Stream Floating: Enabling Proactive and Decentralized Cache Optimizations

Zhengrong Wang
UCLA
seanzw@ucla.edu

Jian Weng
UCLA
jian.weng@cs.ucla.edu

Jason Lowe-Power
UC Davis
jlowepower@ucdavis.edu

Jayesh Gaur
Intel
jayesh.gaur@intel.com

Tony Nowatzki
UCLA
tjn@cs.ucla.edu

***Abstract*—As multicore systems continue to grow in scale and on-chip memory capacity, the on-chip network bandwidth and latency become problematic bottlenecks. Because of this, overheads in data transfer, the coherence protocol and replacement policies become increasingly important. Unfortunately, even in well-structured programs, many natural optimizations are difficult to implement because of the reactive and centralized nature of traditional cache hierarchies, where all requests are initiated by the core for short, cache line granularity accesses. For example, long-lasting access patterns could be streamed from shared caches without requests from the core. Indirect memory access can be performed by chaining requests made from within the cache, rather than constantly returning to the core.**

Our primary insight is that if programs can embed information about long-term memory stream behavior in their ISAs, then these streams can be *floated* to the appropriate level of the memory hierarchy. This decentralized approach to address generation and cache requests can lead to better cache policies and lower request and data traffic by proactively sending data before the cores even request it.

To evaluate the opportunities of stream floating, we enhance a tiled multicore cache hierarchy with stream engines to process stream requests in last-level cache banks. We develop several novel optimizations that are facilitated by stream exposure in the ISA, and subsequent exposure to caches. We evaluate using a cycle-level execution-driven gem5-based simulator, using 10 data-processing workloads from Rodinia and 2 streaming kernels written in OpenMP. We find that stream floating enables 52% and 39% speedup over an inorder and OOO core with state of art prefetcher design respectively, with 64% and 49% energy efficiency advantage.

I. Introduction

Despite the slowing of technology scaling, and in some ways because of it, commodity general purpose processors continue to scale in number of cores (64 cores in AMD EPYC Milan, 56 in Intel Cooper lake, 72 in Knights Landing [54]), as well as last-level cache capacity (quarter of a GB in EPYC Rome). In these systems, the on-chip network bandwidth and latency become increasingly problematic bottlenecks.

Fundamentally, higher core count means more on-chip communication, causing longer latency and higher bandwidth utilization. This exacerbates coherence protocol inefficiencies. Widening the network bandwidth is the common approach, but this does not help much to reduce the overheads of smaller control messages. Private caches, which are critical to avoiding communication overheads, are often not used effectively due to thrashing. For example, we show that on a 64-core system with data-processing workloads from Rodinia and more, 72% of evicted cache lines have no reuse, and caching non-reused data contributes 50% of total network traffic.

On the other hand, many workloads exhibit structure and regularity that is possible to exploit, provided that the cache system is *proactive* and *decentralized*. For example, programs often contain long well-defined patterns of memory access. If the shared last level cache (henceforth L3) was aware of these patterns, this could be exploited by proactively streaming the data with few control messages. This requires some decentralization, as the cache itself would make requests. As another example, much of the data held in the L3 has little finer-grain reuse. If the private caches could be proactively warned of this behavior, this data would not need to be stored there. As a final example, relevant especially in data processing kernels, many cores may stream through the same data at relatively the same time. If the cache was aware of longer-term behavior, this can be proactively detected, and the cache can make a decentralized decision to combine the requests and efficiently service them through multicasting.

Unfortunately, it is not obvious how to implement such style of optimizations with conventional *reactive* and *centralized* cache systems. The reactive nature of caches – that they take actions based on downstream fine-grain (cache-line grain) requests – prevents the cache from being aware of and exploiting long term behavior. Even prefetchers, which try to learn access patterns, are typically activated as a response to cache misses. Furthermore, all memory requests and responses are centralized at the core, even if the core needs to do nothing but initiate the subsequent access.

Goal and Approach: To enable proactive and decentralized cache optimizations, we argue that caches need to be aware of decoupled components of programs corresponding to common access patterns. For this, we can leverage prior decoupled-stream ISAs [52,60], which integrates streaming memory patterns into general purpose ISAs. These prior works use decoupled-streams to enable efficient programmable prefetching. In this work we take this principle to its logical endpoint: allow streams to be decoupled from the core, *floating* them into cache hierarchy. Our goal is to explore how floating streams can enable proactive and decentralized optimizations, ultimately enabling higher efficiency in many-core systems.

To this end, we develop extensions to a tiled multicore's memory system to allow decoupled-streams to float into the shared last-level (L3) cache banks. Streams are managed with stream engines (SEs) at three places in the core/cache hierarchy: the L3 cache bank (SE_{L3}), the private L2 (SE_{L2}), and within the core (SE_{core}). The core's stream engine (SE_{core}), can generate binding prefetches based on the program's streams. SE_{core} can choose to *float* streams to SE_{L3}, which will then proactively generate read requests for the requesting core. These responses

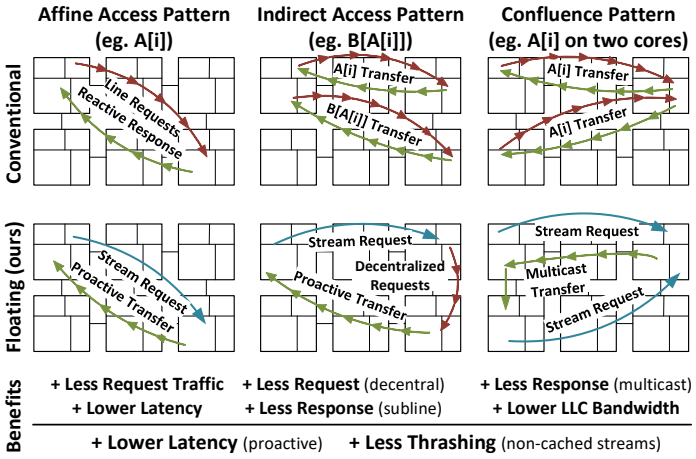
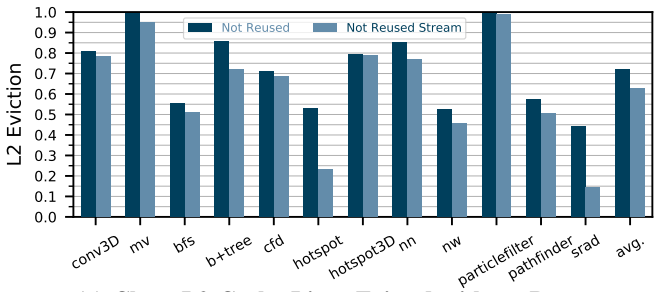
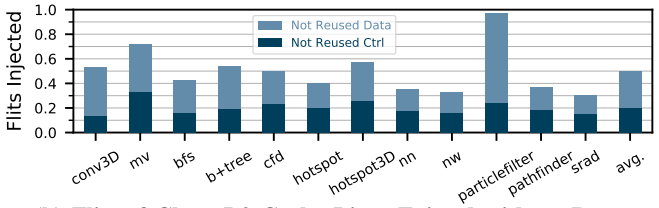


Fig. 1: Stream Floating Optimizations



(a) Clean L2 Cache Lines Evicted without Reuse



(b) Flits of Clean L2 Cache Lines Evicted without Reuse

Fig. 2: Overhead of Caching Data without Reuse

will be buffered by the SE_{L2}, where responses from remote streams will reside before being consumed by SE_{core}.

In this context, we explore three main optimizations, shown visually in Figure 1 along with their benefits:

- **Affine Floating:** Enable an affine stream pattern without reuse to be floated to L3. This optimization is by far the most prevalent in our workloads, while the following are targeted optimizations.
- **Indirect Floating:** Enable a stream to load from an address dependent on a prior stream, decentralized from the originating core. Only the requested subline is returned.
- **Stream Confluence:** Enable streams reaching L3 with the same pattern to coalesce and multicast the responses.

We view stream-floating as a new avenue for achieving less request and response traffic, lower effective access latency, and less L3 bandwidth demand. Enabling these optimizations to work efficiently means overcoming several challenges which we address in this work. This includes: how to avoid the communication overheads of stream offloading and maintaining flow control; how to decide when to offload streams by leveraging both static and dynamic information; how to interface with the coherence protocol; and how to detect when streams have confluence and avoid overheads of stalling cores.

Methodology and Findings: To evaluate the system, we develop an LLVM-based compiler to recognize and extract stream-based patterns and embed them in an X86 ISA, and an execution-driven, cycle-level simulator based on gem5 [12,39] [1]. We evaluate on data-processing workloads programmed using OpenMP, including several workloads from Rodinia. We also compare against a novel version of a state-of-the-art prefetcher (Bingo [9]), where we augment this baseline with an optimization for bulk-prefetch to reduce coherence traffic.

Across a range of data-processing kernels, stream floating improves the performance of a 4-way inorder, 4-way OOO, and 8-issue OOO by 52%, 41% and 39% respectively, and by 64%, 51%, and 49% in terms of energy efficiency. These results include a state-of-the-art multicore prefetcher [9]. Whereas the best prefetcher *increases* the traffic by 28%, stream floating with subline transfer reduces NoC traffic by 36%.

