filter upon buffer write and finds the request address (`0xbeef`) and requester ID hit in the filter. Then, the request is dropped as the push packet carries the data response for the requester.

❸ **Filter De-registration.** At timestamp 3, the router makes a replica of the push packet and updates the destination in the original packet and the replica. After sending out the replica, the corresponding filter de-registers the entry.



(a) Router pipeline.



(b) Router microarchitecture.

Fig. 7: Push Multicast router pipeline stages (a) and micro-architecture (b).

The example in Fig. 6 only shows the filtering of read requests when they enter the router. It is important that the stationary matched read requests also get filtered. A "stationary" read request is one that enters the router earlier than a push of the same line. Besides filter registration for the push packet, it also needs to look for the matched stationary read requests and prune them away.

Fig. 7 shows the augmented 2-stage router pipeline for read request and push data packets as well as the microarchitecture. The filtering actions can run in parallel with the existing pipeline stages as shown in Fig. 7a. In the first pipeline stage, packets perform conventional buffer writes and route computations. A push head flit also registers in the corresponding filters, denoted as *Filter Registration*. Meanwhile, its meta data is forwarded to the request virtual channels (VCs) to prune the stationary read requests and the read requests that arrive in the same cycle, denoted *Stationary Filtering* and *Filtering at Port*, respectively. An incoming read request also performs filter lookup for pruning, denoted *Filter Lookup*. When a push packet leaves the router, it can de-register the filter, denoted *Filter De-registration*.

Fig. 7b shows the augmented router filter structures, where a push from port south and its interaction with the filters at port north are shown for clarity. At each port, we allocate a designated filter for each of the other ports. For example, at port North, we have three filters for the other three ports (South, East, and West). Each filter has a dedicated entry for each input data virtual channel (VC) of the corresponding port. For instance, South Filter has two entries for the two input data VCs of the South port. The filter entry is similar to a snoop filter entry, where the address is used as the tag and the destination ID bit vector is the content. The cacheline data is in the push multicast packet stored in the associated input data VC. A 5-port router with 4 input data VCs per port needs 20 filters and each filter has 4 entries. The detailed filtering actions shown in Fig. 7b are as follows:

- **Filter Registration**: Upon receiving a push head flit, its output ports are computed. Then, its meta data, address, and destinations are registered in the output ports' corresponding filters. For example, a push from the south port registers in the computed north outport's South Filter.
- **Filtering at Port**: Concurrent with *Filter Registration*, the meta data is forwarded to the input port of the computed output direction, north in this example, to filter a same-line read request arriving in the same cycle.
- **Stationary Filtering**: The meta data of the push packet is also forwarded to the input VCs of the computed output direction (north in this example), to filter the same-line read requests that arrive earlier than the push.
- **Filter Lookup**: An arriving request also checks the filter with its address and destination. If a hit, the read request is dropped and the reserved input VC is freed.
- **Filter De-registration**: The tail flit of a push packet can de-register the entry of the corresponding filter at the granted switch output port after it wins switch arbitration. This entry clearance is lazy to cover the link delay so that an incoming read request during link traversal can be captured when it arrives at the input port.

The path of a push should be the reverse of a read request for them to meet. We use deterministic XY routing for requests and YX for responses to maximize the chance of filtering. This design option is for performance rather than correctness. We have more discussion on routing design in §VI.

D. *Dynamic Pause-and-Resume Push Mechanism*

We propose a mechanism to pause and resume pushing to avoid cache pollution and overwhelming NoC traffic by useless pushes. Specifically, a feedback-based push pause knob at each private cache informs the LLC to exclude it from pushing, and a resume knob at the LLC periodically restarts pushing for those push-disabled sharers.

Push Pause Knob at Private Cache. The key idea is for each L2 to monitor if pushes are accurate and useful for it, and decide whether to turn pushing on or off for itself. A pushed line is useful if it serves an outstanding read-shared miss or it is accessed by the core before eviction. We use two counters to record the total number of received pushes and the number of useful ones, namely, `Total Push Counter (TPC)` and `Useful Push Counter (UPC)`. If the ratio of

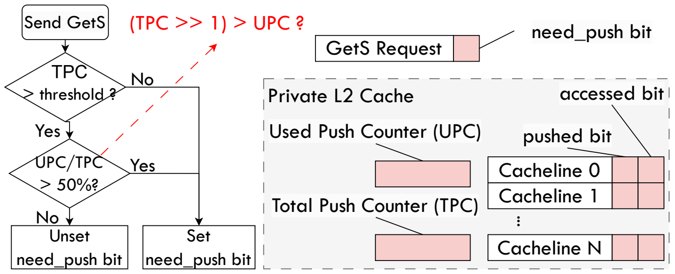


Fig. 8: Push pause knob and its workflow in private L2 cache.

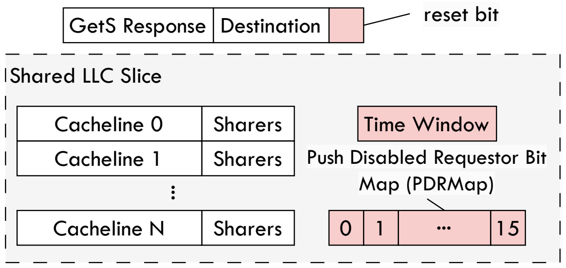


Fig. 9: Push resume knob in shared LLC slice.

the useful pushes over the total received is smaller than a threshold, the L2 disables pushes for itself by resetting an added need_push bit in later requests to inform the LLC to exclude it from pushing. The flowchart in Fig. 8 shows the workflow of the pause knob. A TPC Threshold is used to adjust the monitoring period, during which pushing is kept enabled if TPC is less than TPC Threshold.

To compute the ratio of UPC and TPC, we use simple logic. Specifically, for a 50% ratio threshold as used in our experiments, push is enabled if more than 50% of the pushed data is useful. We can check this by shifting TPC to the right by one and comparing to UPC. Other power-of-two ratio thresholds, such as 25% and 12.5%, can employ similarly simple hardware implementations.

Fig. 8 also shows the hardware modifications in the private cache, including the TPC and UPC. To track if a pushed line is useful, we add two status bits, a pushed bit and an accessed bit, to each cache block. The pushed bit is set when a pushed line is installed and the accessed bit is set when the line is accessed. Upon eviction, the UPC and TPC are incremented accordingly depending on the status bits. The two counters can be reset locally or by the LLC. A context switch resets the counters. Receipt of a reset signal, embedded in a response from the LLC, clears both counters. When the TPC is about to overflow, both TPC and UPC are right-shifted by one bit.

Push Resume Knob at LLC. We adopt a simple periodic push resume mechanism to re-enable pushing for private caches that have paused it. A knob is responsible for maintaining the push disabled requesters and periodically resuming. A Push Disabled Requester Bit Map (PDRMap) is used to record the push disabled requesters in each LLC slice. When a push is triggered, the destinations are calculated by excluding the disabled requesters in PDRMap from the sharers stored in the directory entry. The knob alternates between *Disable Accepting* phase and *Resume* phase, where each phase lasts for a predefined period. The predefined period is set in a Time Window counter; it counts down to zero to change phase and resets the counter. During the *Disable Accepting* phase, a private cache can turn pushing on or off for itself by feeding back a need_push flag to the LLC in a read request. If the flag is false, the knob adds the requester to the PDRMap; otherwise, the requester is removed from the PDRMap if it's present. During the *Resume* phase, when replying to a requester, a reset flag is embedded in the response to clear the TPC and UPC counters of the requester. Meanwhile, the requester is removed from the PDRMap if it's present. Adding to the PDRMap is blocked during the *Resume* phase. Figure 9 shows the knob hardware in the LLC. It adds minimum overhead: only an N-bit PDRMap for an N-core system, a Time Window counter, and a status bit to indicate the *Disable Accepting* and *Resume* phase for each LLC slice.

