Title

General, Flexible and Unified Near-Data Computing

Author

Wang, Zhengrong

Publication Date

2023

UNIVERSITY OF CALIFORNIA
Los Angeles

General, Flexible and Unified Near-Data Computing

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Zhengrong Wang

2023

ABSTRACT OF THE DISSERTATION

General, Flexible and Unified Near-Data Computing

by

Zhengrong Wang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Anthony John Nowatzki, Chair

Over the past decades, the memory hierarchy has increasingly become the bottleneck in general-purpose processors due to a widening gap between the growing demand for large data and the much slower scaling of conventional memory hierarchies. Therefore, conventional in-core computing suffers from increasingly expensive overheads such as excessive request messages, unnecessary data movement and coherence traffic, as well as limited off-chip bandwidth, to bring the data from memory to computing cores. To continue the performance and energy efficiency scaling, architects propose near-data computing (NDC) in which computations are offloaded to where the data is. However, existing NDC techniques fall short of providing generality and flexibility across different application domains, programming paradigms, computing substrates, which are crucial to the wide adoption of NDC.

Our key insight is that the critical missing cornerstone for general and flexible near-data computing is a novel rich-semantic memory abstraction. Unlike existing byte-grained load/store operations, the new interface should express a wide range of rich semantics such as the access pattern, reuse distance, near-data computations, etc. Such high-level information is essential for the system to promptly recognize the program's long-term behavior and adjust accordingly to reach optimal states. More importantly, the new interface should be as transparent as possible to programmers with automatic compiler analysis and runtime library support. Based on this, we can fundamentally revolutionize the memory interface and co-optimize computation and data together.

This dissertation explores a new ISA interface - streams - to precisely capture the program's long-term memory and compute activities. Streams are incorporated into the program's functional semantics and are exposed to the entire system stack to guide various policies. Our evaluation and analysis suggest serval key findings. First, a set of useful and prevalent stream patterns cover a wide range of program behaviors and can be embedded into the program in a lightweight way while still maintaining the sequential ordering. Second, streams naturally decouple the address generation and computation from the core pipeline and can be offloaded as the basic unit for near-data computing. Third, by exposing high-level semantics to the system, we can unify different computing paradigms and codesign the software and data structure. Overall, this dissertation aims to enable a general and end-to-end near-data computing system that wipes out the boundary between computation and data – the computation is freely scheduled in the system near the data, and the data is carefully mapped to the memory resources to provide maximal locality and parallelism. Such data-computation orchestration is the key to continuing the performance and energy efficiency scaling.

The dissertation of Zhengrong Wang is approved.

Harry Guoqing Xu

Glenn D. Reinman

Todd D. Millstein

Anthony John Nowatzki, Committee Chair

University of California, Los Angeles

2023

To my parents. . .

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am profoundly grateful to Prof. Tony Nowatzki, my esteemed Ph.D. advisor, whose unwavering support and guidance have been the cornerstone of my academic journey. Our collaboration commenced in 2017 when I was a master's student, and under his mentorship, I have gained invaluable insights that have significantly shaped the trajectory of my research. His dedication, expertise, and encouragement have been instrumental in every step of this dissertation, and I owe much of its success to his profound influence. Without his steadfast assistance, none of this would have been possible.

I also extend my thanks to my committee members, Prof. Todd Millstein, Prof. Glenn Reinman, and Prof. Harry Xu. Their insights during my oral qualification exam and final defense significantly enriched my research, and I appreciate their valuable contributions.