[1]Open source at: https://github.com/PolyArch/gem-forge-framework/

Contributions:

- The principle of stream floating: transparently offloading long-term access patterns into the memory hierarchy.
- Three novel optimizations (affine and indirect offloading, stream confluence) which reduce the data and request traffic.
- Detailed evaluation across inorder/OOO cores, and comparison to an enhanced state-of-the-art prefetcher.

Paper Organization: We first motivate by discussing overheads in existing reactive caches, and overview how we address these with our three optimizations (§II). We then discuss background on the stream-specialized core (§III). Following that, we develop the hardware extensions and policies necessary for stream floating (§IV), as well as coherence considerations (§V). Finally, we present methodology (§VI), evaluation (§VII), and discuss related work (§VIII).

II. Motivation and Overview

Stream floating is the concept of offloading decoupled portions of the program, specifically address generation and memory requests, near the relevant data. We first motivate by discussing existing inefficiencies in caches. Then we overview the primary optimizations that we explore in this work.

A. Motivation: Reactive Cache Inefficiency

Conventional cache systems reactively attempt to exploit locality: they make a best effort approach to keep data recently used by the core in the hopes that it will be reused later. However, for the N-1 cache levels for which the working set of a program phase does not fit, the data stored there is nearly guaranteed to be evicted with zero reuse. This leads to thrashing, wasting both cache capacity and network bandwidth.

To show the potential inefficiency for working sets that fit in LLC, we simulate 12 data processing workloads on a 64-core CMP with private L1, L2 caches and shared L3 banks (see §VI for hardware parameters, simulation and workload details). The results reveal three major overheads:

- **Cache Thrashing:** Figure 2a shows the number of L2 cache lines evicted in a clean state without being reused, normalized to total L2 evictions. Overall, 72% of evicted

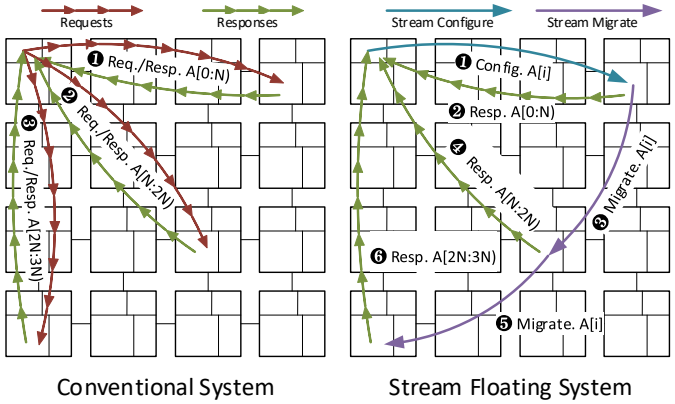


Fig. 3: Affine Floating Optimization

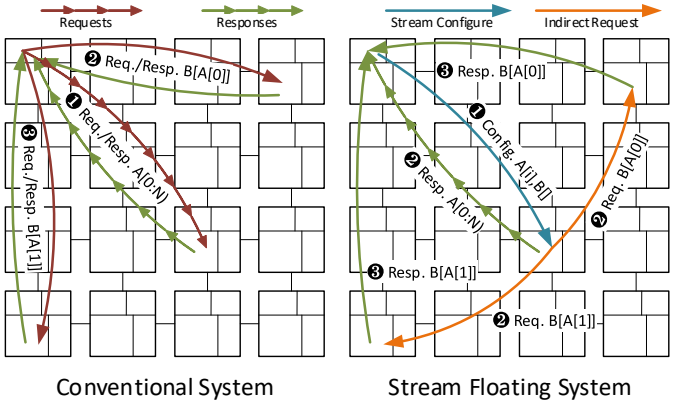


Fig. 4: Indirect Floating Optimization

cache lines have not been reused at all. This cache pollution can hurt performance and energy efficiency.

- **Tracking Coherence State:** Caching data without reuse also implies unnecessary tracking of coherence state, which incurs significant overheads in possible invalidation and writeback for peer cache controllers. Figure 2b measures the flits injected into the NoC due to caching not reused data, normalized to total flits. Flits are classified as data and control flits (for coherence). Caching not reused data contributes 50% of total network traffic, and 20% is from control messages. Notice that this is an underestimate, as it does not include the traffic generated from replacing the "victim" line, which could include useful data.
- **Redundant Request Messages:** Even if we ignore all the overheads mentioned before, the NoC traffic is still not optimal. Existing memory systems require one request per cache line, even if the pattern is very simple.

The fundamental reason behind such inefficiency is that current cache systems are designed to be *reactive* – driven by individual requests from the core. They lack a holistic view of the access pattern, duration of the pattern, presence of dependent accesses and reuse, etc. Without such key information, the cache can only react passively to external events with suboptimal policies. Hence, ISAs with richer abstractions can help to provide this information.

Stream Behavior: In this work, rather than trying to have the caches derive pattern information, we use a specialized ISA (discussed in Section III) that encodes *streams* explicitly. *Streams* are well-defined patterns of memory accesses. They can be as simple as an affine pattern `A[i]` or an indirect pattern like `B[A[i]]`. Figure 2a shows the fraction of the cached data without reuse corresponding to streaming patterns. On average it is 63% out of 72%, indicating that in the applications we target, streams are widely applicable to cover most of the required memory access behavior.

B. Optimization Overview

With streams as the abstraction for floating, we discuss three optimizations that can improve network traffic, coherence overheads, and data prefetching.

Affine Floating: Figure 3 demonstrates floating an affine stream `A[i]`. In a conventional system, the core issues a sequence of requests to the remote L3 banks to fetch the data (multiple arrows on one line). The stream's data may be distributed among multiple tiles, due to address interleaving in the shared L3. Responses are driven by individual requests.

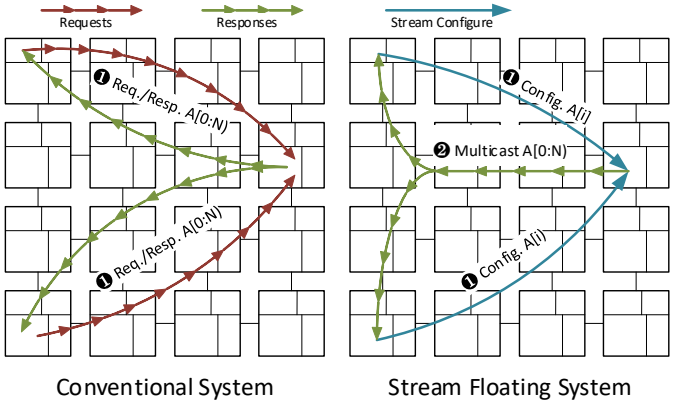


Fig. 5: Stream Confluence Optimization

In stream floating, the core first provides stream information to the cache, including the access pattern, length, etc. After configuration, the cache independently streams data back to the core, without excessive request messages. After some iterations, the stream may attempt to access an address outside the range of that bank (determined by address-interleaving granularity). At this point, the stream will migrate to the appropriate L3 bank to keep fetching data until completing. Stream floating replaces many request messages by a one-time configuration and a few migration messages, and the cache proactively prefetches.

Indirect Floating: Figure 4 shows floating an indirect stream `B[A[i]]`. Normally, the core first gets data from the index array `A[]`, computes the indirect access address, and finally accesses array `B[]`. The cache does not know the access pattern and cannot generate addresses on behalf of the core: the core centralizes all requests.

With stream floating, the indirect stream can be offloaded together with the affine stream. Once the affine stream data is ready, the remote L3 cache can *simultaneously* stream back `A[]` and fetch `B[A[]]` on behalf of the core (both labeled as ❷ in Figure 4). This shortens the chain for indirect accesses.

Stream Confluence: In multi-threaded workloads, it is common that different threads are requesting the same data. However, in existing systems, these accesses are independent from each other, as Figure 5 shows. The cache lacks sufficient information to detect and coalesce identical accesses, as

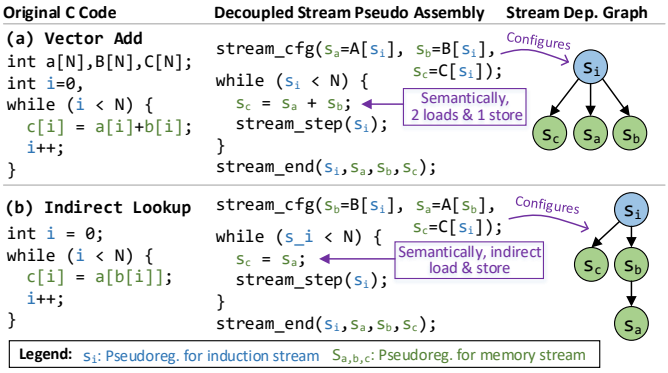


Fig. 6: Decoupled-Stream ISA Examples

individual requests from different cores are short-lived and arrive at different times.

