Near-Stream Computing: General and Transparent Near-Cache Acceleration

Zhengrong Wang, Jian Weng, Sihao Liu, Tony Nowatzki
University of California, Los Angeles
{*seanzw,jian.weng,sihao,tjn*}*@cs.ucla.edu*

***Abstract*—Data movement and communication have become the primary bottlenecks in large multicore systems. The near-data computing paradigm provides a solution: move computation to where the data resides on-chip. Two challenges keep near-data computing from the mainstream: lack of programmer transparency and applicability. Programmer transparency requires providing sequential memory semantics with distributed computation, which requires burdensome coordination. Broad applicability requires support for combinations of address patterns (e.g. affine, indirect, multi-operand) and computation types (loads, stores, reductions, atomics).**

We find that streams – coarse grain memory access patterns – are a powerful ISA abstraction for near data offloading. Tracking data access at stream-granularity heavily reduces the burden of coordination for providing sequential semantics. Decomposing the problem using streams means that arbitrary combinations of address and computation patterns can be combined for broad generality.

With this insight, we develop a paradigm called near-stream computing, comprising a compiler, CPU ISA extension, and a microarchitecture that facilitate programmer transparent computation offloading to shared caches. We evaluate our system on OpenMP kernels that stress broad addressing and compute behavior, and find that 46% of dynamic instructions can be offloaded to remote banks, reducing the network traffic by 76%. Overall it achieves 2.13× speedup over a state-of-the-art near-data computing technique, with a 1.90× energy efficiency gain.

***Keywords*-Stream-Based ISAs; Programmer-Transparent Acceleration; Near-Data Computing;**

I. Introduction

As systems scale, the overheads of data movement and communication become the primary bottlenecks for high performance energy-efficient execution, especially for data-processing workloads that rely on large datasets. A variety of specialized architectures mitigate these overheads by carefully scheduling computation near data and orchestrating data-movement in efficient pipelines. This broad paradigm of near-data processing (NDP) includes near-memory techniques [10,19,21,22,25,26,29,31,49,50,66,71,73], as well as near-cache [1,2,20,23,36,44,57]; the latter is our focus.

Bringing NDP to general purpose computing is challenging because of three competing goals: *transparency* to the programmer, *generality* of computations offloadable, and *autonomy* of offloaded computations to keep overheads low.

Transparency can be provided trivially by performing near-data computation at thread-level [30,56,66]. However, efficient thread-level NDP is limited to workloads that only process a single data-structure, as different data-structures generally have different access patterns and would benefit from a tailored offloading approach. Generality can be provided by instead using finer-grain, sub-thread abstractions for offloading decisions, like offloading special instruction sequences [5,49,57] or short program regions [32,44,61].

While attractive, sub-thread offloading poses significant challenges to all three goals. In terms of *generality*, NDP techniques should support offloading near many types of data structure (arrays, lists, trees, etc.) and flexibly employ various computation strategies (near-data filtering, stores, atomics, reductions, etc.). However, prior works only support a subset [5,32,44,49,57,61].

To provide transparency, sub-thread offloading requires maintaining sequential memory semantics with distributed address generation and computation. To this end, prior instruction-based offloading techniques integrate remote memory access with the coherence protocol [5,57]; however, because offloaded computations are instruction-granularity – and thus not *autonomous* – they require expensive fine-grain coordination. Another approach is to rely on the programmer to provide guarantees on access patterns [32,44,61], but the corresponding APIs generally require expert knowledge.

In this work, our goal is to provide effective and general near-cache computing capability for general purpose cores without programmer help. Our primary insight is that using *streams* – i.e. coarse grain memory access patterns – as the granularity and abstraction for offloading helps to enable general, autonomous, and transparent offloading:

- *Generality:* Streams capture long-term per-data-structure behavior, so optimizations can be more aggressive than with instruction-level offloading. (e.g. stream abstractions enable a reduction operation to be fully offloaded, so that only the final value needs to be returned).
- *Autonomy:* Streams enable efficient autonomous offloading by eliminating coordination overhead (e.g. copying array a[] to b[] only requires two requests from the core, rather than one request per cache line).
- *Transparency:* Streams reduce the overhead of maintaining sequential memory semantics by enabling detection of memory ordering violations using per-data-structure access summaries rather than individual accesses.

Approach: Based on these insights, we develop a paradigm that we call *near-stream computing*. Here, programs express each coarse grain memory access pattern with *stream* configuration instructions. When appropriate, instructions are grouped with streams by the compiler, and they execute together either in the core or in cache. Offloaded computation

is specified with control/memory-free instruction blocks of the original ISA, and is either executed with a small scalar unit within the cache or on the remote core using a lightweight thread context.

To maintain sequential program semantics and preserve coherence, we developed a synchronization protocol that communicates the conservative address range over computation intervals. The coherence protocol and memory-dependence disambiguation hardware are updated to leverage stream-granularity information for conservative operation.

The stream-pattern determines the strategy for offloading and aspects of the synchronization protocol. In our taxonomy, a stream-pattern consists of an address pattern (affine, indirect, pointer-chasing, or multi-operand) and a computation type (filter-load, remote store, read-modify-write, or reduction).

Finally, there are a class of optimizations which are difficult to apply in a fully-programmer transparent way, specifically where the original memory ordering may not be preserved. Our work explores novel compiler transformations which can provide further opportunity with limited programmer intervention (simple pragmas), and we evaluate the potential of full versus near-full programmer transparency.

Overall, our contributions are:

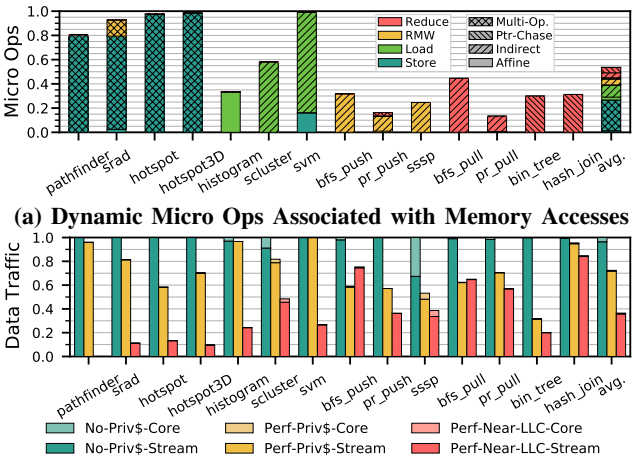
- Exploration of a novel program abstraction and granularity – streams – for performing near-data computing.
- Range-based synchronization and memory disambiguation protocol for maintaining sequential semantics with distributed computation at low overhead.
- Novel compiler techniques that perform aggressive near-data optimizations with simple pragmas.
- Evaluation of near-stream computing against multiple prior near-data approaches [44,57,68].

Implementation and Results: Our implementation[1] includes a CPU ISA extension (x86), a set of LLVM-based compiler transforms and backend, and a microarchitecture. To study the potential of near-stream computing, we choose data-processing kernels from GAP graph suite and Rodinia, representing irregular and regular workloads respectively.

Our evaluation first found that significant traffic reduction was possible; on average 76%, and up to 98%. Performance gains were even higher due to reduced latency, including an average of 2.13×, 2.48× over designs inspired by Livia [44] and Omni-Compute [57] respectively.

Paper Organization: We first make a case for near-stream computing in §II, and also overview our optimizations and compare against prior near-data works. Then we discuss the ISA and the basic in-core operation in §III. We describe our offloading approach and range-based synchronization in §IV, while §V discusses programmer-exposed synchronization-free optimizations. We cover methodology and evaluation in §VI and §VII, followed by additional related work in §VIII.

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



(a) Dynamic Micro Ops Associated with Memory Accesses

(b) Data Traffic Reduction after Offloading Computation

Figure 1: Potential of Sub-Thread Near-Data (View in Color)

II. Motivation and Overview

We first discuss a taxonomy of near-data patterns and the opportunity they provide. Then, we overview our approach, and compare to other sub-thread near-data techniques.

A. Taxonomy and Opportunity

Taxonomy: For sub-thread near-data, there are two dimensions of generality: *address patterns* and *computation types*.

Address patterns intuitively include affine (e.g. `A[i,j]`), indirect (e.g. `B[A[i,j]+w]`), and pointer-chasing (e.g. `P=P.next`). The latter two are data-dependent and non-contiguous, so imply distributed access. We also include multi-operand access patterns, for when a computation operates on multiple data sources (e.g. A[i] op B[i]). This requires further coordination of distributed access, as A[i] and B[i] may not be mapped to the same location.

When such an address pattern is decoupled from the remainder of the program, we refer to it as a **stream**.

Compute Patterns define the relationship between near-memory and in-core work. Four common patterns are:

- **Near-Load-Stream:** Computation may be performed near a dependent load stream to reduce the data traffic, either by reducing bitwidth or filtering data.
- **Near-Store-Stream:** Computations may be performed near a store stream to avoid returning outputs to the core.
- **RMW:** Read-modify-write ops update each data item in place, and greatly reduce the latency and network traffic.
- **Reduction:** Like *near-load-stream* but with accumulation. No intermediate data is communicated to the core.