E. Asynchronous Push Multicast

We implement an asynchronous multicast algorithm with virtual cut-through to simplify the multicast complexity. Synchronous multicasting requires reservation of all the output ports of a multicast packet, which leads to a low success rate in a congested network. This can lead to a hold-and-wait situation and cause a multi-output-port dependence (or multi-output-virtual-channel in virtual-channel routers) and cause routing deadlocks. On the other hand, an asynchronous multicast breaks the dependencies and a multicast packet is kept in the buffer until all replicas are sent out to all the computed output ports. A multicast packet can be first replicated and sent to some of the output ports and retry after packet transmission for the remaining output ports. A full replica delivery requires the buffer to be large enough to hold the whole packet, and therefore a virtual cut-through is used. Note that wormhole flow control can also be used with single-flit data packets with more wiring resources [2], [20].

F. Cache Coherence

We first discuss race conditions that can occur due to pushing and introduce a handling principle to maintain coherence. Then, we detail our proposed coherence extension that implements the race condition handling principle.

1) Race Conditions and Handling Principle

In this work, we assume an invalidation based protocol. To ensure cache coherence, we need to enforce the *Single-Writer, Multiple Readers (SWMR) Invariant* and the *Data-Value Invariant* [49], where a read should obtain the most up-to-date written value.

In our design, race conditions can occur between pushes and reads, as well as between pushes and writes. It is important to note that pushes are triggered by demand reads on shared state and are treated as implicit reads. Therefore, there is no violation of the SWMR invariant, and coherence is not affected. However, the push-write race condition should be carefully handled to preserve the data-value invariant. When a write request is racing with an ongoing push transaction to the

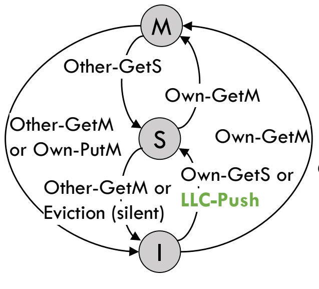
TABLE I: System Configuration Parameters

Parameter	Configuration
System	4×4 or 8×8 tiles @ 2 GHz system clock
OOO Core	3.5 GHz and 8-wide fetch/issue/commit 146/336 IQ/ROB and 144/112 LQ/SQ Entries
Cache Hierarchy	32KB 8-way private L1I/L1D 256KB 16-way private L2 1MB 16-way shared LLC/tile MESI protocol, 4 memory controllers at 4 corners
Prefetchers	L1 Bingo: 2KB spatial region and 16kB PHT L2 Stride: 16 streams, 4 prefetches/stream
DRAM	1600MHz DDR3 12.8 GB/s
NoC	4×4 (or 8×8) mesh with 2 GHz 2-stage router 1-flit/5-flit control/data packet 1-flit/5-flit control/data VC 3 virtual networks (vnet) with 4 VCs per vnet virtual cut-through flow control XY/YX routing for request/response 1-cycle link latency with 128-bit link
Dynamic Knob	`TPC` and `UPC`: 10-bit counter PushAck on 16-core (64-core): `TPC Threshold` = 64 (8) `Time Window` = 500 (1500) OrdPush on 16-core (64-core): `TPC Threshold` = 16 (16) `Time Window` = 500 (1500)

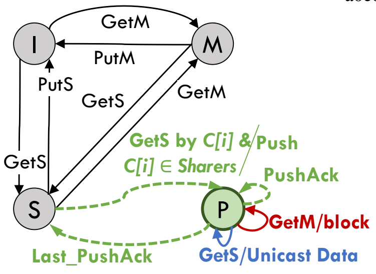
TABLE II: Workloads

Workload	Description	Input (256KB/512KB/1MB L2$)
backprop	NN training algorithm	64K/128K/256K
cachebw [28]	multi-threaded shared array scanning	8 MB array
multilevel [28]	multi-level buffers, partitioned to be scanned by distinct sets of threads	4 level with 2 MB each, 4 partitions
mlp [29], [43]	multilayer perceptron	batch size: 256/512/1024 in/out feature: 1K blocking factor: 8×8×8
mv [58]	matrix-vector multiplication	32×(64K/96K/192K) matrix 64K/96K/192K vector
conv3d [58]	3D convolution	input: 256×256, 16 channels kernel: 3×3, out channels: 64
particlefilter	statistical estimation of target location	2 1000×1000 frames 48000/192000/19200 particles
lud	lower–upper decomposition	1024/2048/2048 matrix dim
pathfinder	dynamic programming grid traversal	1.5M entries, 8 iterations
bfs	breadth-first search	1M/4M/4M nodes[†] approx. 600K/25M/25M edges
blackscholes	option pricing and financial modeling	simlarge
bodytrack	track a human body	simlarge
fluidanimate	simulate an incompressible fluid	simlarge
freqmine	frequent itemset mining in a graph	simlarge
swaptions	compute prices w/ Monte Carlo sim.	simlarge

[†] Inputs are generated using the graph generator from the github repository of Rodinia Benchmark for DPC++: https://github.com/artecs-group/rodinia-dpct-dpcpp (commit: a0e80bd) [12].



(a) Private cache.



(b) Directory/LLC.

Fig. 10: Push-Ack cache coherence protocol extension on MSI: coherence state transitions of private cache (a) and directory/LLC (b).

same line, a private cache may receive an invalidation before the push. Later-arriving pushed data would become stale and violate the data-value invariant.

All private caches *must* observe the same processing order for pushes and writes. To enforce this, we apply the following handling principle: ***serialize racing transactions*** [51]. By serializing pushes and writes, we enforce the data-value invariant and ensure cache coherence.

2) Coherence Extension

We propose two approaches for the needed serialization. The first one extends the protocol to block write requests until a push is completed. The second one relies on an ordered network to ensure private caches observe the same processing order as the request handling order at the directory.

Approach 1: Push Acknowledgment Protocol (PushAck). Fig. 10 shows the extended MSI protocol where actions that do not involve pushes are omitted for clarity. At a private cache, as shown in Fig. 10a, it can receive a push from the LLC (LLC-Push) and change state from I to S by installing the line. Private caches should also send a `PushAck` message to the directory to acknowledge the receipt of a push, which is not shown for simplicity. At the directory as shown in Fig. 10b, the extended protocol has a new transient state, shared push (`P`). A cache line enters `P` state from shared (`S`) state when the LLC triggers a push. The `P` state is a semi-blocking state. When a line is in `P` state, the LLC still serves read requests (`GetS`) to the line by responding with a unicast while other events are blocked (e.g., write requests). The cache line's state is changed back to `S` from `P` state after receiving acknowledgments from all sharers. This extension can be applied to various coherence protocols (e.g. MESI and MESIF) similarly.

Approach 2: Enforcing Push-Invalidation Ordering (OrdPush). This approach relies on an ordered network for serializing push and write operations. From the LLC's perspective, if a push is initiated before a write request, the push *will* arrive at a given private cache earlier than a younger write's invalidation. For this approach, we use deterministic routing so a push and invalidation to the same line are delivered on the same path. We enforce this order by stalling the invalidation if a push of the same line to the same output port is still in the router. Specifically, for an invalidation request packet, if its output port's associated filter has the aliased address registered, it is stalled. This order is free of protocol deadlock. As invalidations and pushes are in two separate virtual networks, the invalidation is ordered after the push and a dependency only exists from the control network to the data network. Thus, the OrdPush protocol is deadlock-free.

Both PushAck and OrdPush enforce the data-value invariant and write invalidations enforce SWMR invariant, thereby ensuring cache coherence. Note that consistency is not affected—pushing is similar to prefetching in that it simply installs data into cache ahead of time. Both approaches are evaluated in this study.