I express my gratitude to an outstanding group of collaborators and lab mates who have played pivotal roles in shaping the diverse facets of this dissertation. Prof. Jason Lowe-Power from UC Davis provided invaluable insights into cache coherence and encouraged my work on gem5. Prof. Nathan Beckmann from CMU engaged in insightful discussions, enhancing my understanding of data affinity for near-data computing. Prof. Lizy John and Prof. Aman Arora offered essential guidance in the detailed modeling of in-memory computing hardware. Dr. Jayesh Gaur contributed significantly to discussions on CPU microarchitecture, while Prof. Jian Weng provided enjoyable insights into compilers and programming languages. Vidushi Dadu's expertise in irregular parallelism and Sihao Liu's patient guidance in hardware intricacies have been instrumental. Special thanks to Christopher Liu for his assistance and the creation of remarkable slides. Each of them has left an indelible mark on this dissertation, and I am deeply appreciative of their collaborative efforts.

A heartfelt thank you goes to my badminton partners, the steadfast companions who brought joy and camaraderie to these years. Dr. Xiaoyu Wu, Dr. Shuaihang Pan, Dr. Hao Liu, Dr. Kaiyuan Jin, Dr. Zizhao Zhang, Ziqing Luo, Jiayu Yi, Jiarui Wang, and Lu Cheng, playing alongside you all was a source of solace during the challenges of the pandemic. Special appreciation to Alex, Andy, Arnold, Harry, Miranda, Yifan, Hallie, Carol, and Dr. Ke Huo for making every Tuesday/Thursday night over the past two years incredibly enjoyable. Xiaoyu, I could not have survived COVID-19 and these years without your help (including cold jokes). Hallie, your impeccable taste in songs added an extra layer of joy; best of luck with your DDS study. To each of you, I extend my wishes for the very best. Keep smashing both on and off the badminton court.

A special shoutout to my dear friends at UCLA, who added vibrant colors to my journey. Dr. Jie Wang, thank you for the exhilarating mountain hikes by Santa Monica Beach, creating cherished memories that will last a lifetime. Dr. Weikang Qiao and Dr. Bing Han, your delightful company and shared culinary adventures have made every meal an enjoyable experience. Congratulations on the arrival of Jojo, and I wish your growing family all the happiness. A heartfelt appreciation to Dr. Licheng Guo, the maestro behind the lens, for skillfully capturing our best moments and turning them into timeless treasures.

I also thank my friends around the world who have left indelible marks on my life. Dr. Zhanhao Su, your unwavering inspiration has been a driving force, urging me to aim high and persevere. Jiajian Kuang, you've been a lifesaver on countless gaming nights, turning each one into a memorable adventure. Hsien-yu Meng, thank you for your steadfast support during my Ph.D. application; I wish you all the best in your endeavors. Liqing Xu, you made Boston unforgettable with your insights and company. To Minjun Huang, your presence has enriched my life in too many ways to count. To the many other friends I've missed during these years of the pandemic, your absence has been felt.

To my dearest parents, Hongbing Wang and Jianhua Yi, words fall short of expressing my gratitude. I may not be the perfect son, but you, without a doubt, are the best parents in the world. Your unwavering love, support, and sacrifices have shaped me into the person I am today. Thank you for being my pillar of strength throughout this journey. I am endlessly fortunate to have you by my side.

VITA

2012–2016 Bachelor of Engineering, Department of Electronic Engineering, Tsinghua University, Beijing, China.

2016–2018 Master of Science, Department of Computer Science, UCLA. Designed a trace-based simulator using LLVM-IR.

2019 Published Stream-based Memory Access Specialization for General Purpose Processors in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA).

2021 Published Stream Floating: Enabling Proactive and Decentralized Cache Optimizations in 2021 International Symposium on High-Performance Computer Architecture (HPCA).

2022 Published Near-Stream Computing: General and Transparent Near-Cache Acceleration in 2022 International Symposium on High-Performance Computer Architecture (HPCA).

2022 Teaching Assistant of CS 33, Spring Quarter, Department of Computer Science Department, UCLA.

2023 Published Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion in 2023 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

2023 Published Affinity Alloc: Taming Not-So Near-Data Computing in 2023 ACM/IEEE 56th International Symposium on Microarchitecture (MICRO).