Near-Stream Opportunity: To understand the potential of near-stream computing, we study how prevalent different compute and address types are across data-parallel workloads (see §VI/§III-B for workload/compiler details). Figure 1(a) shows the breakdown of dynamic micro ops committed that can be associated with streams, where 21% are associated with load-streams (including reduction) and 31% with store and RMW.

Next, we demonstrate that the ideal near-data scheme heavily reduces data traffic with respect to even ideal private caches. Figure 1(b) shows the pure data traffic (bytes $\times$ NoC hops) of three abstract systems. *No-Priv$*: baseline system with no private caches, *Perf-Priv$*: system with perfect private cache (fully-associative, byte-granularity, LRU, 256kB, zero-cost update-based protocol), and *Perf-Near-LLC* where computation is offloaded to LLC banks. All systems have 64-cores, a mesh NoC, and 1MB/bank LLC. We find that adding private caches only reduces 27% of data traffic, due to the large reuse distance. However, near-LLC computing reduces the data traffic by 64%.

B. Optimization Overview

The basic principle of stream-based near-data computing is that a decoupled stream may be offloaded near an LLC bank, along with some computation. The coordination and flow of data varies depending on the aspects of the taxonomy. We begin assuming a simple affine access pattern, and discuss how different compute types would work. Then we generalize to more complex access patterns.

Reduce: Figure 2(a) shows the case of affine reduction (Σ`A[i]`). Conventional systems fetch all the data to the core to accumulate the result (multiple request/response arrows), introducing unnecessary traffic. By coupling the reduction with the stream `A[i]`, the remote LLC can perform the computation in place. As the stream iterates, it automatically migrates to the next LLC bank with the new data and keeps reducing. Nearly all data traffic is eliminated.

Store: Store streams, like memset (i.e. `A[i]=0`), introduce significant overhead, as they require getting write permission and writing back. With near-stream computing, this can be performed in place as the stream migrates.

Load: It may also be beneficial to couple computation with a load stream and respond with the computation result, especially when the computation reduces the data type size. For example, extracting a hash key of a few bits from a larger value. Also, if a computation's result is used by multiple stores, associating it with the load stream instead would avoid redundant compute.

RMW: RMW streams (e.g. `A[i]+=C`) are a hybrid case of both load and store computation. Semantically, they guarantee atomicity of the update.

Access Pattern: Multi-op: Operating on multiple data streams complicates near-data computing, as it requires coordination within the memory system. Figure 2(b) shows the case of vector addition, i.e. `C[i]=A[i]+B[i]`. Our approach is to allow the load streams to compute the location of the store stream, so their data can be forwarded there directly. Here, the computation is performed at the store stream, and is updated in place with minimal data traffic and no writeback traffic at all.

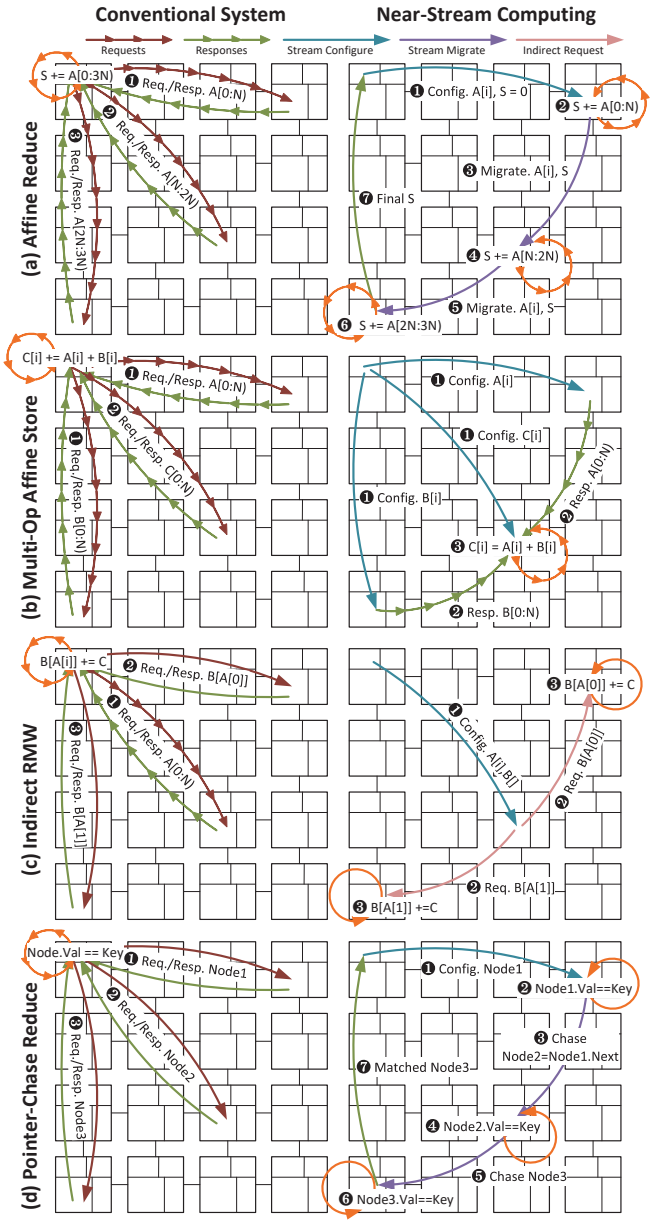


Figure 2: Near-Stream Computing Optimizations

Access Pattern: Indirection: Computations can also be associated with indirect streams. Figure 2(c) shows an indirect RMW on `B[A[i]]`. Instead of fetching `A[i]`, computing the indirect address, and finally bringing in and updating `B[A[i]]`, we can associate the atomic operation with the indirect stream `B[A[i]]`, and generate indirect atomic requests in remote cache banks. This not only reduces the data traffic, but also shortens the long dependence chain and lowers the latency.

Access Pattern: Pointer-Chasing: Figure 2(d) shows an example of searching in a linked list. This example uses a reduction, and chases the pointer among LLC banks. Similar to indirect patterns, this removes the core from the long dependence chain and only the final matched result is sent

	Active Rtng [32]	Livia [44]	Omni-Comp. [57]	Snack -NoC [61]	PIM-En. [5]	Near -Stream
Data Level	HMC	LLC/MC	LLC	LLC	Mem	LLC
Prog. Transparent	No	No	Yes	No	No	Yes
Loop Autonomous	Yes	Yes	No	No	No	Yes
# Patterns (Tab II)	3/16	8/16	9/16	8/16	6/16	16/16
# Workloads	2/14	5/14	10/14	5/14	6/14	14/14

HMC: Hybrid Memory Cube, LLC: Last Level cache, MC: Memory Controller

Table I: Capabilities of Sub-thread Near-data Approaches

Address Pattern	S	Affine A[i]	Indirect A[B[...C[i]]]	Ptr-chasing A = A.next	Multi-op. A[i],B[i]
Load	=f(*S)	O S P N	O P N	N	O S N
Store	*S = f()	L O S P N	L O P N	L N	O S N
RMW	*S = f(*S)	L O S P N	L O P N	L N	O S N
Reduce	σf(S)	A L S N	A N	L N	A S N

A: Active Routing, L: Livia, O:Omni, S: Snack-NoC, P: PIM-en, N: Near-stream
Underline indicates partial support through fine-grain offloading (high overhead).

Table II: Address and Compute Patterns of Near-Data Works

to the core.

Unlike affine streams, the bank for an indirect and pointer-chasing access is data-dependent. Therefore, we do not allow them to have arbitrary streams as operands. An example ineligible stream would be `C[B[i]]+=A[i]`, as it would be burdensome for the `A[i]` stream to compute the bank of `C[B[i]]`. Among the workloads we studied, we never encounter this case. Patterns where a value-producing stream *is* the base stream are supported, like `C[A[i]]+=A[i]`; `A[i]` is included in such an indirect request.

C. Related Sub-thread Near-data Techniques

While prior works have explored sub-thread-level abstractions for near-data offloading, none are both programmer-transparent and support autonomous loops executing remotely. Further, none of them are general enough for all combinations of address and computation patterns. Table I summarizes comparison, and Table II compares supported address and compute patterns to prior works. We explain in detail below.

Active Routing [32] enables offloading of reductions to a network of HMC memories. A programmer specifies a dataflow graph with accesses at endpoints. As it only supports reduction (except pointer-chasing), only 2/14 of workloads are targetable.

Livia [44] offloads single-cache-line accessing functions to the cache or memory controller. Functions may be chained, so Livia can achieve loop autonomy (except for indirect pattern). However, it requires programmers to use an API to identify offload regions, and has no support for multi-operand offload functions. Also, Livia can only modify the data and/or send back a final value, therefore does not support the "load" pattern.

SnackNoC [61] offloads computation dataflow graphs to NoC routers. It requires programmer support through special APIs, and does not support any form of indirect addressing. It also offloads at iteration granularity only.

PIM-enabled [5] offloads programmer-designated instructions to memory; a locality monitor (cache-tag replica) tracks line-level locality and determines whether to offload. Offloading is done at instruction-level only, so offloaded regions are not autonomous (high coordination overheads, gray in Table II).