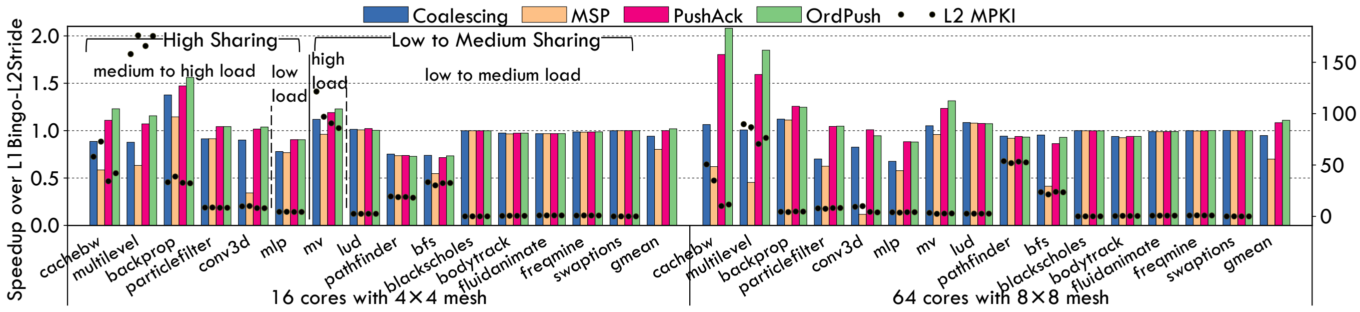


Fig. 11: Execution time speedup (primary) and L2 MPKI (secondary) normalized to L1Bingo-L2Stride.

IV. Evaluation

System Modeling and Configuration: We extend the Ruby cache system and Garnet 3.0 [8] network in gem5 v20.1 [44] for modeling. We configure an out-of-order CPU model that matches the real hardware performance [21], [26]. We swept the `TPC Thresholds` and `Time Windows` values to select the ones with best performance. TABLE I lists the system configuration parameters. In line with previous studies [56], [58], we set the L2 cache size to 256KB to ensure a reasonable simulation time for most workloads with substantial input, thereby putting pressure on the NoC and LLC. To evaluate the sensitivity of our technique to different cache configurations, we also perform experiments with 512KB/1MB and 1MB/2MB L2/LLC cache sizes. We implemented the filter buffer and comparison logic in RTL based on an open source router design [23] and synthesize them using Synopsys Design Compiler and the ASAP 7nm Predictive PDK [19]. Result shows that xdsethe 16.3% area overhead over a baseline router includes 8.8% for combinational logic, 1.5% for buffers, and 6% for other non-combinational logic. Note that router area is only about 3% of the tile area reported by ASIC Piton [5].

Workloads: We select several multi-threaded workloads from Rodinia [14] and a few OpenMP kernels and microbenchmarks that have good amount of data sharing and are throughput-oriented to stress the shared cache and NoC. We also include an irregular graph processing workload, *bfs*, to show the neutral effect of Push Multicast on irregular access patterns due to our dynamic feedback knob. To evaluate Push Multicast on general parallel programs, we test all PARSEC [9] benchmarks whose parallel phase can successfully run to completion in gem5 v20.1. For all the workloads, we warm up the cache by either running the first iteration for iterative applications or loading the input, and report performance results of the following parallel phase. For PARSEC, the parallel phase is the region of interest (ROI). We select simlarge input for PARSEC, and large enough inputs for other workloads to stress the cache and NoC. The workloads and their input data sets are summarized in TABLE II.

We evaluate both *PushAck* and *OrdPush*, comparing with a baseline with *L1Bingo-L2Stride* prefetchers, *Coalesce* [38], and *MSP* [41]. *L1Bingo-L2Stride* adopts a bingo prefetcher [4] at the L1 data cache and a stride prefetcher at the L2 cache. Their settings are listed in Table I. As the code footprint is relatively small compared to data footprint in our evaluated benchmarks, we only push for shared data access. Configurations other than L1Bingo-L2Stride do *not* use hardware prefetching. *Coalesce* groups concurrent requests for the same line at the LLC—only the first request accesses the LLC, but the returned data is multicast to all coalesced requesters. This represents the best previous approach for read sharing. *MSP* is a push implementation without multicasting and filtering; it treats a read request from an existing sharer as the first read of a previously seen/trained sequence to mimic the original MSP [41].

A. Performance Speedup and L2 Cache Miss Rate

Fig. 11 shows the performance of different configurations normalized to L1Bingo-L2Stride. In the 16-core system, Push Multicast achieves a 1.02× geomean speedup (up to 1.56×). Coalesce benefits *backprop*, where it achieves a 1.38× speedup; PushAck and OrdPush also have similar speedups over L1Bingo-L2Stride. Generally, OrdPush is better than PushAck. For workloads with high sharing and medium-to-high load (i.e., memory pressure), Push Multicast can outperform others due to its bandwidth savings. For example, on *cachebw*, for which all threads access the same data in the same order, OrdPush achieves 1.23× speedup. Compared with L1Bingo-L2Stride, Push Multicast achieves similar or better performance for bandwidth-insensitive workloads when the push accuracy is high, as in *backprop* and *particlefilter*. For high sharing with low load (*mlp*), L1Bingo-L2Stride prefetching is more effective. For *mlp*, the implementation has a low compute-to-memory-access ratio as no aggressive SIMD is used. So, it has a relatively light load and is sensitive to latency, making L1Bingo-L2Stride effective. If AI instructions like Intel VNNI or AMX are used, the benefit of Push Multicast would improve. The low-to-medium sharing, high-load case (*mv*) is bandwidth-hungry and can benefit from Push Multicast, while low-load workloads see a neutral effect.

For *bfs*, which has an irregular access pattern, L1Bingo-L2Stride has better performance than Push Multicast. This is because it can prefetch long adjacent lists for nodes with a large degree while Push Multicast has no effect on data touched by only one thread. Further analysis shows L1Bingo-L2Stride can effectively reduce L1D MPKI from 35 to 26.1, although L1Bingo-L2Stride and Push Multicast have similar L2 MPKIs, which includes both demand and prefetch misses

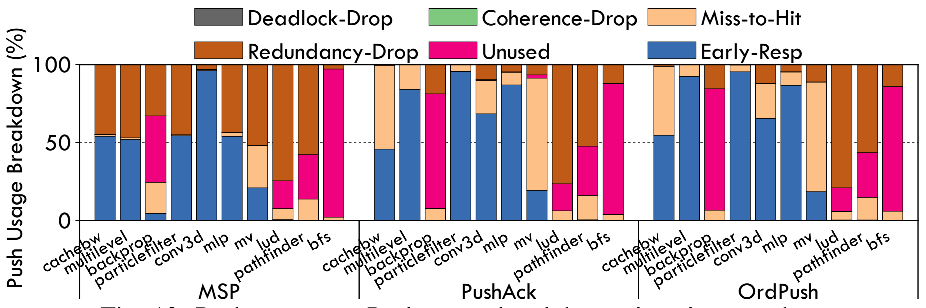


Fig. 12: Push accuracy: Push usage breakdowns in private caches.