CHAPTER 1

Introduction

For decades, conventional computer architectures followed the classical von Neumann model which draws a clear boundary between computation and data – centralized process units (e.g. cores in CPU/GPU) perform all the computation, with memory units (e.g. caches, DRAM) reacts to requests and serves the data. Such a compute-centric interface simplifies the design and works well as long as data movement is relatively cheap compared to performing the computation.

However, the landscape is fundamentally changed by an increasing gap between the rapidly growing demand for large data and the much slower scaling of conventional memory hierarchies. For example, ChatGPT requires ∼14000× more parameters than its predecessor in 2017, while high-end GPUs used to train it only scale by 2.5× in memory capacity and 2.2× in memory bandwidth at the same time. To bridge this gap, modern systems also scale up the cache hierarchy to hold more data on-chip and serve the core's requests more timely. However, with the existing compute-centric interface, the cache hierarchy is inherently reactive and responsive, requiring complicated tiling schemes in the software as well as complex best-effort microarchitecture policies to predict the program behavior. Therefore, even with caches, conventional in-core computing still suffers from increasingly expensive overheads such as excessive request messages, unnecessary data movement and coherence traffic, as well as limited off-chip bandwidth, to bring the data from memory to computing cores. This calls for a fundamental redesign to revolutionize the memory interface and co-optimize computation and data together.

To mitigate such overheads, architects propose a variety of specialized architectures that carefully schedule computation near data and orchestrate data movements through efficient pipelines. This broad paradigm of near-data computing (NDC) covers an extremely large design space: near where (on-chip cache, DRAM, or storage), how to compute (small cores, reconfigurable arrays, or bit-serial logic using bit lines), how to program (domain-specific language or general compiler), as well as so many other aspects. Existing near-data computing techniques are limited to a subset of these design choices and often require manual programming using specific APIs, and hence insufficient to enable general, flexible and unified near-data computing.

Key Question How to build a NDC system in which *general* near-data computations can be *flexibly* scheduled to *all* available memory levels and computing substrates, with a *unified* programming and ISA abstraction and *automatic* analysis and optimization across the full system stack.

Our key insight is that such a general and flexible system is infeasible without a powerful unified abstraction to precisely capture and fuse the high-level memory access behaviors and compute patterns. This dissertation explores streams – general memory access patterns with near-stream computations – as a potential candidate. Streams inherently condense the essential information about memory accesses and near-data computations into a unified offloading basic unit, and unlock many opportunities to enable perfect prefetching, general and transparent near-data computing, automatic data layout optimization, etc.

The remainder of this chapter will focus on first introducing the background of near-data computing, and categorizing the huge design space and corresponding challenges to enable general, flexible and unified near-data computing. Then we introduce our approach to leverage streams as the fundamental near-data computing abstraction and how that presents a unique opportunity. Finally, we summarize our main contribution and the organization of this dissertation.