Omni-Compute [57] offloads RMW instruction chains to LLC banks. Computation is performed in the middle (at the "meet") of remote banks. It has a good expressiveness (covers 10 workloads), but a finer granularity.

Evaluation Baselines: We compare quantitatively against Livia, since it provides loop autonomy and more workloads than Active Routing. We also evaluate Omni-Compute, because it is the only other programmer-transparent technique.

D. Relationship to Prior Stream-Based ISAs

The essential idea of encoding high-level memory access patterns in the ISA to improve various microarchitectural policies has been explored by many prior stream ISA works [18,62,65,67,68], primarily in the context of prefetching. Table III compares their capabilities to generate various access patterns. Note that Prodigy [65] uses a different terminology of *Data Indirection Graph (DIG)* instead of *stream* dependencies.

	Addr. Pattern	Near-Data Compute?
Stream-Specialized Processor [67]	Affine, Indirect, Ptr.	No
Stream-Semantic Register [62]	Affine	No
Unlimited Vector Extension [18]	Affine, Indirect	No
Prodigy [65]	Affine, Indirect	No
Stream Floating [68]	Affine, Indirect, Ptr.	Address Only
Near-Stream Computing (this work)	Affine, Indirect, Ptr.	Addr. + Comp.

Table III: Capabilities of Stream ISA Works

Unlike prior works focusing on address generation, near-stream computing extends streams with a new dimension: computation. With compiler and ISA support (see §III), computations taking or generating stream data are extracted from the original program and associated with streams. They can be *offloaded* along with the stream to the bank (LLC in this work) near the data.

Furthermore, this work develops a coarse-grained synchronization scheme to coordinate the core and remote streams and provide precise states and alias detection (see §IV). When synchronization is not required (through explicit pragmas), near-stream computing introduces new aggressive optimizations (§V), e.g. embedding inner loop streams in outer-loop streams and completely removing the inner loop, which is not supported in SSP [67] or stream-floating [68].

III. Near-Stream Preliminaries

Here we discuss the preliminary components of a near-stream system to enable in-core execution only (i.e. *not* offloading near-data), including the ISA, compiler, and microarchitecture. The approach is inspired by prior work on decoupled stream architectures [67,68], and we note the contrasts in this section. The subsequent section develops the near-data approach.

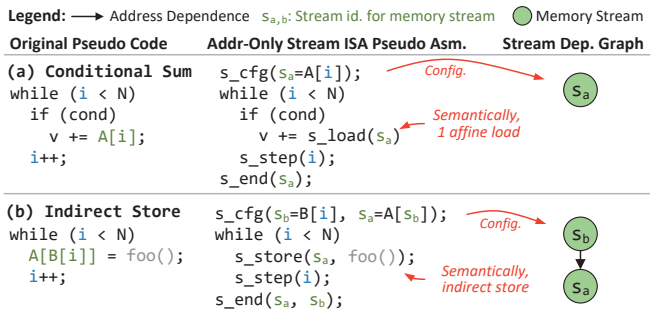


Figure 3: Address-Only Stream ISA Examples

A. Near-Stream Computing ISAs

Address-Only Stream ISA Concepts: The basic ISA abstractions are adapted from decoupled-stream ISAs [67], which focus on address generation and embed no near-data computation. The essential idea is that high-level address patterns can be configured as streams before entering a loop or loop nest, and their data is accessed by special stream load/store instructions. Take the conditional sum example in Figure 3(a). The affine load pattern `A[i]` can be recognized as an affine load stream and configured by a `s_cfg` instruction before entering the loop.

Conceptually, a stream represents a sequence of addresses and data, with a unique stream id (e.g. s_a for `A[i]`), which points to the current iteration. A `s_load` instruction replaces the original load instruction and moves the current iteration's stream data to an architectural register to be consumed by core instructions, and a `s_step` instruction explicitly advances the stream, enabling conditional stream usage and decoupling the address pattern from the control flow. Stream accesses are ordered with other core accesses. Streams with known length autonomously terminate and release all state, e.g. stream id. Streams with data-dependent length are terminated explicitly by a `s_end` instruction.

More generally, stores and atomics can also be represented as streams. Similarly, a `s_store` or `s_atomic` would replace the original access instruction. They have similar semantics as the originals (write or read-modify-write), but with stream generated addresses. Also, the address pattern of a stream can take input from itself or another stream; these dependences are represented within a stream dependence graph, as shown in Figure 3. This enables pointer-chasing and indirect streams, e.g. `A[B[i]]` in Figure 3(b).

Representing Near-Data Computations: Our key innovation is to extend streams with rich and general NDP abstractions, enabling streams to have co-located computations, or *near-stream instructions*. Besides the normal address dependence, streams with computation may also have value dependencies on other streams. Loop-invariant inputs are provided at configuration time. Streams cannot accept loop-variant core values, as it breaks the decoupling boundary.

Near-stream instructions are outlined into a separate function, with the pointer in the stream configuration.

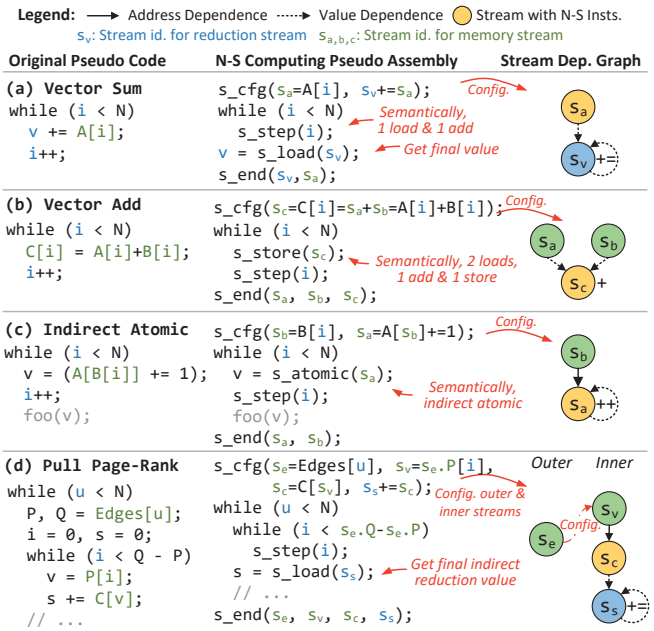


Figure 4: Near-Stream Computing ISA Examples

Computation is wrapped in a loop to facilitate pipelined execution of instances of the near-stream instructions. These functions have no memory access and are stackless. They use `s_load/store` to communicate the stream inputs/result, and `s_step` to advance to next ready computing iteration. These instructions convey no shared memory semantics in this context, and are only used for communication. This approach is general enough for the targeted workloads.

Examples: Figure 4 shows four examples in the near-stream computing ISA, each demonstrating a specific feature.

Reduction - 4(a): A reduction stream s_v sums a load stream s_a. s_v has value dependencies on s_a and itself. The in-loop `s_load` is eliminated as the reduction is decoupled from the core. Instead, after exiting the loop, a `s_load` retrieves the final result.

Store - 4(b): A store stream s_c has two value dependencies on load streams s_a and s_b. The `s_store` recieves *both* the address and stored value from the store stream s_c, and semantically it completes several near-stream operations: 2 loads, 1 addition and 1 store.

Atomic - 4(c): A `s_atomic` instruction performs the atomic operation on the indirect stream address, and returns the new value, which is consumed by `foo()`.

Nest - 4(d): To avoid frequent configuration of short inner loop streams, we extend the stream ISA [67] to allow *nesting* of stream configuration. The inner loop streams' configuration and trip count must only depend on outer stream or loop-invariant data. Each outer stream iteration instantiates a new inner loop stream. A conditional inner loop can also be nested, as long as the condition purely depends on outer streams; this is transformed into predication in the configuration.

Stream Configuration: After code generation, the `s_cfg`

instruction will be split into a sequence of instructions starting with a `s_cfg_begin`, which triggers the hardware to read the stream configuration from cache. This may be followed by a sequence of `s_cfg_input` instructions, which feed any runtime parameters to the hardware (e.g. trip count). Finally, a `s_cfg_end` completes the configuration and the stream can begin executing.

B. Compiler Support

The role of the compiler is to recognize streams by analyzing address patterns (we take a similar approach as prior work [67]), and assign computations to streams to minimize communication. For the latter, we discuss here the heuristics we use for each of the four compute types, within the context of a static-single assignment compiler IR [39]:

Load: For each load stream, the compiler performs a BFS on its user instructions, and checks if visited instructions forms a closure, i.e. no outside users except the candidate final instruction. If so, and the final instruction has a smaller data type, the compiler slices out visited instructions, with the final instruction as the return value. The compiler iterates to find larger closures with fewer bits total in live outputs.

Store: Similar to loads, the compiler searches for instructions computing the stored value, and records a value dependence when encountering a load instruction (or its final instruction).

Reduce: Reduction variables are typically represented as *phi* nodes in the loop entry basic block, and can be recognized by searching backwards for computation instructions. The initial value for reduction is recorded either directly in the configuration (if constant) or as a live input at runtime.