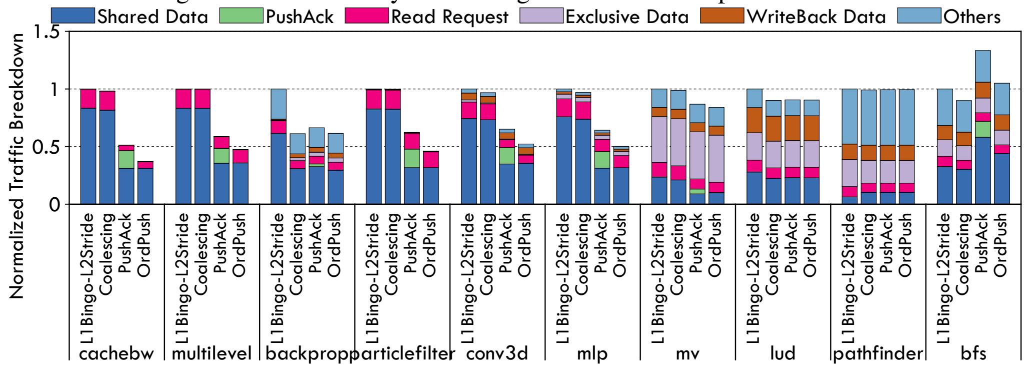
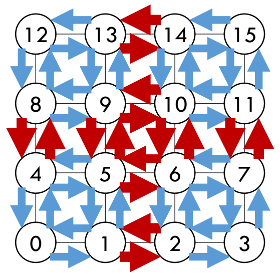
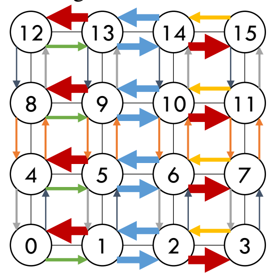


Fig. 13: Traffic breakdowns measured in flits and normalized to L1Bingo-L2Stride.



(a) L1Bingo-L2Stride link loads.



(b) OrdPush link loads.

Fig. 14: Average link loads of *cachebw* in (a) L1Bingo-L2Stride and (b) OrdPush.

from L1D. An ablation study in Section IV-E further explains Push Multicast does not bring extra overhead for a baseline without prefetching. Without support for multicasts and in-network filtering, MSP causes significant performance loss in most workloads. This is mainly due to too many redundant predictions leading to redundant traffic. On low-load PARSEC benchmarks, the effect of Push Multicast is neutral. Fig. 11 also shows the average L2 MPKI. On bandwidth-hungry workloads, Push Multicast is effective in reducing L2 misses.

Fig. 11 also shows the performance speedup for a 64-core system to demonstrate the scalability of Push Multicast. In larger systems, bandwidth problems can become more critical. In addition, Push Multicast can better exploit the higher degree of data sharing with more cores. For bandwidth-hungry workloads, Push Multicast achieves better improvements for the 64-core processor than the 16-core one, with up to 2.08× speedup over the L1Bingo-L2Stride. For low-load PARSEC workloads, Push Multicast is neutral. As PARSEC has low load and pushes are disabled most of the time, to save space and show clear comparisons, we exclude them and focus on the 16-core system in the rest of the paper.

B. Push Accuracy

Push accuracy is critical, since inaccurate pushes may pollute caches and consume extra bandwidth. Fig. 12 shows a categorization of pushes at the private caches. Deadlock-Drop, Redundancy-Drop, and Coherence-Drop are dropped pushes due to deadlock avoidance, a line already existing, and a conflict with a transient coherence upgrade, respectively. Unused means the data is evicted without being used. The beneficial categories are Miss-to-Hit and Early-Resp. Miss-to-Hit means a push correctly speculates the use of a line and turns a miss into a hit, saving miss penalty and read request traffic. Early-Resp indicates a read request is filtered in a router and responded to by a push. For most workloads with significant performance improvements (*cachebw, multilevel, mlp, mv, particlefilter*), push accuracy is close to perfect. For *backprop*, there is significant cache pollution. Even then, *backprop* sees benefits, likely because the private cache miss ratio is very high, and the traffic savings from multicasts compensates for overheads from unused pushes. In MSP, redundant requests can cause large amounts of traffic, consuming extra bandwidth and hurting performance.

C. Bandwidth[2]

Network Bandwidth. Fig. 13 shows the network traffic breakdown normalized to L1Bingo-L2Stride. For workloads improved by Push Multicast, the shared data traffic is reduced significantly, with a maximum of 60% in *cachebw* with OrdPush. One curious point is why OrdPush reduces traffic in *cachebw* by 60% while improving performance only by 25%. In addition to traffic volume, we also analyze the traffic load of each network link shown in Fig. 14. Fig. 14a and Fig. 14b show the network link loads for the baseline and OrdPush, respectively, where the thickness of the arrows indicates the load. In the baseline, traffic is distributed over the network with the bi-sectional links busiest (red arrows). Although OrdPush significantly reduces traffic through multicasts, YX routing plays a role in traffic distribution. As push packets replicate when they start to move in the X dimension, the traffic tends to increase toward the edge routers, making the links toward

[2]As MSP triggers overwhelming traffic, we exclude it in the bandwidth results to show clear comparisons for the remaining schemes.

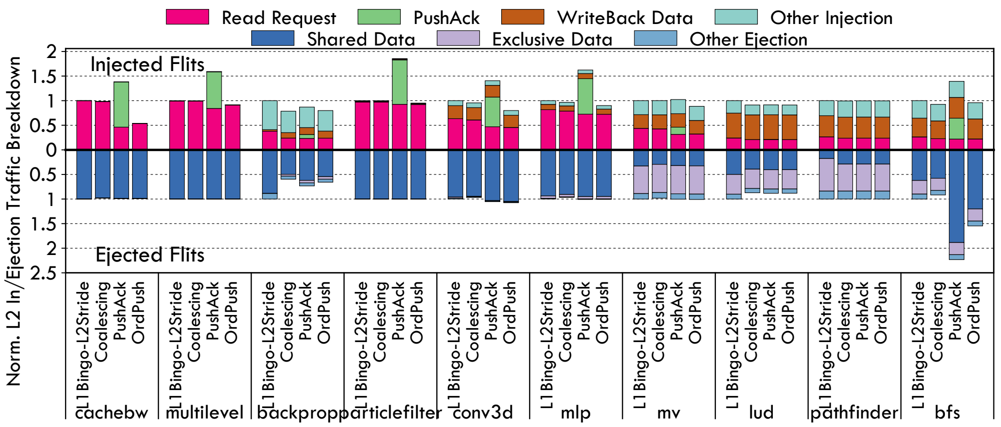


Fig. 15: L2 traffic breakdown over L1Bingo-L2Stride at injection and ejection.

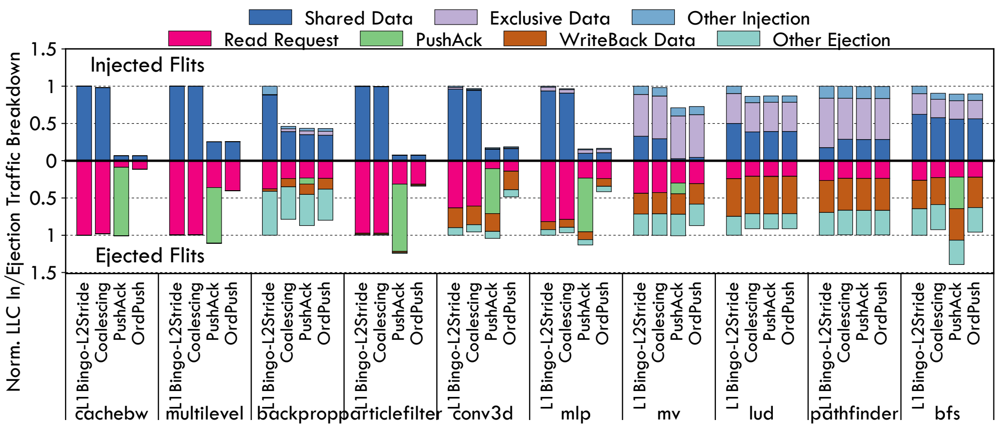
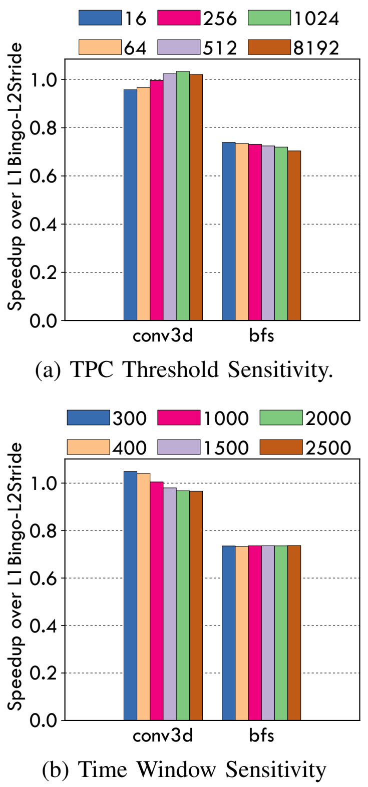


Fig. 16: LLC traffic breakdown over L1Bingo-L2Stride at injection and ejection.