Streams, on the other hand, encode access patterns and are much easier to be compared and coalesced. Streams accessing the same data tend to have the same parameters: e.g. start address and stride. Also, streams are generally long enough to describe long-term behaviors, which exposes more multicast opportunities. In Figure 5, streams accessing the same data can be transparently merged by the cache, turning them into one multicast stream and further reducing the NoC traffic.

Overall, leveraging streams as a coarse grain unit of offloading can empower proactive and more intelligent caches.

III. Background: Decoupled Streams

In this section, we discuss background on decoupled stream ISAs and their required microarchitecture support [60].

A. Decoupled-Stream ISAs

Basic Concepts: Stream-config instructions (`stream_cfg`) encode patterns of memory access (e.g. `A[i]`), and can be chained to support indirection (e.g. `A[B[i]]`)[2]. Streams are configured before loops to define (and inform the microarchitecture) of their access patterns. Streams are deconstructed explicitly with a `stream_end` instruction; this enables data-dependent loop bounds.

Stream data is consumed semi-bindingly[3], through *pseudo-registers*, which are normal registers that are renamed to point to stream data (this allows any instruction to use stream data). Pseudo registers are set in the configuration instruction. Finally, `stream_step` instructions advance the position of the stream. Decoupling stream use from stream advancement allows control dependent use of stream data. Because a pseudo-register can be reused before advancement, the ISA defines that the first use defines the program order for that load.

Examples: Figure 6 shows two example programs in the decoupled stream ISA; for each we show the original code, a pseudo-assembly version with intrinsics, and the dependence graph representing the relationship between streams. The vector add loop in 6(a) requires configuring an induction variable stream, along with three memory streams that are dependent on it. The statement s_c=s_a+s_b corresponds to a typical add instruction in the ISA, but with pseudo-registers. Hence, this implies 2 loads and one store operation. Figure 6(b) shows an example with indirection, where one stream is configured to be dependent on another.

[2]The set of indirect patterns is described in §IV-B.

[3]*Semi*-binding because streams allow control-dependent use through decoupling access from advancement (using the `stream_step` instruction).

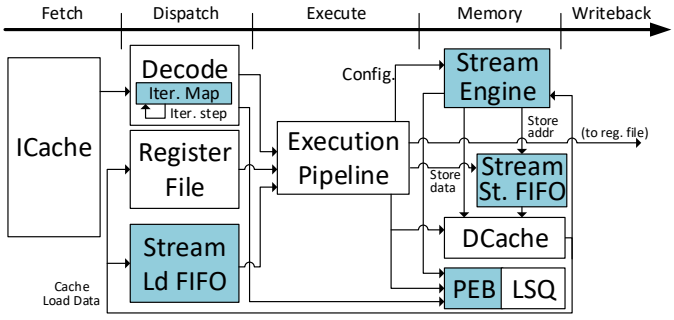


Fig. 7: Extensions for Supporting Decoupled Streams

B. Hardware Extensions for Decoupled Streams

Figure 7 overviews microarchitecture extensions to support a decoupled-stream ISA, as we describe below.

Stream Engine (SE): The stream engine (SE) holds streams' definition and manages their state. After being configured by `stream_cfg`, it allocates stream FIFOs and issues requests to L1. It also receives stream data, used to compute indirect addresses. Streams are deallocated by `stream_end`.

Stream FIFOs: There are two stream FIFOs, one each for loads and stores. The load FIFO buffers prefetched stream data and is accessed by core instructions that consume stream data. The store FIFO combines the store address (from the SE) and the store data (from core pipeline) and sends them to the store buffer when the core instruction commits. Stream FIFOs may be accessed at vector width by SIMD instructions.

Iteration Map: The iteration map in the decoder essentially performs renaming of stream registers, for the core to maintain consistency with streams. Specifically, it maps a stream register access to its current sequence index (its iteration). A `stream_cfg` initializes it to 0, and it is incremented with each `stream_step`. When decoding, consuming registers are renamed to stream FIFOs according to iteration count.

Memory Ordering: As discussed, the first-use of a stream element defines the program order for that load. In the microarchitecture, on first use, a stream load is dispatched to the load queue for alias detection.

Since the stream engine (SE) issues requests ahead of the core, a prefetch element buffer (PEB) is added to track the RAW dependence between stores and prefetched stream elements. It can be viewed as a logical extension of the LQ. When a store is committed to the store buffer, the PEB is searched for possible aliased stream elements. If stream aliasing is detected in either structure, all prefetched elements in PEB are flushed and their requests are reissued. The SE disables prefetching for the aliased stream to avoid future memory order violations. Elements are freed from the PEB when the first user is dispatched into the LQ or when released as unused.

Inorder Decoupled-stream Core: While not previously proposed, decoupled-stream ISAs are particularly attractive for inorder cores, as the required hardware support is modest, but the inorder core essentially gets the latency hiding capability of the OOO core for well defined patterns. Because it reorders

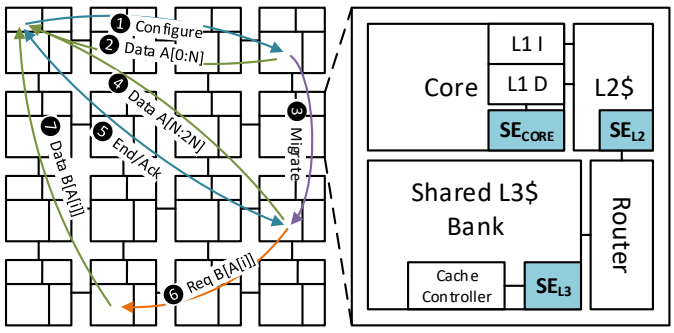


Fig. 8: Stream Floating Overview

memory instructions, it retains the PEB of the OOO core (and small LSQ) for memory disambiguation.

IV. Stream Floating Design

In this section, we develop the detailed microarchitecture and policies for transparently supporting stream floating.

Figure 8 overviews the stream floating system. In addition to the core stream engine (SE_{core}), we add a "stream engine" to the L2 and L3 cache levels (SE_{L2}, SE_{L3}) to manage stream interactions there. We refer to the tile consuming the stream data as the "requesting" tile, and the tile where the floating stream is offloaded to as the "remote" tile. In general, the remote SE_{L3} (Figure 10) generates requests and sends stream data back to the requesting SE_{L2}. The requesting SE_{L2} (Figure 9) buffers the stream data and matches it with requests from the SE_{core}. We first show a detailed example of an affine stream, and then generalize to indirect streams and stream confluence.

A. Affine Stream Floating

Stream Configure: SE_{core} decides whether a load stream should be floated to cache using its pattern and history information (details in §IV-D). If so, SE_{core} sends a stream configuration packet to SE_{L2}, containing the hardware context id (same as core id if no SMT), the stream id, and its pattern (i.e. base address, stride, etc.). This is ❶ in Figure 8.

SE_{L2} sets up the stream context and allocates the stream buffer. Then it computes and translates the address of the first stream element (see §IV-E). SE_{L2} sends a configuration message over the NoC to the remote L3 bank where the first element is mapped to. Upon receiving the packet, the configure unit in the SE_{L3} initializes the stream state, and the issue unit starts to generate requests based on the stream pattern.

Stream Request: Once configured, SE_{L3} computes addresses and sends requests to the colocated L3 cache controller. The issue unit selects ready streams in round-robin order. Besides address and type, requests also contain the stream id and the element index. The L3 cache controller is directed to send the data response to the original requesting tile (❷ in Figure 8). Thus, we generate requests at the remote tile on behalf of the requesting tile, and eliminate the unnecessary NoC traffic. The stream data will be buffered at SE_{L2}, not cached by the L2 cache. SE_{L2}'s buffer is address-tagged for memory disambiguation (see §IV-E).

Note that SE_{core} still generates requests to prefetch the stream data. These requests are also tagged with its stream id and the element index, and are intercepted by the SE_{L2} if matched to a floating stream. If so, the SE_{L2} checks its stream buffer, and either responds or delays if the data is not ready yet.

Most commonly, stream data is not present in the requesting private cache. However, in some scenarios (e.g. inadvertently floating a stream with high reuse) the data may already be cached in the L1 or L2. To avoid stalling the core, the L1 and L2 cache still perform normal tag checking for floating streams' requests, and respond immediately if hitting in the cache. The SE_{L2} will also be notified that the stream request is already served so that it can correctly advance the stream buffer.

	Field	Bits	Description	Field	Bits	Description
Affine	cid	6	Core id.	ptbl	48	Page table addr.
	sid	4	Stream id.	iter	48	Current iter.
	base	48	Base virt. addr.	size	8	Element size.
	strd	48	Mem-stride (3×)	len	48	Length (3×)
Ind.	sid	4	Stream id.	size	8	Element size.
	base	48	Base virt. addr.			

TABLE I: Affine and Indirect Stream Configuration

Stream Migrate: As the stream is iterating in the SE_{L3}, eventually the next element will no longer be mapped to the current L3 bank. At this point, the migrate unit (see Figure 10) constructs a stream migration packet similar to the stream configuration packet, but also with the current iter and remaining flow control credits (explained later in this section). This migration packet is sent to the L3 bank which holds the next element, and the stream continues there (❸,❹ in Figure 8).