RMW: A load and the following store to the same address are merged into a single update stream. Atomics are handled similarly to stores, with a possible return value. The compiler only targets atomics with relaxed memory order, which only guarantee atomicity and can be reordered with other memory accesses. Therefore, they should not be used for synchronization, e.g. locks. The compiler wraps the near-stream instructions into a loop, and outlines this to a function, with stream instructions to communicate the operands/result. It also inserts necessary stream instructions in the original program to communicate with streams, e.g. `s_store`.

C. Core Microarchitecture

The primary extension is the core's stream engine (abbreviated SE_{core}), which is essentially a programmable prefetcher, supporting the address and compute patterns discussed earlier. Its role is to arbitrate memory requests between concurrent streams, configured by `s_cfg` instructions, and provide data to the core instructions through a FIFO interface (i.e. a load and store FIFO for stream loads and stores).

Near-Stream Computation: Simple scalar near-stream instructions (e.g. `min`) are performed on the SE, similar to other address computation. However, many important workloads are data-parallel and require a vector unit that would be inefficient to replicate. Instead, our approach is to use lightweight thread contexts for executing more general near-stream computation.

The **stream computing manager (SCM)** manages the execution of near-stream thread contexts, arbitrating between requests of the local streams on its SE_{core}, and remote streams from its SE_{L3}, as explained later. Instances of the near-stream function are executed on a lightweight thread called a stream computing context (SCC), for execution with simultaneous multithreading (SMT) [52]. SCCs are lightweight, as near-stream instructions do not contain loads/stores, and as such do not incur long instruction latencies. Therefore, they are allocated minimal physical registers and reorder-buffer (ROB) entries, and no LSQ entries.

As explained earlier, instances of the near-stream function are executed in a loop to avoid the pipeline bubble to trigger a new computation. The SCM is responsible to schedule computation instances onto iterations of this loop. Near-stream instructions access the stream FIFO via stream load/store instructions to read input streams' data and output results. Exceptions in SCCs (e.g. divided by zero) are recorded in the output FIFO entries, and are triggered when the core commits that iteration (similar to prior work [67]). When an SE encounters a near-stream function for the first time, it configures the SCM with the function pointer and any loop invariant operands. Once started, the SCC keeps running until blocked by unready stream inputs (via `s_load`), and is terminated when reassigned to new a computation.

Overall, this scheme allows instruction-level parallelism across near-stream function instances, and provides a low-cost strategy for executing near-stream functions in the core.

Memory Ordering: Similar to prior work, a prefetch element buffer (PEB) is added for memory disambiguation of prefetched data before it is ordered by core memory access instructions [67]; it is a logical extension of the load queue. If an alias is found when checking against an earlier store, all prefetched elements are flushed and reissued, and any dependent stream element is also discarded and recomputed.

Relation to Stream-prefetching/floating: With the system described so far, it is possible to enable stream-based prefetching without necessarily performing near-data computing. With stream-based prefetching only, our design would perform similarly to the stream-specialized processor (SSP) [67].

Stream-floating [68] is an alternate near-data approach with a simpler ISA and microarchitecture, that can offload *only memory read streams with no computation*. It supports *none* of the near-data computing patterns identified in our taxonomy, as it lacks ISA abstractions and microarchitecture for 1. offloading computation, 2. inter-stream dependencies for multi-operand computation, 3. remote writes, and 4. streaming atomics. The following section will describe the challenges and our approach for adding this support.

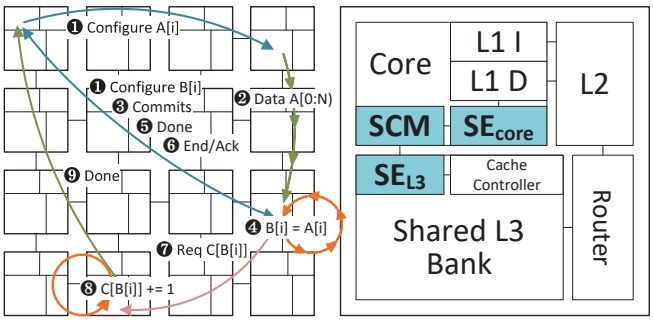


Figure 5: In-Cache Near-Stream Computing Overview

IV. Near-Stream Computing

We first present an overview of the primary challenge and solution, then detail the key innovation of range synchronization in depth, and finally address crosscutting concerns.

A. Major Challenge and System Overview

Challenge and Insight: One major challenge is to synchronize after decoupling streams and computations to the cache. This involves maintaining the precise state and detecting aliasing between streams and the core. A conventional core uses a centralized LSQ to reorder aliased memory accesses. However, in near-stream computing, a remote store stream can also write to memory, making it especially challenging to synchronize.

Intuitively, offloaded computations should not be aliased with other streams or the core, as frequent synchronization eliminates the benefits of offloading. Also, because streams access a single data structure, their addresses tend to be confined in a limited range. In this work, we will further assume this observation extends to physical address ranges, due to the use of large pages or the OS's support for transparently promoting continuous pages into huge pages to reduce fragmentation [55]. Therefore, the synchronization scheme can be coarse-grained and conservative, minimizing the control at the price of false positives. The principle of our approach, range-based synchronization (*range-sync*), is to only synchronize every few iterations and check aliasing against the range of touched addresses instead of individual accesses[2].

Proposed System Overview: Figure 5 shows our proposed system. Besides the core stream engine (SE_{core}), we add an analogous SE to shared L3 banks (SE_{L3}) (Figure 6). The tile where the stream is offloaded is called the "remote" tile.

Near-stream operation begins when the SE_{core} decides offloading would be profitable, and sends the request to the remote SE_{L3} (Figure 6), which requests the stream data from the L3 cache and schedules computations (either on a small scalar unit within the SE_{L3} if simple enough, or issued to the SCM within the same tile). The SE_{L3} also forwards stream data to any dependent streams in other remote SE_{L3}s, and writes results to L3 for store/atomic streams. The SE_{core} issues flow control credits and commit messages to synchronize with remote SE_{L3}s.

[2]Larger but more accurate approximation could also be used to reduce false positives, e.g. bloom filter used in BulkSC [14], and this would not require per-data structure physical address contiguity.

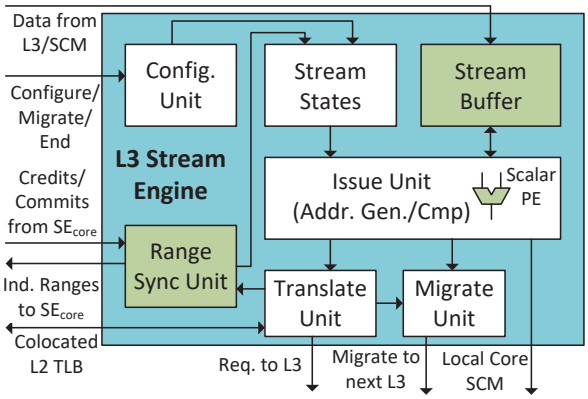


Figure 6: L3 Stream Engine (SE_{L3})

B. Range-Based Synchronization

We first introduce the key concept of ranges and required hardware units. Then we present details of different phases of range-sync, and how it maintains precise state.

Alias Check with Ranges: To amortize synchronization overheads, alias check between core and offloaded streams is performed at *ranges* of touched addresses instead of individual accesses. Specifically, offloaded streams report the accessed physical address ranges $[min, max)$ to SE_{core}. When the core commits an access, it checks against the range for possible alias. Remote streams' progress is either written back after the core commits the corresponding iteration without detecting alias, or discarded in cases of alias, context switch or fault.

Hardware Units: We add a stream buffer to SE_{L3} to hold operands and intermediate states before they are committed (see Figure 6). The range unit listens to translated addresses (by colocated L2 TLB) to build ranges for streams. SE_{L3} caches the current translation so there is only one TLB access per page, (and it also participates in TLB shoot down). For affine streams, since the address pattern is predefined, ranges are built by SE_{core} instead of SE_{L3}, further reducing the synchronization traffic.

Coarse-Grained Protocol: We build the synchronization protocol using ranges, with all control messages designed to be coarse-grained, i.e. one for multiple iterations. This amortizes traffic overhead, and is the key to retaining benefits of decoupling computation to remote tiles. Details follow:

Stream Configure: SE_{core} makes the offloading decision based on the stream's configuration and history information (similar to [68]). If a stream's memory footprint (inferred from the pattern and length) cannot fit in the private cache, it can be directly offloaded. Otherwise, SE_{core} records its miss

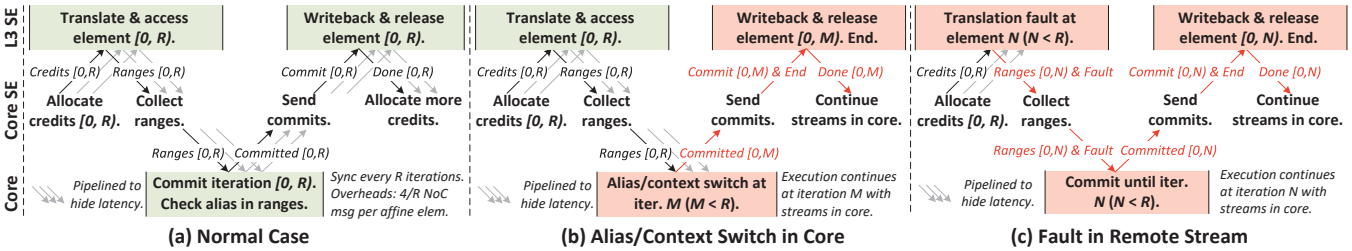


Figure 7: Timeline of Range-Synchronization

and reuse rate in the private cache as well as whether it has aliased with other streams or core accesses. Only streams with high miss rate and no reuse or aliasing are offloaded.