(a) TPC Threshold Sensitivity.

(b) Time Window Sensitivity

Fig. 17: Sensitivity to TPC Threshold with a 2000-cycle time window (a) and Time Window with a 16 TPC threshold (b).

the east and west hotspots. Better traffic distribution should further improve performance.

Private L2 Cache Bandwidth. Fig. 15 shows the average L2 cache's injection and ejection bandwidth normalized to L1Bingo-L2Stride. PushAck may increase injection traffic due to push acknowledgements—every received pushed line incurs a PushAck message. Some of the lines also have a read request issued, so the total injection traffic may increase. OrdPush reduces injection traffic thanks to miss reduction because of accurate and early pushes. On the ejection side, the bandwidth is almost the same for workloads with accurate pushing because multicasts do not reduce the traffic at the destination.

LLC Bandwidth. Fig. 16 shows the injection and ejection bandwidth of the LLC normalized to L1Bingo-L2Stride. LLC injection traffic savings are from read-shared data due to multicasts. A sharing degree of 16 can reduce this by 16×. For ejection bandwidth, PushAck messages in PushAck can incur extra bandwidth consumption.We also observe fewer read requests arriving at the LLC, especially in push-friendly workloads (*cachebw*, *multilevel*, *particlefilter*, and *conv3d*). Most shared lines have 16 sharers in *cachebw* and *particlefilter*, and 4 sharers in *multilevel* with 16 cores. We profiled the average number of destinations in each data response per LLC read-shared request. Results show 15.4, 13.7, and 4 for *cachebw*, *particlefilter*, and *multilevel*, respectively, i.e., very close to the theoretical maximum, explaining the significant bandwidth savings.

D. Sensitivity Study

TPC Threshold and Time Window Sensitivity. We conduct sensitivity analyses on the TPC Threshold and Time Window with two benchmarks sensitive to these parameters, as shown in Fig. 17. Lower TPC thresholds can disable pushing for a private cache earlier, but this may occur during the warm-up phase when confidence for pushing is still low. As shown in Fig. 17a, a smaller TPC threshold benefits *bfs* by disabling pushes sooner. In contrast, *conv3d* risks losing optimization opportunities if pushing is disabled too early when confidence is low. To address this, we introduce a time window to resume pushes, allowing for faster recovery with a smaller window. Fig. 17b illustrates that a small time window can restore performance benefits for *conv3d* even with a low TPC threshold, while still maintaining performance for *bfs*.

NoC Bandwidth Sensitivity. Fig. 18 shows the speedup of PushAck and OrdPush over L1Bingo-L2Stride under different link bandwidths. For *cachebw* and *multilevel*, as the NoC bandwidth increases, the benefit over baseline also improves. This is because both workloads are bandwidth bound even with higher bandwidth, which can alleviate the hotspot similar to

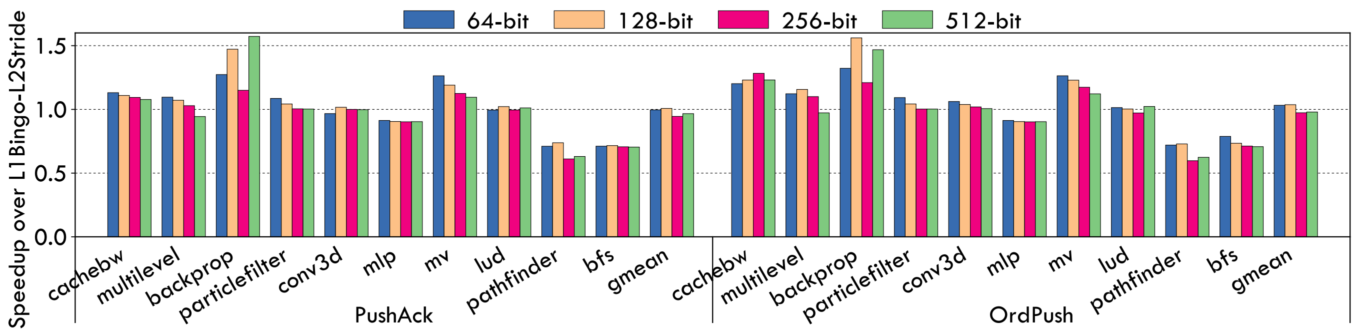


Fig. 18: Speedup of PushAck and OrdPush normalized to L1Bingo-L2Stride over different link widths.

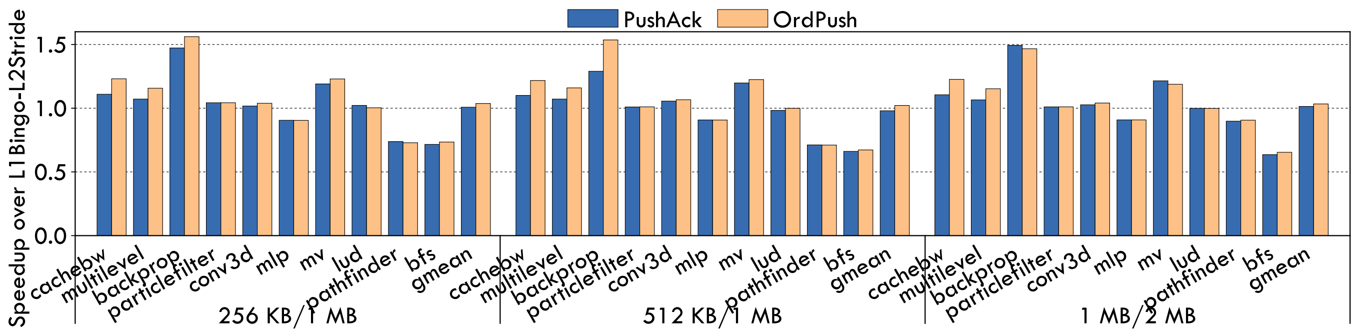


Fig. 19: Speedups of PushAck/OrdPush normalized to L1Bingo-L2Stride varying L2/LLC-slice cache sizes in a 16-core system.