Stream End: When a stream completes, the SE_{core} constructs a "stream end" packet to terminate the floating stream. The SE_{L2} uses the last allocated element's address to determine where to forward the packet, and the SE_{L3} will ack once done (❺ in Figure 8). Floating streams with known length can be silently terminated with no stream end packets. Notice that the stream end packet also enables the SE_{core} to terminate the stream early. This can be useful for implementing context switching, as well as reversing the decision to float a stream (i.e. sinking the stream) when there is L1/L2 locality (see §IV-D).

Coarse-Grained Flow Control: Since the stream data is buffered at the SE_{L2}, we need a flow control scheme to synchronize the SE_{L2} and SE_{L3} and avoid overwhelming the SE_{L2}'s buffer and network. We use a credit-based flow control scheme, where the SE_{L2} sends credits to the SE_{L3} indicating the vacancy of the stream buffer. These credit messages are handled by the flow unit in Figure 10, and the issue unit stalls the stream when running out of credits. This scheme is coarse-grained, as SE_{L2} only sends out credits when half of the allocated buffer is available; this helps amortize the overhead of flow control messages. SE_{L2} computes the last allocated element's addresses to determine which L3 bank it should send credits to.

Configuration Size: Table I summarizes the fields in a stream configuration packet. We assume 48-bit virtual address. Notice that we support up to a 3-level affine pattern to enable broad applicability and coarse grain patterns. The total size is 450 bits, which is less than one cache line.

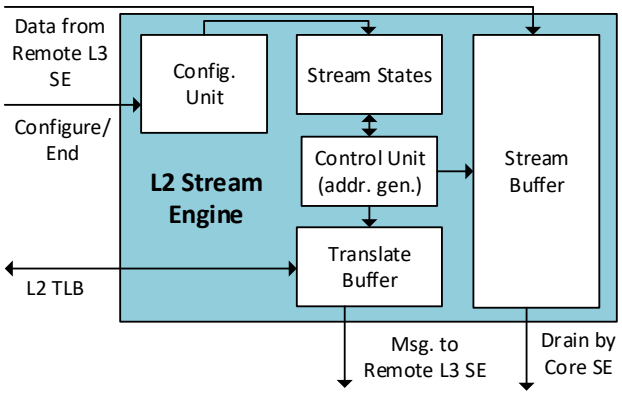


Fig. 9: L2 Stream Engine (SE_{L2})

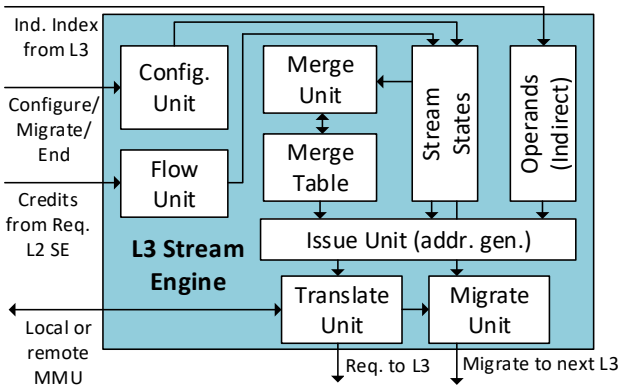


Fig. 10: L3 Stream Engine (SE_{L3})

B. Indirect Streams and Subline Transmission

Indirect streams are supported by combining their pattern with the corresponding affine stream – they are configured, migrated, and ended together, and share the same flow control credits. When a floated stream is indirect, the L3 cache controller notifies the colocated SE_{L3} when the indirect index is ready. This index is buffered in the operands table (Figure 10) and is used to compute the indirect access address. Finally, the indirect request is sent to the target L3 bank which responds to the requesting core with indirect data (❻,❼ in Figure 8).

Supported Patterns: The general indirect access pattern is:

$$\underbrace{i_0^{\text{len}_i} j_0^{\text{len}_j} k_0^{\text{len}_k}}_{\text{any order}} w_0^{\text{size}} B[A[i][j][k]+w] \quad (1)$$

The i,j, and k iterators can be reordered (by changing the strides in Table I), to support strided access. The w loop iterates over multiple consecutive data items from the indirect address. This enables the stream to support iterating over the fields of a structure (i.e. `A[i].x` and `A[k].y`) with one stream. It can also be used to iterate over a small linear range at each indirect location. Finally, by encoding further stream configuration within the indirect request, it is possible to support longer indirect chains like `C[B[A[i]]]`.

It is common in stencil workloads that two streams `A[i]`, `A[i+K]` have a constant offset (and thus reuse) [16]. If such reuse distance can fit in SE_{L2}'s buffer (after accommodating other streams), SE_{L3} would only send the first K elements of the first stream `A[i]`, and SE_{L2} would reuse data from the second stream `A[i+K]` for the following elements. This essentially provides the `A[i]` stream with a prefetch with distance of K elements. This approach is compatible with the aliasing detection scheme in IV-E.

Benefits: Floating indirect streams can 1. shorten the dependence chain for indirect accesses by generating the address at the remote L3 bank instead of returning to the requesting core; and 2. in most cases, indirect accesses have low inter-line locality, so we need only need transmit the required portion of the cache line, which can further save network traffic.

Configuration Size: Table I lists the fields of an indirect stream, which are appended to the base affine configuration and require 60 bits per indirect stream.

Field	Description	Field	Description
`sid`	Stream id	`request`	# stream requests
`reuse`	# priv. cache reuses	`miss`	# priv. cache misses
`aliased`	Aliased with stores		

TABLE II: Stream History Table

C. Stream Confluence

As an optimization, SE_{L3} transparently detects when multiple cores simultaneously request the same streaming data, and merges these requests. When adding a stream to the SE_{L3} (either configuring or migrating), the merge unit compares the new stream's parameters with those of existing streams (one comparison per cycle). Affine streams from different cores, but with same address space and parameters, form a confluence group, which is recorded in the merge table. The issue unit records any merged streams information in the request, and the response is multicast to their requesting tiles.

Although it is possible to merge streams from any two cores, it increases the hardware complexity and yields fewer benefits if they share no common path through the mesh NoC. Thus, we divide mesh tiles into smaller 2-by-2 blocks, and only merge streams from the same block. Each confluence group contains at most 4 streams, sorted by their progress (i.e. number of issued elements). The issue unit delays streams with more progress, so that those lagging behind can catch up and form a confluence request.

D. Policy for Floating and Sinking Streams

The SE_{core} decides whether to float a stream by considering both the current pattern as well as history information. If the stream's length is known and its estimated memory footprint is already larger than the private L2 cache, it can be directly floated. Otherwise, the SE_{core} will defer floating, and record its runtime behaviors in a stream history table, as in Table II. This includes the stream id, the number of requests sent and private cache misses. The private cache tag array is extended to remember the id of the stream which brought the line in. When a "stream" line is reused, the cache controller notifies the SE_{core} to increment the `reuse` field in the history table. Finally, the `aliased` bit is set to true if the core detects an aliasing store. After accumulating a certain number of stream requests, SE_{core} floats the stream if it exhibits no reuse, has a high miss ratio in the private cache and is not aliased.

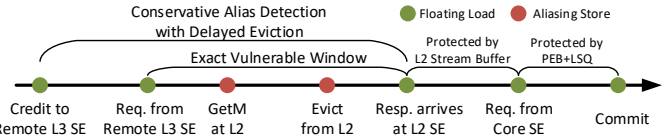


Fig. 11: Detecting Aliasing to Floating Stream Load

SE_{core} may "sink" a floating stream (undo the offload), by terminating it and starting to cache its data. It can be beneficial to sink a stream when the core detects an aliased store. Another case is when the floating stream starts to hit in the private cache. To handle this, SE_{core} sinks a stream if it hits in the private cache several times consecutively (we use 8 as the threshold).

E. Crosscutting Concerns

Address Translation: Since stream patterns generate virtual addresses, SE_{L2} and SE_{L3} addresses need to be translated. We assume each core has its own private two level TLB within each tile. Addresses generated by SE_{L2} are satisfied by the L2 TLB. TLB access is infrequent, as only the configure/end and coarse grain flow control messages are translated here.

As for SE_{L3}, we include a TLB in its translate unit in Figure 10, which again only needs to be queried for indirect access and at the beginning of a page for affine access. For SE_{L3} TLB misses, there are several options. One option is to send the translation request to the processor's IOMMU [7,8,28]. Another option is to use the requesting core's MMUs for translation, as was explored in prior work for accelerators [24]. This allows the reuse of the core's page table walker, MMU cache, and data cache for caching the page table entries. A third option is to use the remote core's MMU, but the potential downside is disturbing its MMU's caches if it is executing an unrelated workload. Therefore, if the thread running on the remote core is within the same address space as the requesting thread, SE_{L3} will access the remote core's MMU to avoid extra traffic, otherwise SE_{L3} will access the requesting core's MMU to avoid polluting the remote MMU.

Memory Disambiguation: Since floating streams load data before the core, we must detect aliasing. Figure 11 visualizes the life of a floating stream load. Starting backwards from commit, there are three windows that aliasing could happen:

After the SE_{core} issues the request, the floating load is protected by the PEB and LSQ, similar to other non-floating stream loads. Also, since a core stream request always checks the private cache's tag, it will get the updated value if the modified line is still present in the private cache. This mitigates the problem of detecting an aliasing store being *evicted* before the core stream request is serviced.