When SE_{core} decides to offload, it sends out a stream configure message to SE_{L3}, containing the stream's configuration, hardware context id (same as core id if no SMT), and address patterns for receiving streams (here `A[]` $\rightarrow$ `B[]`) to determine its current location. When received, SE_{L3} starts to generate stream requests and schedule computations (Figure 5 ❶).

Stream Forward: Once configured, SE_{L3} computes the addresses and issues requests to the colocated L3 cache controller. If the stream data is used by another offloaded stream, SE_{L3} also generates the receiving stream's address of the same iteration, and sets the receiving SE_{L3} as the requester so that the data is forwarded there (Figure 5 ❷). The response contains the stream id and element index, and is buffered in the receiving SE_{L3}'s stream buffer. Streams are issued round-robin.

Compute in SE_{L3}: The issue unit schedules ready computations to a scalar PE (for simple computations) or the local core's SCM within the same tile to fully reuse existing hardware resources. Data in the stream buffer is tagged with the core id, stream id and the iteration number to be able to disambiguate multiple simultaneous iterations.

SCCs executing the same function can be shared among streams from different threads, as each instance is stateless, and this reduces the need to have many SCCs. SCCs in the remote tile are released after all user streams are terminated or migrated out. Since SE_{L3} sends memory requests directly to the L3 cache, now there is no need for the core to issue requests for `s_store/atomic`.

Precise State: Range-sync helps define the architectural state of offloaded streams consistently with the core: a stream element is considered committed if its first user instruction is committed in the core. Figure 7(a) shows how range-sync maintains the precise states for offloaded streams under normal circumstances (R is the granularity in iterations).

To start the range-sync protocol, SE_{core} sends credits to SE_{L3}, allowing it to prefetch and forward the data. Meanwhile, the range unit listens to the translated addresses and builds the touched range $[min, max)$ for each stream. After collecting ranges for a few iterations (currently 8), SE_{core} checks if there is aliasing between streams. If not, the core can commit until the latest iteration with complete range info. Before the core commits a load/store, it checks the address against the ranges for possible alias, and terminates the offloaded streams if it finds aliasing.

If there is no aliasing, SE_{core} sends commit messages to SE_{L3} for store and RMW streams; only then can streams write back to cache. Subsequently, SE_{L3} will reply to SE_{core} with a "done" message, so that SE_{core} can allocate more credits (Figure 5 ❸-❺).

When SE_{core} detects an alias involving offloaded streams (e.g. a false positive due to conservative range check), or when a context switch or exception happens, SE_{core} issues an end message to the remote SE_{L3} to write back committed iterations and release the stream (Figure 7(b)). After collecting all done messages from remote streams, the precise state is restored and the core may continue with streams back in the core. A fault in remote streams also triggers the ending procedure to let the core manage, as shown in Figure 7(c).

Stream Migrate & End: Similar to [68], streams automatically migrate to the next L3 bank as necessary due to address interleaving. To terminate a stream, SE_{core} sends out an end message to SE_{L3} (Figure 5 ❻). Streams with known length can be silently released in SE_{L3} (after committing all work).

Coherence & Consistency: SE_{L3} issues requests to the L3 controller to collect and write back stream data, which can be served normally if no private cache has a copy. Otherwise, depending on the request type (load or store), the L3 cache controller reuses normal invalidation transactions to clear private copies and get the latest version. Coherence states are extended to lock the line for atomic operations (see §IV-C).

At the instruction level, near-data streams only support weak consistency, as remote stores/atomics are written out of order (serializing stores [48] is possible, but reduces near-data benefits). It is the compilers responsibility to ensure strong memory consistency for data-race free programs, which we accomplish by limiting near-stream computation to synchronization-free regions (except atomics with relaxed ordering).

Resource Management: We statically divide the stream state table and buffer in SE_{L3} among cores to avoid sharing. SE_{core} keeps track of resource utilization and may pause issuing credits to avoid possible deadlocks. Another approach is to let SE_{L3} dynamically allocate resources among streams, and

have the SE_{core} terminate streams with no progress after a timeout period (to break potential deadlock). This could lower the hardware overhead, and is left to future work.

C. Efficient Indirection Support

Indirect computation can be offloaded along with the affine stream. Figure 5 shows an indirect atomic increment. After receiving the commit message, indirect store/atomic streams issue the indirect request, compute the result in the indirect SE_{L3}, and reply to SE_{core} (Figure 5 ❼ - ❾).

Intra-Stream Ordering: Range-sync only covers inter-stream and core-stream aliasing. Aliasing within the same stream is not a problem for affine patterns, as they are not self-aliasing and written back in order. However, indirect requests may arrive out of order and violate the memory ordering.

To retain the ordering for indirect streams, the remote SE_{L3} includes the last iteration issued to that bank in newly issued requests. The indirect SE_{L3} can check this against the latest iteration it has seen to detect missing inflight requests and reorder them if needed.

Supporting Atomics: Indirect atomics are common in graph workloads. To guarantee atomicity, the target cache line is locked in the L3 and other accesses are blocked. This usually takes only a few cycles since the computation is fairly simple.

However, the locked window is much longer if we have to send back the value to the core for further processing and wait for commit messages. To mitigate this, we observe that many atomics do not change the value (e.g. compare-exchange in `bfs`, min in `sssp`), and can be served concurrently by recording them in the coherence state (similar to recording the private sharers) and blocking others that modify the value. This hardware multi-reader single-write lock eliminates on average 97% of the contention for `bfs_push` and `sssp`, and reduces the conflict rate to 0.6%. Atomics from the same stream can always proceed even if they modify the same memory, as they are ordered by SE_{L3}.

Indirect atomics may also cause deadlocks, as locks are acquired out of order but released in order when committed by range-sync. The programming model requires shared memory, and it is impossible to eliminate such deadlocks. Therefore, SE_{core} must timeout an offloaded stream with no progress and restore precise state (similar to Figure 7(b, c)). However, this deadlock is very rare and never happened in our experiments.

Indirect Reduction: Reducing over indirect streams is more difficult than affine reduction, as data are likely randomly scattered among banks. A naïve scheme to perform the reduction sequentially, following the data, eliminates the benefits and may introduce more traffic overheads.

To break the recursive dependence, we limit indirect reduction to associative operations, e.g. $+$, $\times$. When offloaded, partial results are reduced in each visited indirect bank, and collected by a multicast message when the stream terminates. SE_{core} performs the final reduction, and only considers offloading if the stream is longer than a threshold (we choose $4\times$ # of banks) to avoid overheads of short indirect reduction.

	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 ($3\times$)	`len`	48	Length ($3\times$)
Ind.	`sid`	4	Stream id.	`size`	8	Element size.
	`base`	48	Base virt. addr.			
Cmp.	`type`	4	Compute type.	`fptr`	48	Func pointer.
	`sid`	4	Arg. sid ($8\times$).	`size`	3	Arg. size 2^n ($8\times$).
	`ret`	3	Ret. size 2^n.	`data`		Const. arg.

Table IV: Near-Stream Computing Configuration

Pointer-Chasing Stream: Pointer-chasing streams migrate among LLC banks following the pointer chain. Similar to indirect streams, SE_{L3} builds and sends back the accessed range. By checking the sending bank of range messages, SE_{core} knows the current location of the stream to send future credits.

D. Stream Encoding

Table IV lists fields of a stream configuration, separated as the access pattern and possible associated computation. We support up to 3-dimension affine patterns. For near-stream computing, we encode simple scalar computations directly in `type`, e.g. $+$, $\times$, RMW, etc., which can be executed by the ALU in SE_{L3}. Otherwise, the computation is encoded in the function pointed by `fptr`, and executed by the local SCM. We support up to 8 inputs (required for 3D stencil) of either streams (with non-zero `sid`) or constants (`data`). The input stream records the receiving stream's address pattern, to determine where to forward the data.

To avoid excessive migration traffic, one optimization is to remember visited banks, and only send core id, stream id and changing fields (e.g. iteration number) when migrating to a visited bank. To terminate an offloaded stream, the end message is multicast to all configured SE_{L3}s. This is left as future work, as we found migration traffic is relatively low.

V. Synchronization-Free Optimization

Although range-sync amortizes the control overhead with coarse-grained messages, it still introduces extra traffic and longer dependence chains. In many scenarios, inter-stream aliasing never happens, and programmers may be willing to sacrifice precise state for performance. This inspires us to introduce the synchronization-free optimization (sync-free), which reduces the control overhead and allows offloaded streams to commit ahead of the core.