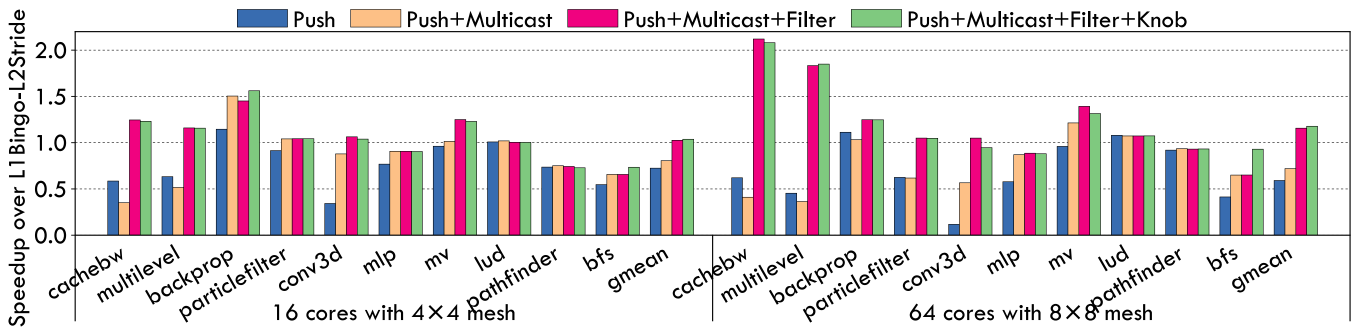


Fig. 20: Ablation studies of OrdPush on 16-core and 64-core systems.

the one depicted in Fig. 13b. On the other hand, the speedup of *mv*, *mlp*, and *particlefilter* over baseline decreases as link bandwidth increases. Although *mv* has the highest network load, it has limited data sharing. Increasing link bandwidth improves the behavior of private data, which is dominant, and the benefit of pushes diminishes. As link width increases, *particlefilter* becomes less bandwidth limited. Although pushes can improve latency and turn misses into hits, an out-of-order core can hide much of the LLC hit latency, especially for the evaluated core with an aggressive design (TABLE I). So, performance improvements decrease. As Push Multicast does not help or harm other benchmarks, they are not sensitive to link bandwidth changes.

Cache Configuration Sensitivity. To assess Push Multicast's sensitivity to cache configurations, we simulate a 16-core system with larger cache sizes, including 512KB/1MB and 1MB/2MB for L2/LLC (per core). With larger cache sizes, we configure larger input data for some workloads to stress NoC and LLC; the input configurations are listed in Table II. The results in Fig. 19 demonstrate that Push Multicast shows a consistent trend across cache configurations, highlighting its potential for larger system setups.

E. Ablation Studies

We conducted studies by adding pushes, multicasts, filtering, and dynamic control one at a time to show the synergy of these proposed features. Fig. 20 shows 16-core and 64-core simulation results for OrdPush. Pushes can degrade performance for applications with moderate to high loads because they can flood the NoC. Push+Multicast can turn the Push degradation to improvement by reducing traffic for workloads with moderate loads, such as *mv* and *mlp*. Interestingly, it causes more degradation for high-load kernels like *cachebw* and *multilevel*. Push+Multicast can mitigate network contention and more read-shared requests are sent to trigger more redundant push multicasts. In contrast, Push makes the network more congested and delays shared-read requests causing fewer redundant pushes. Adding the in-network filter can effectively prune same-line read-shared requests in the network to eliminate redundant pushes and achieve significant improvements. Lastly, our dynamic control can pause push in time to avoid performance harm as shown in *bfs*. In summary, the proposed design components work in synergy to achieve performance gain for push-friendly workloads while avoiding performance degradation on others.

V. Related Work

Decoupled Access/Execute (DAE) Architecture. DAE architectures separate memory accesses and computation into two program streams on two processors, and the two communicate through queues [25], [27], [54] to overlap memory accesses and computation. Recent DAE proposals offload regular access patterns as memory streams to memory access units to continuously provide data to computation units [50], [56]–[58]. Helix-RC proposes to propagate shared data in a ring network upon a store [11]. This does not solve the bandwidth problem of repeated read-shared sequences due to private cache eviction rather than writes. Moreover, these techniques require intrusive changes in both software and hardware. Instead, we propose a hardware-only solution.

Prefetching. Data prefetching is a widely used optimization to hide memory access latency [3], [33], [34], [62]. Some prior studies leverage software optimizations [10], [56] or exploit domain knowledge [18], [39] to tailor a prefetch strategy to improve the accuracy for data-intensive workloads (e.g., graph analytics [7]). In comparison, our work focuses on read-shared data access traffic reduction to alleviate NoC and LLC bandwidth problems rather than hiding latency.

Network Optimization. The NYU Ultracomputer, a distributed shared memory machine, combines later requests with an earlier request to the same address in the network switches, behaving similar to MSHRs; a following response can be delivered to all the registered requesters to reduce latency [24]. Note that it is not hardware coherent. To maintain coherence, the home node needs to see all requests to a given line, or sharer information can be lost. Similarly, in GPUs, packet coalescing and response multicasting can be used to optimize NoC bandwidth [38], [55]. Both techniques multicast for requests arriving within a small time window and do not speculate about other sharers.

Coherence Prediction and Update-Based Protocol. Prior work proposes to predict read sharers following a write or upgrade triggered by a store in producer-consumer and migratory sharing patterns [36], [37] such as memory sharing prediction [41]. This is similar to an update-based protocol [17] to tackle coherence misses. In contrast, Push Multicast targets read-shared data accesses due to capacity misses. Specifically, we target re-references due to private cache eviction on large datasets instead of invalidations. Applying prior techniques to this problem directly without multicasts and in-network filtering can over-predict and generate redundant data traffic and consume extra bandwidth. In addition, they aim to hide remote access latency while we mitigate the manycore LLC and NoC bandwidth problem.

In-Network Coherence Optimization. NoC optimizations have been proposed for specific coherence protocol flows. An in-network coherence protocol maintains sharers as a tree in the routers instead of a centralized directory in the home node [22]. It enables requests to be served by a nearby sharer rather than the home to reduce latency. However, each shared line has duplicate tag entries in routers between the home node and the sharers, even for the routers whose associated private cache does not hold the line. This incurs large hardware overhead and is not scalable. Multicasts and combining have been applied to invalidation packets and the corresponding acknowledgments, respectively [40], [45]. Push Multicast multicasts shared data packets. In iNPG [61], the first read-exclusive request can register in the router and the router can generate invalidations for later requests to the same line, saving snoop time and improve performance. In contrast, Push Multicast focuses on shared data accesses.

Cooperative Caching. Cooperative Caching [13] can obtain data from a nearby private cache. It incurs overhead for a Central Coherence Engine to maintain the location of data. This can reshape on-die traffic, but does not reduce the number of messages. In comparison, Push Multicast mitigates both NoC and LLC bandwidth pressure.

Data Direct IO (DDIO) and Cache Stashing. Pushing data to specific caches can save memory access time. In commercial systems, Intel DDIO technology enables IO to directly store data in the LLC instead of memory [31]. Cache stashing also provides hints to store data at particular caches [2], [52]. IO-driven cache injection also provides a similar capability [42]. Instead of being initiated by a single destination, Push Multicast targets shared data and pushes data to several sharers speculatively. To mitigate cross-core communication in the producer-consumer sharing pattern, Virtual-Link [59] attaches hardware queues to the NoC for groups of producers and consumers to facilitate their direct communication. Building on this concept, SPAMeR [60] further speculatively pushes messages to a predicted consumer based on the Virtual-Link framework. Other related work also targets producer-consumer sharing patterns, including cache injection [46], [47], curious caching [15], [16], and speculative pushes [53], [60]. In contrast, Push Multicast targets a multi-reader sharing pattern and optimizes the efficiency of read-shared data accesses through speculative push multicast. This approach effectively mitigates the LLC and NoC bandwidth issues caused by high capacity misses from read-shared accesses associated with large working set sizes.

VI. Discussion and Future Directions

Push Multicast provides a new way to optimize shared-memory multi-threaded applications: having a shared level of the cache hierarchy speculate about future accesses to shared data. However, there is significant room for future research to refine the concept.

Pushes for GPUs. GPGPU programs often trigger similar behavior across GPU compute engines at any given point in time. This makes it easy to infer the sharing behavior and shared data access pattern. This is a good match for Push Multicast. Further, the simple design of Push Multicast would be a great fit for GPGPU throughput processors, which we plan to explore in the future.