When the L2 cache evicts a dirty cache line, it searches the L2 stream buffer for a possible aliasing floating load. If found, it can either update the stream buffer with latest data, or simply mark the floating stream aliased and let the SE_{core} sink it. This search can be performed in parallel while the L2 cache is waiting for the ack from the L3, and is not on the critical path. This covers the second window.

Finally, there is a race condition when the store happens after the remote SE_{L3} issues the request and is written back before the response comes back. Since the floating load happens at the remote L3 tile in a decentralized fashion, we take a conservative approach to cover a slightly larger vulnerable window, starting from sending the credit to the remote SE_{L3}. Specifically, we maintain two sequence numbers (`head` and `tail`) for in-flight credits: newly sent credits remember and increment `head`, and incoming floating responses are reordered by their sequence number and increment `tail`. The L2 cache tags the line with `head` when it sees a dirty eviction from the L1 cache. Eviction of dirty cache lines will be delayed if its sequence number is greater than `tail`, as that means there are possible aliasing in-flight floating stream loads. This case is rare since the window is relatively short. However, there is a potential deadlock when the remote L3 bank happens to be waiting on the writeback. To break the dependence cycle, the SE_{L2} will notify the SE_{core} to sink a floating stream if it causes a long delay.

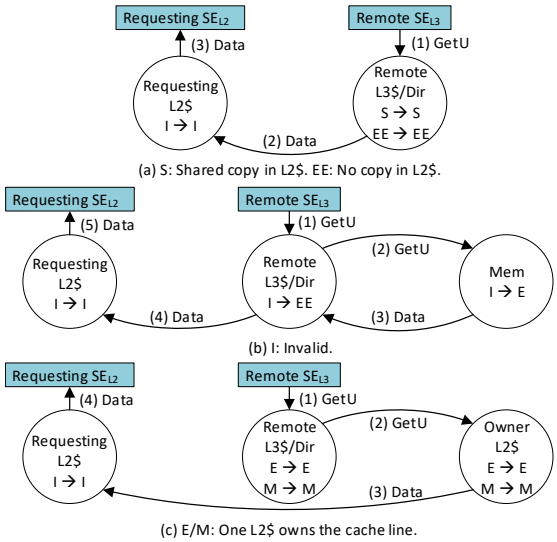


Fig. 12: Coherence Protocol Interaction

Precise State and Context Switch: Stream-floating adds no architectural state to the decoupled-stream ISA. On a context switch, SEs will discard/flush all floating streams. On switching back, all streams are initially not floating.

V. Coherence and Consistency

As discussed in §II, one of the major overheads for caching lines without reuse is that eviction causes traffic to the coherence directory (to update snoop filters to avoid unnecessary invalidations). Our goal is to avoid directory updates for data without reuse, so we can see the maximum potential of stream floating. We first outline the approach we take in our implementation, which does not allow for memory consistency of stream accesses (but allows for software to provide stream consistency). Then we outline an alternate that would, but which has other tradeoffs.

A. Our Approach: Uncached Stream Data

Our approach to avoiding clean-eviction traffic is to simply let the stream data reside in SE_{L2}'s buffer without being in a cached state from the perspective of coherence. The consequence is that we cannot support a traditional notion of consistency for streams, as another core can perform a store to stream data that is not detected by the directory. Note that this is rare in data-processing workloads, as writes

System Params	2.0GHz, 8x8 Cores
IO4 CPU (4-issue)	4-wide fetch/issue/commit 10 IQ, 4 LSQ, 10 SB
OOO4 CPU (4-issue)	24 IQ, 24 LQ, 24 SQ+SB 256 Int/FP RF,96 ROB
OOO8 CPU (8-issue)	64 IQ, 72 LQ, 56 SQ+SB 348 Int/FP RF, 224 ROB
Func. Units (×2 for OOO8)	4 Int ALU/SIMD (1 cycle) 2 Int Mult/Div (3/12 cycles) 2 FP ALU/SIMD (2 cycles) 2 FP Div (12 cycles)
L1 D/I TLB L2/SE_{L3} TLB	64-entry / 8-way 2k/1k-entry / 16-way, 8-cycle lat.
L1 I/D Cache	32KB / 8-way, 2-cycle lat.
Priv. L2 Cache	256KB / 16-way, 16-cycle lat
L1 Stride Pf. L1 Bingo Pf. L2 Stride Pf.	16 streams, 8 pf. per stream 8kB PHT, 2kB region 16 streams, 18 pf. per stream
NoC	256-bit 1-cycle link 5-stage router, multicast 8x8 Mesh, X-Y routing Memory controller at 4 corners
Shared L3 Cache	1MB per bank / 16-way 20-cycle lat., MESI coherence Static NUCA, 64B Interleave
Replacement Policy	Bimodal RRIP, $p = 0.03$
DRAM	1600MHz DDR3 12.8 GB/s
SE_{core}	IO4/OOO4/OOO8 256B/1kB/2kB FIFO, 12 streams
SE_{L2}	16kB FIFO, 12 streams
SE_{L3}	12 streams per core, 768 total

TABLE III: System and Microarchitecture Parameters

Benchmark	Dataset Parameters
conv3d	H/W: 256 × 256, I/O: 16×64, K: 3 × 3
mv	matrix 256 × 65536
b+ tree	1m leaves, 10k lookups, 6k range queries
bfs	1m nodes, 599970 edges
cfd	fvcorr.domn.193K
hotspot	1024 × 1024, 8 iters
hotspot3D	512 × 512 × 8, 8 iters
nn	768k entries
nw	2048 × 2048
particlefilter	48k particles, 1000 × 1000
pathfinder	1.5m entries, 8 iterations
srad	512 × 2048, 8 iterations

TABLE IV: Workload Datasets

to streaming data are otherwise synchronized. Streams *are* guaranteed to see stores prior to the creation of the stream, which is accomplished by waiting to offload the stream until the stream configuration instruction is committed. It is thus the compiler's responsibility to ensure that this guarantee is sufficient for correct execution. Our compiler's strategy is to limit stream lifetime to synchronization-free regions.

Uncached Coherence Extension: To support the uncached requests performed by SE_{L3}, we add a minor extension to a standard 3-level MESI protocol. Specifically we add a new request: GetUncached (`GetU`), which means the requested data will not be cached in the private cache. Figure 12 summarizes the transition and action for stable states involving `GetU`. These are for when (a) the data is present in L3 (e.g. S state), (b) the data is not present (e.g. I state), and (c) another L2 owns the data (e.g. M state). In all cases, the transitions follow a typical `GetS`, except that the requesting core is not added as a sharer. In (c), when another L2 owns the data, we let that core forward the data, again without altering its state.

B. Alternate: Stream-grain Coherence

In stream-grain coherence we let stream data be cacheable at the core (stream data still uses SE_{L2}'s stream buffer), and perform coherence at the *granularity of streams*. Instead of tracking the coherence state of stream data in the directory, we let the SE_{L3} track the coherence state on a per-stream basis, for example by keeping the accessed ranges of each stream with base/bound registers (false positives due to conservative range check will be rare). When another core accesses the directory, it also checks the SE_{L3} to see if it needs to invalidate the stream data (which would eventually cause the stream to re-execute and sink to SE_{core}). Also, the SE_{L3} will need to be informed when to deallocate a stream's range data. This would be performed when the core commits the `stream_end` instruction. The SE_{L2} would keep track of which SE_{L3}'s to deallocate for each stream. This idea is inspired by prior coarse grain coherence tracking works [40,41,47,50,65], but uses streams as the granularity.

The main advantage of this approach is, of course, that we could still use coherence events to implement consistency speculation for streams in the traditional way[4]. One disadvantage is that the range checks may have false positives (if a write is in between the reads of a stream) leading to unnecessary invalidations (though we suspect this is uncommon). A second disadvantage is the additional messages to deallocate streams in SE_{L3}, which can be an overhead for short streams which touch multiple banks (e.g. due to striding or indirect access).

Implementing stream-grain coherence is future work. However, we do not expect its disadvantages to be significant for the workloads we evaluate: streams are relatively long, and writes do not generally appear in the middle of stream ranges.

VI. Methodology

Simulator: We extended X86 gem5 [12,39] with partial AVX-512 support for higher vector width and simulate all cores in execution-driven, cycle-level detail. The in-order CPU is extended with a SE_{core} to support decoupled-stream ISA extensions. For the NoC, we use Garnet [3].

Compiler: We develop an LLVM-based compiler to generate stream-specialized programs with X86 backend, with similar support and mechanisms to prior work [60]. One difference is the that we use explicit load/store instructions (i.e. `stream_{load|store}`) to access stream data, instead of using pseudo-registers[5].

Benchmarks: We simulate 10 OpenMP benchmarks from Rodinia [14] and two tiled kernels: matrix-vector multiplication (`mv`) and 3d convolution (`conv3d`), as they are important workloads with stream behavior. Programs are compiled with `-O3` and AVX-512. Table IV summarizes the parameters.