Specifically, programmers can add a pragma `s_sync_free` to a loop (Figure 8), indicating that streams in this region never alias. When offloaded, such streams

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 Priv. L2 Cache Replace Policy	32KB, 8-way, 2-cycle lat. 256KB, 16-way, 16-cycle lat. Bimodal RRIP, $p = 0.03$
L1 Bingo Pf. L2 Stride Pf.	8kB PHT, 2kB region 16 streams, 16 pf. per stream
NoC	256-bit 1-cycle link, 8x8 Mesh 5-stage router, multicast X-Y routing, 4 corner mem. ctrl.
Shared L3 Cache	1MB per bank / 16-way 20-cycle lat., MESI coherence Static NUCA, 64B Interleave
DRAM	3200MHz DDR4 25.6 GB/s
SE_{core} (IO4-OOO8)	256B/1kB/2kB FIFO, 12 streams 2 SCCs, total -/32/64 ROB-entry 4/4/4-cycle lat. to SCM
Stream Buf.	16kB FIFO
SE_{L3}	12 streams per core, 768 total 64kB stream buffer, 1kB per core 4-cycle lat. to local SCM

Table V: System and Microarchitecture Parameters

Benchmark	Addr. Cmp	Parameters
pathfinder [15]	MO. Store	1.5M entries, 8 iters
srad [15]	MO. Store	1k×2k, 8 iters
hotspot [15]	MO. Store	2k×1k, 8 iters
hotspot3D [15]	MO. Store	256×1k×8, 8 iters
histogram	Aff. Load	12M 32b value 8b key
scluster [15]	Ind. Load	768k×64B, 5 iters
svm [51]	Ind. Load	384k×64B, 2 iters
bfs_push [9]	Ind. Atomic	Kronecker generated
pr_push [9]	Ind. Atomic	256k nodes
sssp [9]	Ind. Atomic	3.6M edges
bfs_pull [9]	Ind. Reduce	A/B/C: 0.57/0.19/0.19
pr_pull [9]	Ind. Reduce	weight [1,255]
bin_tree	Ptr. Reduce	128k nodes, 8B key 512k uniform lookups
hash_join	Ptr. Reduce	8B key, 256k ⋈ 512k Hit Rate 1/8

Table VI: Workloads (MO: Multi-Op)

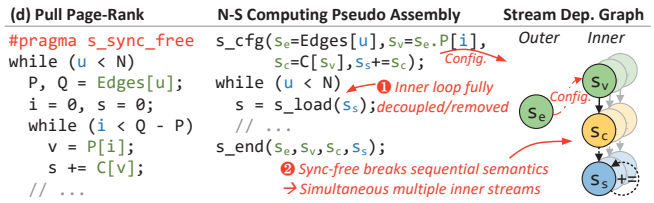


Figure 8: Fully Decoupled Loop (Same Legend as Figure 4)

can commit immediately without sending commit messages or indirect ranges. Streams still report their progress to SE_{core}, and the core is limited to not commit ahead of offloaded streams to avoid complete desynchronizing. This eliminates some control overhead, and importantly, shortens the dependence chain.

Coarse-Grain Context Switch: Without synchronization, streams are free to commit until there are no remaining credits. Therefore, once the credits are sent out the SE_{L3}, there is no sequential point in the original program order. However, coarse-grain context switch is still possible by stopping credit-issue and collecting all done messages. Offloaded streams' progress are included in the architectural state and restored during a context switch.

Fully Decoupled Loop: Synchronization-free streams break the sequential execution semantics to enable aggressive optimizations. As shown in Figure 8, all memory accesses and computations in the inner loop are captured by streams, and all inner loop streams' parameters are from outer loop streams. In such a case, the compiler can eliminate the loop, and these fully decoupled streams are stepped independently by SE_{core}, further reducing core instruction overhead.

More importantly, now SE_{core} can *simultaneously* advance multiple instances of fully decoupled nested streams, increasing potential parallelism (3 concurrent instances in Figure 8).

VI. Methodology

Evaluation Stack: We use gem5-20 [45] for execution-driven, cycle-accurate simulation, extended with partial AVX-512 support, with Garnet [3] for the NoC and DRAMsim3 [40] for DDR4. We implement an LLVM-based compiler with x86 backed to recognize streams and associated computations as described in §III-B.

Benchmarks: We simulate 14 OpenMP workloads from Rodinia [15], MineBench [51] and Gap Graph Suite [9], covering both affine patterns and irregular accesses (see Table VI). `bfs` and `pr` have both a push (using atomic) and pull (using reduction) version. Programs are compiled with `-O3` and vectorized with AVX-512. If not specified, we simulate to completion.

Systems and Comparison: Table V lists system parameters. Energy consumption is estimated using McPAT [41] at 22nm (extended to model the stream engines). We use huge pages for large data structures. In real systems, continuously-used base pages are likely to be promoted into huge pages [55].

The baseline core's L1 uses the Bingo [8] spatial prefetcher, the best multi-core prefetcher in DPC3 [59]. We also add an L2 stride prefetcher, as it improves performance. All other designs have hardware prefetchers turned off:

- **Inst-Level NDC (INST)**: Near-stream computations are offloaded to LLC at iteration granularity, with data forwarded to the "meet" of operands' banks to perform multi-operand computation (similar to Omni-Compute [57]). Reduction cannot be supported due to fine-grained offloading.
- **Single-Line NDC (SINGLE)**: Single cache line accessing functions are offloaded to cache, and may be chained to support pointer-chasing pattern. This resembles Livia [44] and has sync-free optimizations in §V (as Livia does because of programmer guarantees), but without offloading to memory controllers.

  INST and **SINGLE** both benefit from stream-based prefetching even when the compute pattern is not supported. This makes them a stronger baseline than the Bingo prefetcher.

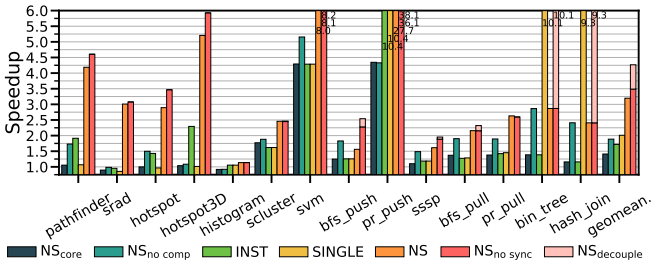


Figure 9: Overall Speedup over Base OOO8 Core

- **In-core Streams (NS_{core})**: SE_{core} is only used as an in-core prefetcher (similar to SSP [67]).
- **Address-only Near-Stream ($NS_{no\ comp}$)**: Streams may be offloaded but *without offloading computation* (similar to Stream Floating [68])
- **Near-Stream Computing (NS)**: Computations are offloaded along with streams among last-level cache banks, with range-sync ensuring sequential semantics and coherence described in §IV.
- **Synchronization-Free Optimizations**: $NS_{no\ sync}$ turns off range-sync as programmers guarantee alias-free. $NS_{decouple}$ further removes unnecessary fully-decouplable loops so multiple streams may be executed simultaneously.

The best baseline for NS is INST (both programmer transparent), and the best baseline for $NS_{decouple}$ is SINGLE (both programmer exposed).

VII. Evaluation

Here we evaluate the overall performance and energy efficiency improvements, generality, and autonomy of near-stream computing with respect to prefetching and prior near-data techniques. Then we discuss sensitivity studies to computation throughput and offload latency, as well as area overheads.

A. Overall Performance/Energy/Area

Figure 9 presents the speedup relative to the baseline OOO8 core. Near-stream computing (NS) significantly outperforms prior prefetching and near-data techniques, achieving 3.19× speedup over the OOO8 core, 1.69× over $NS_{no\ comp}$, and 1.85× over INST. With sync-free support, $NS_{decouple}$ achieves 4.27× speedup over the OOO8 core and 2.12× over SINGLE.

Figure 10 shows the normalized energy-performance tradeoff of different core sizes across workloads. All core types see similar speedups, with inorder cores benefiting the most (4.28× for NS over IO4). Because of the reduced communication and improved performance (less static energy), NS and $NS_{decouple}$ achieve 2.85×/ 3.52× energy efficiency improvement respectively for OOO8 (similar tradeoffs for less powerful cores).

Area: Most of the area comes from the SRAM to store stream configuration and data, and we estimate the area

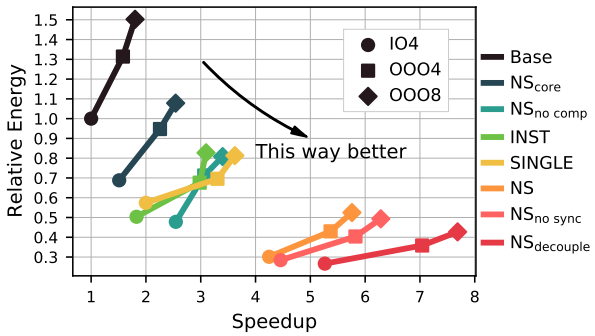


Figure 10: Energy vs. Speedup for IO4, OOO4, OOO8

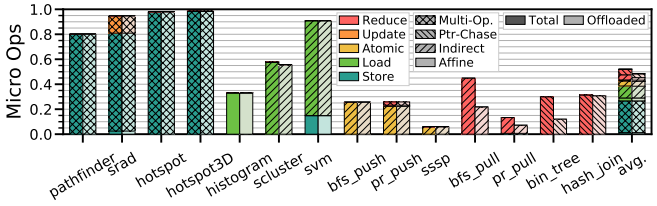


Figure 11: Breakdown of Dynamic Micro Ops

using CACTI and McPAT (22nm). SE_{core}'s stream buffer takes 0.09mm^2 [68]. SE_{L3} requires a 64kB buffer to hold the stream operands and results, which takes 0.195mm^2. Adding the SE_{L3}'s stream configuration (48kB, 0.11mm^2) [68] and other components, the whole chip area overhead is 2.5% for IO4 and 2.1% for OOO8 (SE_{core} in OOO8 has larger stream FIFOs).