Pushes and Routing Design. We use XY for request and YX for response to maximize the chance of filtering. If we use another scheme or adaptive routing, it only affects performance, not correctness. For example, an unfiltered GetS can trigger another push in OrdPush and a unicast in PushAck if the previous push is outstanding (i.e., the cache state is P). An interesting topic is designing push-aware request and multicast routing algorithms. Despite multicasts saving bandwidth, the final copies that branch out from a multicast packet go to the original number of destinations. As shown in Fig. 14, the network bottleneck may be shifted to new locations. Designing a multicast algorithm with balanced load while maximizing filtering is challenging.

Instruction Hints and Prefetching Multicast. Instruction hints can be used to steer hardware. For example, instructions can embed hints for cache management such as prefetching, cache bypassing and cache stashing [2], [52]. Similarly, an instruction could provide a hint to the LLC for pushing. One interesting direction is to exploit the read-sharing insight and explore software and hardware prefetching techniques to trigger speculative multicasts. For instruction hints and software prefetch-triggered multicasts, challenges include how to specify the sharers, especially in a virtualized environment, and how to efficiently handle duplicate multicasts. For a hardware-based approach, smart sharer prediction is required for accurate speculative multicasts.

Interplay of Push and Prefetch. This work focuses on throughput-oriented workloads with significant sharing. Such workloads often exhibit regular access patterns that facilitate accurate prefetching. However, their bandwidth-intensive nature can exacerbate network and LLC contention due to prefetching, delaying demand requests or responses. For these workloads, prefetching at different cache levels offers limited benefits. In contrast, multi-level prefetching is more effective for latency-sensitive workloads. Enhanced prefetching can hide more latency while reducing the impact on bandwidth-sensitive workloads. However, effective push multicast consistently delivers substantial benefits in bandwidth-sensitive scenarios. An intriguing future direction is to combine push multicasting and prefetching. Our preliminary findings indicate that enabling both pushing and prefetching does not consistently yield good performance in all cases. For example, with accurate prefetching, pushing for prefetching requests can bring performance gains for high sharing and medium to high load cases, such as *cachebw, multilevel, particlefilter*. But for other high load or high sharing scenarios (*mv, conv3d*), the combination cannot easy to bring benefits. For low load cases, the combination can get benefit of prefetching. In summary, their effective interplay is nontrivial and necessitates precise prefetching or intelligent throttling mechanisms, which we propose for future research.

General Push Multicast. The current Push Multicast only targets read-shared sequences caused by capacity misses. This work can be extended to cover more patterns such as producer-consumer and migratory sharing. We can also explore enabling push multicasts for LLC misses. A sharer predictor decoupled from the directory can be designed to support protocols with either silent or non-silent evictions of shared lines and enable the possibility of push multicasting on LLC misses.

Multi-Level Caches. Our design only pushes data one level up from the shared cache. In multi-level private caches, data can be propagated to upper levels. For systems with multiple levels of shared cache, our design can be applied to each shared level. Further research can explore sharer prediction that decouples from the directory to propagate and trigger push multicasts across several shared levels. Challenges include how to guarantee the design is free of deadlock and livelock, and proper handling of race conditions.

Push Multicast on Other NoCs. Although this paper focuses on meshes, our design can be adapted to other NoC architectures, including rings and chiplet-based networks. For example, in a ring topology, the filter can be implemented in a ring stop and the request and response routing can be reversed. For a chiplet-based network, the die-to-die interface or boundary router can be augmented with an in-network filter and the multicast can be applied across chiplets.

VII. Conclusion

Shared data accesses are a major bandwidth consumer in manycore processors. Our characterization shows that they can be inferred and that speculative multicasting is promising for reducing bandwidth consumption. We conceptualize push multicasts and propose a hardware solution to send a speculative multicast to cover future shared data accesses. The pushes act as a representative of a shared line in the network. This enables routers to filter on-the-fly read requests to the same line from a requester that is included in the push. Our evaluation shows that Push Multicast is effective for parallel programs whose threads perform similar tasks with a geomean of 1.02× (1.11×) speedup in a 16-core (64-core) system. We also outline the future direction of Push Multicast towards shared data access optimizations for general parallel programs.

Acknowledgments

We thank the shepherd and anonymous reviewers for their valuable comments and suggestions. We sincerely thank Xinfeng Xie for the discussions on benchmarks in the early phase of this work. We especially appreciate Zhengrong Wang for providing the gem5 version with AVX-512 support and the related benchmarks. This work was supported in part by the National Key R&D Program of China (No. 2024YFB4505800), the National Natural Science Foundation of China (No. 62402411), the Guangdong Provincial Talent Program (No. 2023QN10X252), the Guangzhou-HKUST(GZ) Joint Funding Program (No. 2024A03J0624), Research Grants Council of HKSAR (16213824), and a gift from Intel.

References

[1] AMD, "“Bergamo”' 4th Gen AMD EPYC™ 97x4 Processors: Built for Cloud Native Workloads," https://community.amd.com/t5/epyc-processors/quot-bergamo-quot-4th-gen-amd-epyc-97x4-processors-built-for/ba-p/612701, 2023, [Online; accessed 3-August-2023].

[2] ARM, "AMBA 5 CHI Architecture Specification, Issue F," 2022.

[3] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 176–186.

[4] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[5] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "Openpiton: An open source manycore research framework," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 217–232, 2016.

[6] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2009, pp. 86–97.

[7] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[8] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[10] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 40–52, 1991.

[11] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "Helix-rc: An architecture-compiler co-design for automatic parallelization of irregular programs," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 217–228.

[12] G. Castaño, Y. Faqir-Rhazoui, C. García, and M. Prieto-Matías, "Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing," *Journal of Parallel and Distributed Computing*, vol. 165, pp. 120–129, 2022.

[13] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 264–276.

[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, 2010, pp. 1–11.

[15] D. Chiou and B. S. Ang, "Method and apparatus for curious and column caching," Apr. 9 2002, US Patent 6,370,622.

[16] D. Chiou *et al.*, "Extending the reach of microprocessors: Column and curious caching," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[17] G. Chirkov and D. Wentzlaff, "Seizing the bandwidth scaling of on-package interconnect in a post-moore's law world," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 410–422.

[18] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung, "A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching," *ACM Transactions on Computer Systems (TOCS)*, vol. 22, no. 2, pp. 214–280, 2004.

[19] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[20] W. Dally, "Reflections on 21 years of NoCs," 2022, keynote II at the 2022 International Symposium on Networks-on-Chip (NOCS). [Online]. Available: https://youtu.be/Nk3oQm9NxcY

[21] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[22] N. Eisley, L.-S. Peh, and L. Shang, "In-network cache coherence," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 321–332.

[23] A. Galimberti, F. Testa, and A. Zeni, "RTL Router Design in SystemVerilog," https://github.com/agalimberti/NoCRouter, 2017, [Online; accessed 17-July-2024].

[24] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The nyu ultracomputer—designing a mimd, shared-memory parallel machine," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 27–42, 1982.

[25] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 503–514.

[26] U. D. C. A. R. Group, "gem5 skylake config," https://github.com/darchr/gem5-skylake-config.

[27] T. J. Ham, J. L. Aragón, and M. Martonosi, "Desc: Decoupled supply-compute communication management for heterogeneous architectures," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 191–203.

[28] A. Heinecke, "ArchBenchSuite," https://github.com/alheinecke/ArchBenchSuite/tree/485cb48.

[29] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 981–991.

[30] Intel, "Intel® Architecture Instruction Set Extensions and Future Features Programming Reference," Tech. Rep., September 2022.

[31] Intel, "Intel® Data Direct I/O Technology," https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2022, [Online; accessed 21-November-2022].