Systems and Comparison: Table III summarizes the default system parameters including added hardware structures. We use McPAT [35] to estimate the energy at 22nm, and extended to model SE_{core}, SE_{L2} and SE_{L3}.

[4]Also, the need to prevent cache line evictions for alias detection (see Section IV-E) becomes unnecessary, as the SE_{L3} can inform the requesting core if it is attempting write ownership of stream data (indicates alias mispeculation).

[5]Pseudo-registers implicitly map certain registers in a region to stream data, and can eliminate some instruction overhead. Enabling pseudo-register support would further reduce the instruction overhead and shift more pressure to the cache and thus provide even more opportunities for stream floating.

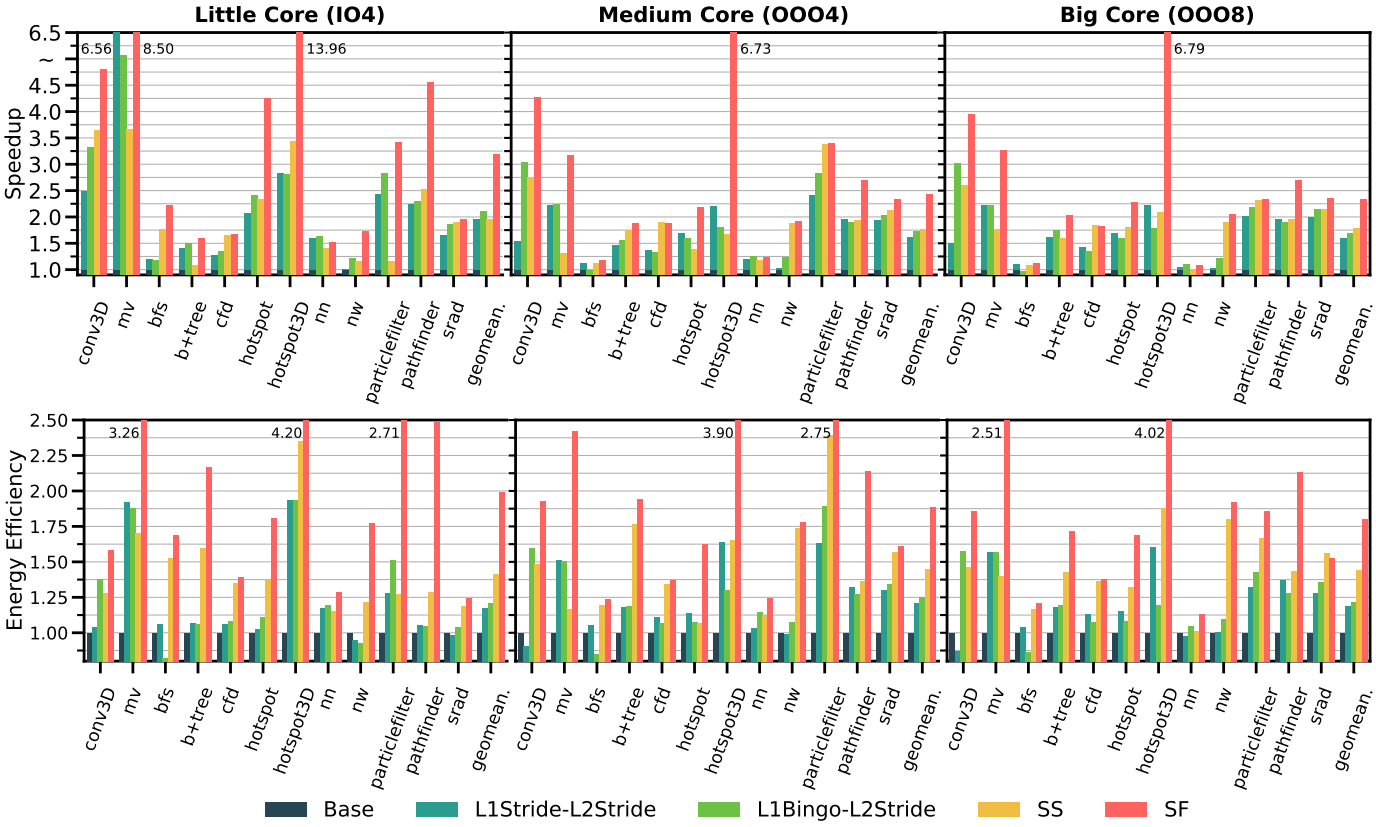


Fig. 13: Overall Speedup and Energy Efficiency

We choose two different prefetchers to compare against: traditional strided, because they capture the streaming behavior of these workloads, and the Bingo spatial prefetcher [9], because it won 1st place for multi-core prefetching in DPC3 [48] in 2019. By experimenting with different configurations, we found that adding an L2 prefetcher to both Bingo and the L1-stride prefetcher also improved performance.

We also implemented a "micro-architecture-only" version of the concept of coarse-grain requests to L3: **bulk prefetch**. Specifically, we augmented the L2 stride prefetcher to group consecutive prefetch requests as a single message if they are to the same L3 bank. We group 4 requests, as this reduced NoC traffic and avoided overfetch. This optimization can only be applied when the L3 address interleaving granularity is greater than 64B (one cache line). Since this helped performance less than just using 64B interleaving, it is only shown in the traffic analysis (Figure 15).

Specifically, we compare a **Base** core with no prefetching to:

Stride Prefetching (L1Stride-L2Stride): Baseline core with L1 and L2 stride prefetcher. Single-cycle request gen.; 16 streams and 8 (16 for L2) prefetching requests per stream.

Bingo Prefetching (L1Bingo-L2Stride): Baseline core with L1 Bingo spatial prefetcher [9], and L2 stride prefetcher.

Stream Specialized Processor (SS): Stream-specialized core as described in §III. It gets the benefits of stream-based prefetching, but not floating.

Stream Floating Processor (SF): Stream floating as described in this paper. Unless mentioned otherwise, SF uses 1kB L3 interleaving to reduce stream migration.

VII. Evaluation

Our evaluation attempts to address two main questions: First is how much potential exists in exploiting streaming patterns in reducing network traffic and coherence overheads, and second is whether that potential is only attainable when streams are embedded in the ISA. We begin by evaluating the overall performance and energy efficiency, then analyze how stream floating reduces network traffic, as well as its sensitivity to network bandwidth, NUCA mapping scheme and system size.

A. Overall Speedup, Energy Efficiency and Area

Performance: Figure 13 shows the speedup and energy efficiency over different baseline cores. For small cores (IO4), SS-IO4 (1.95×) is slightly worse than BG-IO4 (2.10×) due to limited FIFO size (256B). SF-IO4 further improves the speedup to 3.20×. The performance benefits can be attributed to network and coherence benefits, because the prefetchers generally recognize and optimize for the same patterns. The two exceptions are `bfs`, as our evaluated prefetchers do not support indirection, and `nw`, which failed on the stride prefetcher (blocked 2D array accessed in diagonal order). SF-IO yields 64% more performance than SS-IO, as it floats streams to the cache to reduce network traffic and latency. For OOO cores, the speedup of SF over SS is still significant, at 37% (OOO4) to 31% (OOO8), even though the wide OOO can hide much more memory latency.

Energy: For OOO8, the stride prefetcher and Bingo improve the energy efficiency by 19% and 21% respectively. Prefetching may hurt energy efficiency due to inaccuracy (`bfs` and `nw`).

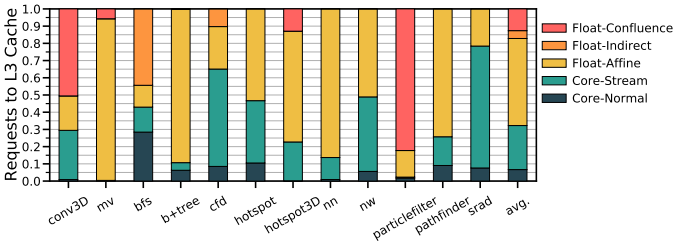


Fig. 14: Requests to L3 of SF-OOO8

SS-OOO8 achieves 1.44× energy efficiency, and SF-OOO8 pushes it to 1.80× with 25% improvement over SS-OOO8.

Area: Most of the area comes from the SRAM to store stream configuration and data, and we estimate the area using CACTI and McPAT (22nm). Each SE_{L3} can hold $12 \times 64 = 768$ streams and uses 48kB (0.11mm^2) to store stream configuration, as well as a 1k entry TLB (0.04mm^2). These sum to 4.5% overhead for the L3. At L2, the stream buffer takes 0.09mm^2 and the configuration state takes 0.05mm^2. The 35-bit L2 tag is extended with 4-bit stream id and 12-bit sequence number (§IV-E). Summing together, stream floating introduces 9% area overhead for L2 (0.16mm^2 / 1.85mm^2). The whole chip area overhead is 1.6% for IO4 and 1.4% for OOO8 (OOO8 also uses larger stream FIFOs in SE_{core}).