B. Advantages of Stream-Based Offloading

With programmer transparency, our NS matches or exceeds INST in all workloads, and our programmer-exposed approach matches or exceeds SINGLE in all tested workloads (while requiring simpler programmer support). This can be attributed to advantages in generality and autonomy.

Generality: Figure 11 shows the breakdown of computing micro ops associated with streams relative to total micro ops (atomic and update are listed separately for clarity). The second bar shows the fraction that is actually offloaded at runtime. NS is capable of offloading computations in all workloads, on average 93% of the possible operations are offloaded. A few short reductions with reuse in private cache (e.g. `bfs_pull`) are kept in-core to avoid frequent stream configuration and termination.

These results also explain why INST underperforms on the last 4 workloads: because it cannot support reduction patterns and can only offload single iterations. Likewise, it explains why SINGLE underperforms on the first four workloads, as they are array codes that operate on multiple arrays. The baseline prefetcher also only excels on affine patterns. Indirect prefetchers may be able to recognize such patterns [53,75], but require training at runtime.

Autonomy: A key benefit of stream-based offloading is to provide autonomy, thus reducing NoC traffic. We evaluate this by examining the NoC traffic and utilization in Figure 12.

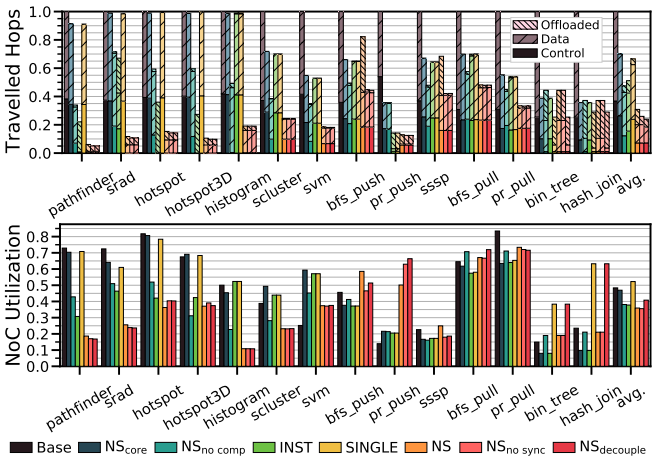


Figure 12: OOO8 NoC Traffic (top) and Utilization (bottom)

Traffic is classified as either *offloaded:* data and coordination messages for near-data computing (e.g. credits, indirect ranges, commits, etc.), *control:* coherence/prefetch messages, or *data:* non-offloaded data accesses and writebacks.

NS heavily reduces traffic (by 69%) by co-locating data and computation. This is accomplished by eliminating control traffic for affine workloads, as now store streams can also be offloaded. More importantly, it also greatly reduces data traffic, as operands are directly forwarded to the bank of the final store. Indirect workloads also benefit; e.g. in `scluster` the stream sends back a scalar value of computed Euclidean distance rather than the original high-dimension data, thus reducing the data traffic. Indirect atomic streams in `pr_push` perform the update in place without bringing the line to the core. Range-synchronization itself accounts for only 11% of NS's traffic. For `bfs_push` and `sssp`, synchronization is more expensive, as it takes two round trips to collect results and commit the buffered indirect atomics.

With synchronization eliminated in $NS_{decouple}$, a total traffic reduction of 76% is achieved. This is especially helpful for performance on `bin_tree` and `hash_join`, as multiple fully-decoupled inner streams can be offloaded simultaneously.

Compared to prior near-data approaches, INST also reduces traffic (by 49%), but has significant overhead due to fine-grain iteration-level offloading. This is apparent on affine workloads, where the traffic is 3-5× higher than NS. SINGLE is of course highly-autonomous, and provides high traffic reductions on the indirect workloads where it is applicable, matching $NS_{decouple}$ in many cases. The traffic is sometimes higher, as SINGLE cannot achieve autonomy on indirect atomics and falls back to iteration-level offloading.

C. Sensitivity to Offload Capability

Figure 13 shows the performance of $NS_{decouple}$, $NS_{no\ sync}$ and NS with varying latency for SE_{L3} to issue a computation to the SCM, normalized to NS-OOO8 with 1-cycle latency. Irregular workloads are insensitive to this latency, as their

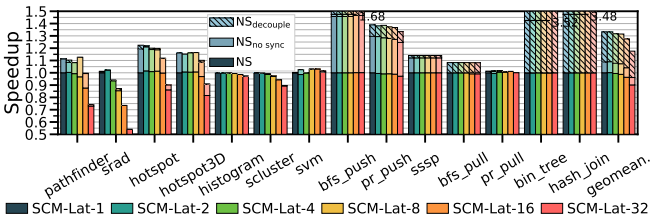


Figure 13: NS, $NS_{no\ sync}$, $NS_{decouple}$ with 1-32 SCM Latency

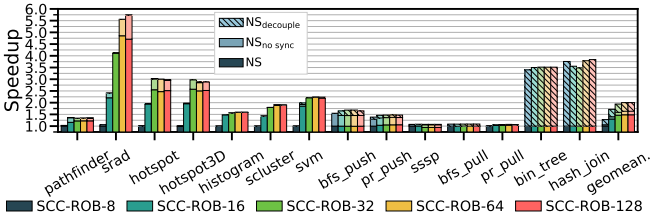


Figure 14: Sensitivity to 8-128 SCC ROB-Entry

computation is simple enough to be handled by the SE_{L3} (except one kernel in `pr_push` and `pr_pull` to update the score). On the other hand, workloads with vector computation are more susceptible to its changes, especially for `pathfinder` and `srad`, which contains a significant portion of short SIMD computations. Overall, near-stream computing can hide much of this latency by overlapping with other streams, and with 16-cycle latency the performance of $NS_{decouple}$ drops by 11% over the default 4-cycle latency.

We also show how the performance changes with limited ROB entries for stream computing contexts (SCC) in Figure 14. As expected, graph and pointer-chasing workloads are not bounded by a small ROB, as their computations are mostly single scalar instructions with less than 10-cycle delay. However, workloads with SIMD operations need a larger ROB to overlap computations and hide the latency to access the local SCM. On the other hand, this also shows that near-stream computing shifts the pressure from data accesses to the real computation, which accounts for the significant speedup. We believe 2 SCCs is a reasonable choice for OOO cores, since it requires less resources than a real hardware thread (less ROB/IQ and no LSQ entries). Overall, we set the default OOO8 configuration to 2 SCCs with total 64 ROB entries.

D. Other Sensitivity Studies

Affine Range Generation: For affine streams, since the address pattern is known at configuration time, SE_{core} can generate the ranges to avoid the traffic of sending them from SE_{L3}, at the cost of duplicate address generation and translation. Figure 15 shows the speedup and traffic of five affine workloads in NS with affine ranges sent by SE_{L3} or generated by SE_{core} (default behavior). The traffic data is classified as control, data and offloaded (same as Figure 12). For indirect workloads, SE_{L3} always sends the range as addresses are data dependent. Overall, generating affine ranges

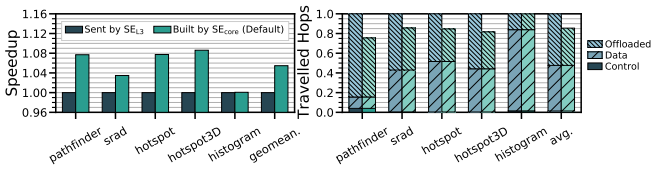


Figure 15: Sensitivity to Affine Range Generation (NS)

at SE_{core} saves 15% traffic and achieves 5% performance improvement. $NS_{no\ sync}$ and $NS_{decouple}$ do not generate ranges as they require no range-based synchronization.

Lock Type: Figure 16 shows the performance of exclusive and multi-reader single-writer lock (MRSW) on the three graph workloads with atomic operations (see §IV-C). Atomics in `pr_push` always modify the value and thus do not benefit from MRSW lock. For `bfs_push` and `sssp`, many atomics do not change the value, and MSRW lock eliminates 97% of contention with 1.29× speedup (NS). For $NS_{no\ sync}$ and $NS_{decouple}$, since there is no synchronization, atomic operations can be committed immediately without waiting for the core, significantly shortening the locking window. Thus, both lock types achieve similar performance. By default, we use MRSW lock.