[32] Intel, "Four Takeaways from Intel's Investor Webinar," https://www.intel.com/content/www/us/en/newsroom/news/four-takeaways-from-intel-investor-webinar.html, 2023, [Online; accessed 3-August-2023].

[33] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 247–259.

[34] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[35] K. Kabir, A. Haidar, S. Tomov, and J. J. Dongarra, "On the design, development, and analysis of optimized matrix-vector multiplication routines for coprocessors," in *High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, ser. Lecture Notes in Computer Science, J. M. Kunkel and T. Ludwig, Eds., vol. 9137. Springer, 2015, pp. 58–73.

[36] S. Kaxiras and J. R. Goodman, "Improving cc-numa performance using instruction-based prediction," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE, 1999, pp. 161–170.

[37] S. Kaxiras and C. Young, "Coherence communication prediction in shared-memory multiprocessors," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6*. IEEE, 2000, pp. 156–167.

[38] K. H. Kim, R. Boyapati, J. Huang, Y. Jin, K. H. Yum, and E. J. Kim, "Packet coalescing exploiting data redundancy in gpgpu architectures," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–10.

[39] N. Kohout, S. Choi, D. Kim, and D. Yeung, "Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2001, pp. 268–279.

[40] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 71–82.

[41] A.-C. Lai and B. Falsafi, "Memory sharing predictor: The key to a speculative coherent dsm," in *Proceedings of the 26th annual international symposium on Computer architecture*, 1999, pp. 172–183.

[42] E. A. León, K. B. Ferreira, and A. B. Maccabe, "Reducing the impact of the memorywall for i/o using cache injection," in *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*. IEEE, 2007, pp. 143–150.

[43] LIBXSMM, "LIBXSMM," https://github.com/libxsmm/libxsmm/tree/4ac333b.

[44] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[45] S. Ma, N. E. Jerger, and Z. Wang, "Supporting efficient collective communication in nocs," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[46] A. Milenkovic and V. Milutinovic, "Cache injection on bus based multiprocessors," in *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*. IEEE, 1998, pp. 341–346.

[47] V. Milutinovic, A. Milenkovic, and G. Sheaffer, "The cache injection/cofetch architecture: initial performance evaluation," in *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 1997, pp. 63–64.

[48] S. S. Mukherjee and M. D. Hill, "Using prediction to accelerate coherence protocols," in *Proceedings of the 25th annual international symposium on Computer architecture*, 1998, pp. 179–190.

[49] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Springer Nature, 2020.

[50] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 416–429.

[51] N. Oswald, V. Nagarajan, and D. J. Sorin, "Protogen: Automatically generating directory cache coherence protocols from atomic specifications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 247–260.

[52] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim, "Location-aware cache management for many-core processors with deep cache hierarchy," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[53] R. Rajwar, A. Kägi, and J. R. Goodman, "Inferential queueing and speculative push for reducing critical communication latencies," in *Proceedings of the 17th annual international conference on Supercomputing*, 2003, pp. 273–284.

[54] J. E. Smith, "Decoupled access/execute computer architectures," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 231–238.

[55] L. Wang, X. Zhao, D. Kaeli, Z. Wang, and L. Eeckhout, "Intra-cluster coalescing to reduce gpu noc pressure," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 990–999.

[56] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 736–749.

[57] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, "Near-stream computing: General and transparent near-cache acceleration," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 331–345.

[58] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream Floating: Enabling Proactive and Decentralized Cache Optimizations," in *Proceedings of the 27th International Symposium on High Performance Computer Architecture (HPCA-27)*, February 2021.

[59] Q. Wu, J. Beard, A. Ekanayake, A. Gerstlauer, and L. K. John, "Virtual-link: A scalable multi-producer multi-consumer message queue architecture for cross-core communication," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 182–191.

[60] Q. Wu, A. Ekanayake, R. Li, J. Beard, and L. John, "Spamer: Speculative push for anticipated message requests in multi-core systems," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–12.

[61] Y. Yao and Z. Lu, "inpg: Accelerating critical section access with in-network packet generation for noc based many-cores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 15–26.

[62] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.

Artifact Appendix

A. Abstract

The artifact contains the codes for Push Multicast, along with its setup and running descriptions. We provide instructions and click-to-run scripts for reproducing the main results presented in our paper. Specifically, we reproduce the results for Fig. 2, Fig. 3, Fig. 11–13, and Fig. 15–20.

B. Artifact check-list (meta-information)

- **Program:** Multi-threaded workloads from Rodinia [14] and PARSEC [9] benchmark suites, and a few OpenMP kernels and microbenchmarks from Gem Forge Framework [58] and Intel's library [29].
- **Run-time environment:** We provide a Docker for users, which removes large burdens in setting up the environment.
- **Compilation:** We include the installation of specific compiler (like LLVM) in our click-to-run script.
- **Data set:** We include the data set in our project.
- **Experiments:** We include the command for simulation in our click-to-run script. We also provide a standalone script for running simulations.
- **Metrics:** We evaluate the execution time, traffic, MPKI, and push accuracy in our evaluation.
- **Output:** The outputs of the artifact are figures in PDF format that reproduce the main results of our paper.
- **How much disk space required (approximately)?:** The disk space should be larger than 2 TB.
- **How much time is needed to prepare workflow (approximately)?:** The preparation includes building a Docker image, cloning the Git repository, and compiling. With a core count of 64, we estimate the time required to be approximately 2 hours.
- **How much time is needed to complete experiments (approximately)?:** The simulation time varies among different benchmarks. The shortest time spent on some benchmarks is a few minutes, while the longest time spent on others can be up to a week. In total, we have 444 jobs for all experiments, and we limit the core count to 64, so we estimate the time required to be approximately 11 days.
- **Publicly available?:** Yes.

C. Description

1) How to access

The artifact is archived in Zenodo[3]. It can also be accessed from GitHub, as the command shown below:

```
$ git clone https://github.com/redbird-
    arch/push-multicast-artifact.git
```

2) Hardware dependencies

For reference, we list our system configurations here:

- OS: CentOS Linux release 7.9.2009
- CPU: Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz (64 cores); Other CPU would work.
- DRAM: 2 TB
- Disk: 2 TB

[3]https://doi.org/10.5281/zenodo.14355343

3) Software dependencies

The software dependencies are resolved by the Docker environment we provide. Users are required to support Docker commands on their machines if they are using the provided Docker environment.

D. Installation

We provide two scripts: one to build the Docker image and another to enter the Docker container. The commands are shown below:

```
# Build the docker image
$ bash build-docker-image.sh

# Enter the docker container
$ bash enter-docker-container.sh
```

E. Experiment workflow

We provide two scripts: one executes the compilation and runs a quick experiment to produce the results shown in Fig. 4, while the other runs the remaining experiments. The commands are shown below:

```
# Compilation and quick experiment
$ bash run-violin.sh

# Remaining experiments and figures
$ bash run-remain.sh
```

We also provide a single script for clicking to run the full artifact. The command is shown below:

```
# Compilation, experiments, and figures
$ bash run-all.sh
```

We also provide individual scripts for each step in the workflow. The commands are shown below:

```
$ cd push-multicast

# Compile benchmarks
$ bash benchmark-compilation.sh

# Compile gem5
$ bash gem5-compilation.sh

# Run quick experiments
$ bash run-experiment-violin.sh

# Run remaining experiments
$ bash run-experiment-remain.sh

# Generate figures.
$ bash plot-figure.sh
```

F. Evaluation and expected results

The results and figures can be found in the directories `m5out` and `figures/reproduce`, respectively.

G. Notes

The `README.md` file of the artifact provides additional information on the organization of the code and detailed steps for running experiments.