B. Floating Requests

Figure 14 breaks down requests to the L3 cache into normal/stream requests from the core and requests from floating affine/indirect/confluence streams. On average, 68% of the requests are generated by SE_{L3}, showing that a significant portion of memory accesses can be floated. Most of the requests are from affine floating (50%), and only 5% are from indirect floating (`bfs` and `cfd`). For stream confluence, SE_{L3} can successfully recognize multicast, e.g. the input feature map in `conv3D` constituting 51% of requests.

C. Network Traffic

Figure 15 shows the total number of traveled hops of all injected flits, normalized to Base, as well as average network utilization. The traffic in the first graph is classified by the packet type (from bottom to top): coherence control, data, and extra messages to manage floating streams (config., migration, termination, flow control). Besides SS and SF, we include SF-Aff with only affine floating enabled, and SF-Ind which adds indirect floating. We also add the *bulk* optimization for the prefetchers, as described in §VI, which groups 4 contiguous prefetch requests.

Streams are accurate: L1Bingo-L2Stride actually increases the NoC traffic by 34% due to inaccurate pattern and aggressive prefetching. This can be mitigated by dynamically trading-off prefetching aggressiveness with accuracy and timeliness. However, the decoupled-stream ISA extension provides *accurate* stream information and SS does not increase traffic (except 3% for `cfd` and `hotspot3D` due to interference between SE_{core} and normal core requests).

SF fundamentally reduces traffic: The bulk prefetching optimization reduces traffic by 6%, but it is still limited by the inaccurate pattern and unnecessarily caches the data with no reuse. On the other hand, offloading affine streams reduces

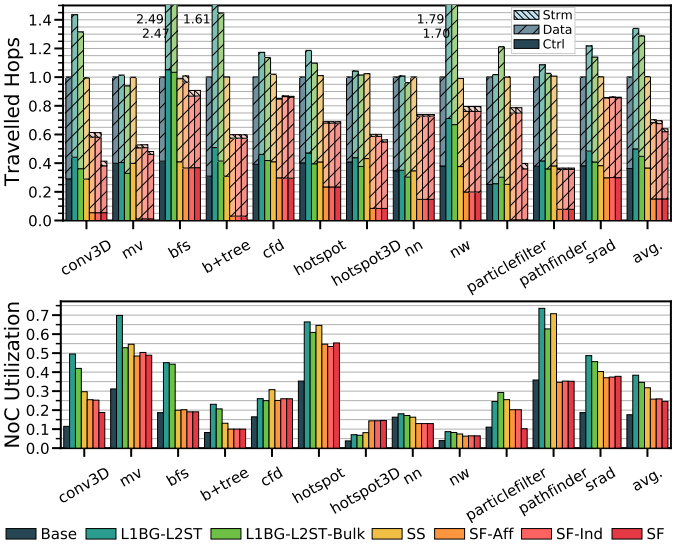


Fig. 15: OOO8 NoC Traffic and Utilization

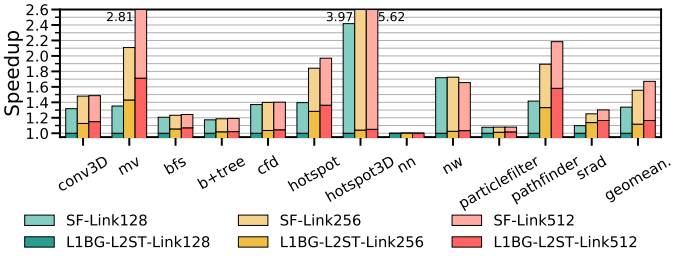


Fig. 16: SF vs. Bingo with 128, 256, 512-bit link (OOO8)

the traffic by 30%. Most of the reduction comes from control messages, as SF eliminates redundant requests and simplifies the coherence protocol for offloaded streams. More importantly, data traffic is also sometimes reduced (e.g. `pathfinder`), as not caching stream data without reuse prevents pollution. Finally, only 2% traffic overhead is needed to configure and migrate streams, as they capture long-term behavior.

Among these benchmarks, only `bfs` and `cfd` contain indirect streams, and indirect offloading helps in the case of `bfs` due to subline transfer. For `cfd`, the traffic slightly increased by 2%, as a small fraction of indirect stream data is already cached. Finally, stream confluence shows significant benefits on `conv3D` (sharing the same input feature map) and `particlefilter` (resample through the same accumulated weight array). Overall, SF reduces network traffic by 36% and average network utilization from 35% (Bingo) to 25%.

D. Sensitivity to NoC Bandwidth

Figure 16 shows the performance of SF and Bingo under different link widths, normalized to Bingo with 128-bit links. For some benchmarks, increasing link width does not cause speedups because: 1. for computation intensive workloads (e.g. `particlefilter`) a 128-bit link can already transfer data fast enough; 2. when streaming from main memory (e.g. `nn`), memory bandwidth/latency becomes the bottleneck.

Compared to Bingo with the same link width, SF performs better as link width increased from 128-bit (1.34×) to 512-bit (1.43×). This is because with 512-bit link, data messages are

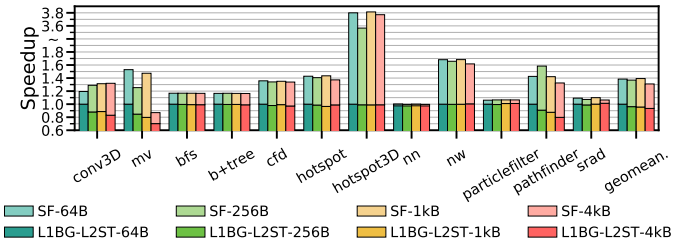


Fig. 17: Effect of NUCA Interleaving Granularity (OOO8)

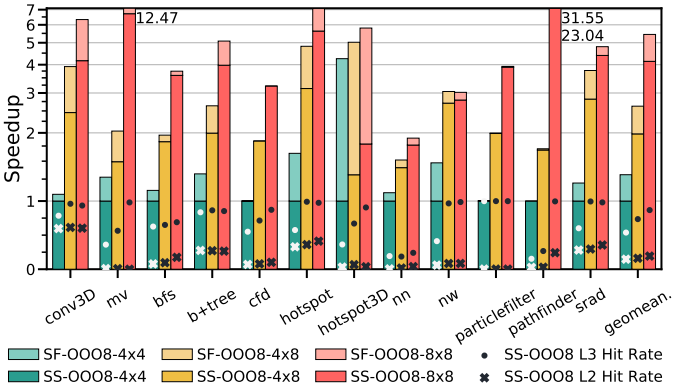


Fig. 18: Core Scaling

broken into fewer flits and can be transmitted faster, meanwhile the latency of control messages becomes proportionally more critical. SF benefits more from higher link width, as it eliminates unnecessary control messages.

E. Sensitivity to NUCA Mapping

Addresses are interleaved in L3 banks to avoid hotspots. We evaluate how interleaving granularity affects performance for simple linear static NUCA, as finer-granularity implies more migrations. Figure 17 shows the performance of Bingo and SF with 64B, 256B, 1kB and 4kB interleaving granularity, normalized to Bingo-64B. Some benchmarks do suffer from hotspots with coarse interleaving granularity (e.g. `mv`) and Bingo-4kB performs slightly worse than Bingo-64B (0.93×). SF performs the best with 1kB interleaving, as network traffic caused by stream control messages is still negligible (1.5% for SF-1kB vs. 1.1% for SF-IO-4kB), while also avoiding hotspots in L3 banks. For 64B interleaving, streams constantly migrate, generating 12% stream control traffic compared to Base, but still reduce the total network traffic by 22%. Also, floated streams run ahead of the core, and migration latency is hidden.

Although we consider static NUCA, dynamic NUCA may also have interesting opportunities. E.g. aggressively migrating cache lines closer to the requesting tile based on the stream pattern, or the center of multiple requesting tiles if they are offloading the same stream (i.e. during stream confluence).

F. Sensitivity to Core Scaling

Figure 18 shows SF's speedup over SS with varying core counts, normalized to SS-4x4. Dots indicate L2 and L3 hit rate in SS. For some benchmarks (e.g. `pathfinder`), stream floating has better scaling than SS, especially when the working set can be cached in L3 and the L2 hit rate is low, as the NoC bandwidth pressure is reduced and L2 cache capacity is saved

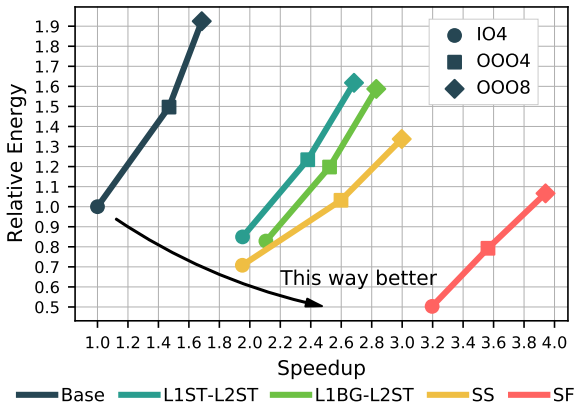


Fig. 19: Energy vs. Speedup for IO4, OOO4, OOO8

for reused data. However, when the data cannot be cached on chip, SF suffers from the same memory bottleneck as SS and yields marginal speedup, e.g. `mv-4x8`. Overall, SF achieves slightly better speedup for 8x8 (1.32×) than 4x4 (1.30×).