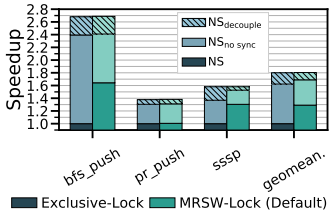


Figure 16: Exclusive vs. MRSW

Scalar PE: Both SE_{core} and SE_{L3} has a scalar PE to handle simple operations and avoid invoking SCM. Figure 17 shows the performance sensitivity for this optimization. As expected, affine workloads mainly contain vectorized instructions and are not sensitive to this feature. Indirect and pointer-chasing workloads benefit from this scalar PE as it reduces the computing latency. Overall, for $NS_{decouple}$, adding the scalar PE improves the performance by 2.5%, but indirect and pointer-chasing workloads significantly benefit from this (1.1× for `hash_join`), as it reduces the computing latency.

VIII. Additional Related Work

We discuss additional related work here; see §II-C for comparison to sub-thread level offloading techniques.

Coarse-grain Offloading: Many near-data approaches use coarse grain abstractions for deciding what to offload. Kernel-level offloading is used in most domain-specific systems [7,12,33,37,43,72,78,79].

Programmable architectures give varying degrees of control over how to schedule threads near data [4,11,19,27,50]. Thread-level offloading also enables programmer-transparency. For example, in the context of GPUs, TOM [30] and Pattnaik et al. [56] transparently decide which code to offload based on dynamic bottleneck analysis and predictive models respectively. AMS adaptively schedules threads in systems with asymmetric memories, using dynamically profiled miss curves [66]. While transparent, they only make decisions at thread granularity.

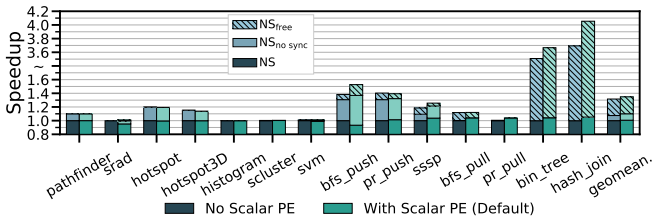


Figure 17: Sensitivity to Scalar PE in SE_{L3}

Near What?: Near-data computing is applicable in many contexts: in-cache [57], near mem-controller [44], near router [61], near-memory [4], near-storage [38], etc. It is future work to evaluate stream-based abstractions, coordination, and offloading in these other settings. Also, several works perform near-data computing using the memory structure itself as bit-serial computation units, either in SRAMs [2,20,23] or DRAMs [24,42,63]. These techniques could provide highly-parallel computation substrates for use in a near-stream system.

Speculative Multithreading: Swarm [34,35,64] along with the T4 compiler [74], executes sequential programs speculatively in parallel as a series of tiny tasks. It supports scheduling such tasks near on-chip data [36]. In T4, near-data optimization is only applied for single cache line tasks.

Coherence and Synchronization: Recent works provide better support for near-data accelerator (NDA) coherence and synchronization. CoNDA speculatively executes NDA kernels while recording their memory accesses in bloom filters, condensing coherence traffic [11]. SynCron [27] provides specialized synchronization without needing coherence. Near-stream's range synchronization protocol supports coherence efficiently by condensing coherence information on a per-stream basis, and is inspired by prior non-NDA work [13,14,46,47,58,60,76].

Prefetching: Prodigy [65] encodes indirect access patterns (similar to nested streams) in the program to efficiently prefetch into L1 cache. Event-triggered prefetcher [6] and Minnow [77] are programmable private-cache prefetchers for irregular accesses. However, prefetching-only techniques still suffer from the traffic overhead to fetch data into the core. Decoupled spatial architectures also leverage stream information for prefetching in accelerator designs [16,17,54,69,70].

EMC [28] augments a memory controller with the capability to execute miss-generating data-dependent instructions. This does provide support for near-data offloading, but only for address generation.

IX. Conclusion

This work explores the idea of using streams as the abstraction for near-data computing. Streams are ubiquitous in data-processing kernels, they enable coarse-grain offloading protocols with low overhead, and they are simple enough

to be extracted with modest compiler extensions. Our implementation enables near-data processing with either zero or minimal (via sync-free) programmer effort, as the compiler and microarchitecture work together to recognize near-stream computing opportunities while retaining precise state. Further, it requires little additional hardware, as the core's pipeline is reused for near-data computation through multithreading.

More importantly, this work breaks with the core-centric view, and enables a new class of optimizations for memory and communication-bound workloads. We believe this approach can enable continued performance scaling and energy efficiency improvements in future large-scale systems.

X. Acknowledgments

We sincerely thank Jayesh Gaur for his extensive feedback and suggestions. This work was supported by funding from Intel's FoMR program, as well as NSF grant CCF-1751400.

References

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *ASPLOS '20*.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *HPCA '17*.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," *ISPASS '09*.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA '15*.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA '15*.

[6] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *ASPLOS '18*.

[7] B. Asgari, R. Hadidi, J. Cao, D. E. Shim, S.-K. Lim, and H. Kim, "Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction," in *HPCA '21*.

[8] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *HPCA '19*.

[9] S. Beamer, K. Asanovi, and D. Patterson, "The gap benchmark suite," 2017.

[10] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *ASPLOS '18*.

[11] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, "Conda: Efficient cache coherence support for near-data accelerators," in *ISCA '19*.

[12] D. S. Cali, G. S. Kalsi, Z. Bingl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Norion, A. Scibisz, S. Subramoneyon, C. Alkan, S. Ghose, and O. Mutlu, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *MICRO '20*.

[13] J. Cantin, M. Lipasti, and J. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," in *ISCA '05*.

[14] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *ISCA '07*.

[15] 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 *IISWC '09*.

[16] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: Exposing the value of flexibility for graph processing accelerators," in *ISCA '21*.

[17] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *MICRO '19*.

[18] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited vector extension with data streaming support," in *ISCA '21*.

[19] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," in *ISCA '17*.

[20] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ISCA '18*.

[21] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA '15*.

[22] I. Fernandez, R. Quislant, E. Gutiérrez, O. Plata, C. Giannoula, M. Alser, J. Gómez-Luna, and O. Mutlu, "Natsa: A near-data processing accelerator for time series analysis," in *ICCD '20*.

[23] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *ISCA '19*.

[24] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *ASPLOS '18*.

[25] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *PACT '15*.

[26] M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *HPCA '16*.

[27] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gmez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "Syncron: Efficient synchronization support for near-data-processing architectures," in *HPCA '21*.

[28] M. Hashemi, E. Ebrahimi, O. Mutlu, Y. N. Patt *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," in *ISCA '16*.

[29] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating linked-list traversal through near-data processing," in *PACT '16*.

[30] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *ISCA '16*.

[31] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *ICCD '16*.

[32] 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 *HPCA '19*.

[33] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *MICRO '20*.

[34] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *MICRO '15*.

[35] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez, "Harmonizing speculative and non-speculative execution in architectures for ordered parallelism," in *MICRO '18*.

[36] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-Centric Execution of Speculative Parallel Programs," in *MICRO '16*.

[37] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *ISCA '16*.

[38] G. Koo, K. K. Matam, T. I., H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading communication with computing near storage," in *MICRO '17*.

[39] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04*.

[40] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*.

[41] 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*.

[42] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO '17*.

[43] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *MICRO '18*.

[44] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *ASPLOS '20*.

[45] J. Lowe-Power and et al., "The gem5 simulator: Version 20.0+," *arXiv:2007.03152*, 2020.

[46] A. Moshovos, "RegionScout: exploiting coarse grain sharing in snoop-based coherence," in *ISCA '05*.

[47] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, "JETTY: filtering snoops for reduced energy consumption in smp servers," in *HPCA '01*.

[48] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence, second edition," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020.

[49] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA '17*.

[50] R. Nair and et. al., "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, 2015.

[51] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *IISWC '06*.

[52] M. Nemirovsky and D. M. Tullsen, "Multithreading architecture," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 1, pp. 1–109, 2013.

[53] 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 *ISCA '18*.

[54] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA '17*.

[55] A. Panwar, S. Bansal, and K. Gopinath, "Hawkeye: Efficient fine-grained os support for huge pages," in *ASPLOS '19*.

[56] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *PACT '16*.

[57] 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 *ISCA '19*.

[58] 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 *MICRO '13*.

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

[60] V. Salapura, M. Blumrich, and A. Gara, "Improving the accuracy of snoop filtering using stream registers," in *MEDEA '07*.

[66] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive scheduling for systems with asymmetric memory hierarchies," in *MICRO '18*.

[61] K. Sangaiah, M. Lui, R. Kuttappa, B. Taskin, and M. Hempstead, "Snacknoc: Processing in the communication layer," in *HPCA '20*.

[62] 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," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, 2021.

[63] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *MICRO '17*.

[64] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *ISCA '17*.

[65] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. OBoyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *HPCA '21*.

[67] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *ISCA '19*.

[68] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream floating: Enabling proactive and decentralized cache optimizations," in *HPCA '21*.

[69] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *ISCA '20*.

[70] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA '20*.

[71] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, "Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching," in *ISCA '21*.

[72] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator," in *HPCA '21*.

[73] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and A. Arvind, "Aquoman: An analytic-query offloading machine," in *MICRO '20*.

[74] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *ISCA '20*.

[75] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *MICRO '15*.

[76] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *MICRO '07*.

[77] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *ASPLOS '18*.

[78] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing communication for pim-based graph processing with efficient data partition," in *HPCA '18*.

[79] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *MICRO '19*.