Figure 19 shows the energy vs. speedup across IO4, OOO4, and OOO8. For these workloads, SS slightly outperforms state-of-the-art prefetchers for OOO cores. After enabling streams to float into the cache hierarchy, significantly new tradeoffs emerge: SF-IO4 even outperforms SS-OOO8, and has much lower energy consumption.

VIII. Related Work

Decentralizing Compute: A large body of work explored the idea of bringing near-data computing to general purpose systems. One example is PIM-enabled instructions [4], which enables offloading remote memory instructions. Active Routing [27] maps computation kernels to the memory network in a hybrid memory cube. While sending packets for operands, the network builds a routing tree for the computation, and computes the result at the least common ancestor of the operands. SnackNoC [51] leverages the idle time of NoC router to perform computation. A centralized packet manager distributes computation to routers, which chains computation by forwarding. Livia [38] enables user-defined, single-input computation kernels to be offloaded to the highest level of the memory hierarchy where the data exists, including memory.

Stream floating focuses on long-term data-movement (inspired by [18,44,60–62]) and only computation offloading of address generation, simplifying hardware requirements. Furthermore, all of the above require APIs/programmer support, whereas stream floating leverages an ISA targetable by a simple compiler.

Near-data for GPUs: EMC [25] augments a GPU memory controller with compute, and enables miss-generating data-dependent instructions to be executed at the memory controller. Pattnaik et al. explore offloading RMW instruction chains to remote cores, as well as computing at the "meet" of two remote inputs, to reduce traffic [45].

Programmer Control of Locality: Jigsaw exposes cache data placement and allocation to software for locality and avoiding cache interference [10,11]. Jenga [57] can configure the hierarchy, avoiding unnecessary levels. Coherence Domain Restriction (CDR) [19] adds a mapping between logical and

physical cores to limit coherence needs to fewer cores, reducing traffic and latency. These require some programmer/system support, and (save for Jenga) will not apply to sharing across all cores.

Cache Policy Optimizations: Dead-block techniques predict which cache lines have little reuse, and help quickly replace or avoid caching them [26,32–34,37]. Other bypassing techniques follow similar principles [13,22,53,56,63]). It is future work to selectively decide whether to bypass or allow floating stream data to enter cache. Similarly, several cache replacement policies avoid thrashing by initially assuming little reuse [29,30,49].

Another body of work recognizes data sharing patterns to simplify coherence operations (e.g. producer-consumer [15,31], migratory [17,23], false-sharing [58]), and also to enable forwarding to hide latency [1,42,59]. The pattern can be detected by hardware, or supplied by software.

None of the above support accesses whose requests originate remotely, which is required for stream floating optimizations.

Prefetching: Stream-floating is heavily inspired by many prior prefetching works. For example feedback directed prefetching [55] monitors prefetching usefulness, lateness and pollution to throttle the prefetcher; our design also monitors usefulness for floating and throttles based on timing.

Specifically for indirect (data-dependent) prefetching, prior work explored software techniques [5] and hardware techniques like IMP [64] and CATCH [43]. Our approach identifies similar patterns. An even more general approach is the event-triggered prefetcher [6], which allows specialized prefetching hardware to run simple programs which can respond to prefetch events. Our prefetcher is less general, focusing on common forms.

Buffets [46] are an efficient composable storage idiom for accelerators that enables efficient data-reuse without the overheads of caching or inflexibility of double-buffering with scratchpads. They do not integrate with general caches.

In-Memory/In-Cache Computing: Another line of work attempts to perform computation using the same substrate as memory devices [20,36]. Compute Cache [2] and Duality Cache [21], enable SRAMs to serve as both caches and bit-serial computation units. These suggest a path forward for enabling stream floating to also offload computation.

IX. Conclusion

This work explores the idea of leveraging inherent program access patterns – streams – as the units of near-data offloading. We find that streams are prevalent in data processing workloads, and encode useful information that can help eliminate coherence and traffic overheads. By exposing stream information to the caches, they can proactively prefetch with optimized cache policies and mechanisms. Our microarchitecture can correctly identify beneficial streams and transparently float them among the caches, reducing the network traffic as well as improving the cache utilization.

More broadly, as we continue to scale multicores, especially without the help of technology improvements, it is important to consider new avenues for innovation beyond microarchitecture-only solutions. We believe that the concept of exposing higher level abstractions like streams can help to enable new memory system optimizations.

X. Acknowledgments

We sincerely thank the anonymous reviewers for their helpful suggestions and encouragement, as well as our friends and families for their support during this pandemic. This work was supported by funding from Intel's FoMR program, as well as NSF grants CCF-1751400 and CCF-1823562.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve, "An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors," in *Proceedings Third International Symposium on High-Performance Computer Architecture*, 1997, pp. 204–215.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 481–492.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 33–42, 2009.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[5] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 305–317.

[6] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 578–592, 2018.

[7] AMD, "Amd i/o virtualization technology (iommu) specification," December 2016. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf

[8] ARM, "Arm system memory management unit architecture specification smmu architecture version 3.0 and version 3.1," 2017.

[9] 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)*, Feb 2019, pp. 399–411.

[10] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 213–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523752

[11] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 538–550.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[13] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 293–304.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. USA: IEEE Computer Society, 2009, p. 4454. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[15] L. Cheng, J. B. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 328–339.

[16] Y. Chi, J. Cong, P. Wei, and P. Zhou, "Soda: stencil with optimized dataflow architecture," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[17] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, p. 98108, May 1993. [Online]. Available: https://doi.org/10.1145/173682.165146

[18] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 924939. [Online]. Available: https://doi.org/10.1145/3352460.3358276

[19] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 686–698.

[20] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 114. [Online]. Available: https://doi.org/10.1145/3173162.3173171

[21] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 397410. [Online]. Available: https://doi.org/10.1145/3307650.3322257

[22] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 81–92. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000075

[23] H. Grahn and P. Stenstrm, "Evaluation of a competitive-update cache coherence protocol with migratory data detection," *Journal of Parallel and Distributed Computing*, vol. 39, no. 2, pp. 168 – 180, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731596901641

[24] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 37–48.

[25] M. Hashemi, E. Ebrahimi, O. Mutlu, Y. N. Patt *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 444–455.

[26] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior," in *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 209–220.

[27] J. Huang, R. R. Puli, P. Majumder, S. Kim, R. Boyapati, K. H. Yum, and E. J. Kim, "Active-routing: Compute on the way for near-data processing," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 674–686.

[28] Intel, "Intel virtualization technology for directed i/o, architecture specification," June 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf

[29] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 208–219.

[30] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815971

[31] A. Kayi and T. El-Ghazawi, "An adaptive cache coherence protocol for chip multiprocessors," in *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, ser. IFMT '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1882453.1882458

[32] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 175–186.

[33] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, April 2008.

[34] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 144–154.

[35] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO '09*.

[36] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 288–301.

[37] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, Nov 2008, pp. 222–233.

[38] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 20. New York, NY, USA: Association for Computing Machinery, 2020, p. 417433. [Online]. Available: https://doi.org/10.1145/3373376.3378497

[39] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mck, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and der F. Zulian, "The gem5 simulator: Version 20.0+," 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[40] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, "Jetty: filtering snoops for reduced energy consumption in smp servers," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 85–96.

[41] A. Moshovos, "Regionscout: Exploiting coarse grain sharing in snoop-based coherence," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 234–245, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1080695.1069990

[42] M. Musleh and V. S. Pai, "Automatic sharing classification and timely push for cache-coherent systems," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[43] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 96–109.

[44] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 416–429. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080255

[45] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic computing in gpu architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 210–223.

[46] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 137–151. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304025

[47] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated cpu-gpu systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013.

[48] S. Pugsley, "3rd data prefetching championship," June 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/slides/dpc3_closing.pdf

[49] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[50] V. Salapura, M. Blumrich, and A. Gara, "Improving the accuracy of snoop filtering using stream registers," in *Proceedings of the 2007 Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*, ser. MEDEA '07. New York, NY, USA: ACM, 2007, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/1327171.1327174

[51] K. Sangaiah, M. Lui, R. Kuttappa, B. Taskin, and M. Hempstead, "Snacknoc: Processing in the communication layer."

[52] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," *arXiv preprint arXiv:1911.08356*, 2019.

[53] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "The direct-to-data (d2d) cache: Navigating the cache hierarchy with a single lookup," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 133–144. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665694

[54] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[55] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 63–74.

[56] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive gpu cache bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-8. New York, NY, USA: ACM, 2015, pp. 25–35. [Online]. Available: http://doi.acm.org/10.1145/2716282.2716283

[57] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 652–665. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080214

[58] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, "Deft: Design space exploration for on-the-fly detection of coherence misses," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1970386.1970389

[59] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, "Deft: Design space exploration for on-the-fly detection of coherence misses," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1970386.1970389

[60] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 736–749. [Online]. Available: https://doi.org/10.1145/3307650.3322229

[61] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

[62] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 703–716.

[63] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 76–88.

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

[65] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 314–327. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.5