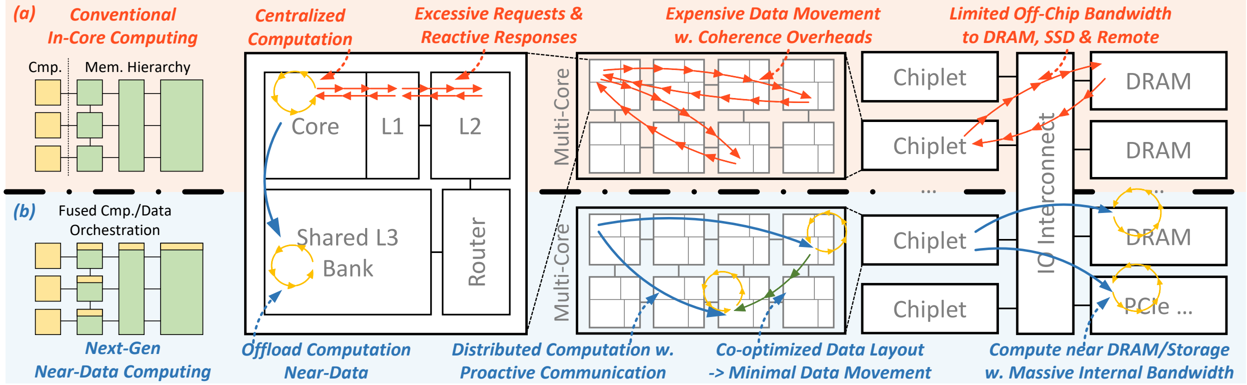

(a)
Conventional In-Core Computing
Cmp.
Mem. Hierarchy
Centralized Computation
Excessive Requests & Reactive Responses
Expensive Data Movement w. Coherence Overheads
Limited Off-Chip Bandwidth to DRAM, SSD & Remote
Core
L1
L2
Shared L3 Bank
Router
Multi-Core
Multi-Core
Chiplet
Chiplet
...
Chiplet
Chiplet
IO Interconnect
DRAM
DRAM
...
DRAM
PCIe ...
(b)
Fused Cmp./Data Orchestration
Next-Gen Near-Data Computing
Offload Computation Near-Data
Distributed Computation w. Proactive Communication
Co-optimized Data Layout -> Minimal Data Movement
Compute near DRAM/Storage w. Massive Internal Bandwidth

Figure 1.1: Conventional In-Core (Top) vs. Future Near-Data Computing (Bottom)

1.1 Existing Near-Data Computing

In this section, we first compare near-data computing with conventional in-core computing, then define the design space for near-data computing and the insufficiencies of existing near-data computing techniques.

In-Core Computing Fig 1.1 compares conventional in-core computing (top) and near-data computing (bottom) across various abstract levels of a multi-core system. In conventional in-core computing, all computations are centralized in the core, which incurs significant data movement and communication overheads to fetch and write back the data. For example, to perform a simple vector addition `A[i]=B[i]+C[i]`, the core needs to fetch all three vectors and write back `C[]` even when there is no data reuse, as the computation is fixed in the core. These overheads are only going to be worse as modern systems continue to scale up in the number of cores and memory hierarchy levels, leading to longer data movement distance. Also, the growing demand for large data puts more pressure on the cache hierarchy to hold and reuse the working set, making it wasteful to move and cache the data.

Why Caches are Insufficient To mitigate these overheads, architects spend tremendous efforts to improve the cache hierarchy in terms of capacity and policies. For capacity, modern CPUs employ larger caches with more levels, e.g. 2MB private L2 cache and 2.7MB shared

L3 cache per core for Intel Xeon 4th Gen Sapphire Rapids. However, a more complex cache hierarchy requires a more complex tiling scheme to improve the data reuse, and is infeasible when the application has no reuse. On the other hand, architects also propose various microarchitecture policies to dynamically bypass certain cache levels, to replace unused cache lines, to prefetch future data, etc. However, these are all best-effort approaches to recovering the original high-level program semantics from the primitive memory interface that operates on individual cache lines, and are inefficient if the program's behaviors are too complicated or transient to be captured.

Fundamentally speaking, with the existing computing-centric interface, caches are inherently reactive and lack a holistic view of the program to proactively adjust to application phases. Even with an ideal cache that perfectly predicts the program's future behaviors, the data movements are inevitable due to the fixed in-core computing. Therefore, we need a fundamental redesign of the memory interface and hierarchy to address these challenges.

Near-Data Computing To overcome these overheads, near-data computing wipes out the boundary between computation and memory. As shown in Fig 1.1, the memory hierarchy is enhanced with computing capability at different levels, so that computation can be scheduled across the memory hierarchy near the data in on-chip caches, off-chip DRAM, or even near the storage. This replaces the superfluous requests and data movements with coarse-grained offloading messages and minimal data traffic to collect the operands. This is the key to continuing the scaling of performance and energy efficiency.

Design Space As Fig 1.1 suggests, near-data computing touches the entire system stack and forms a broad design space that includes at least the following dimensions. Table 1.1 characterizes a representative subset of recent near-data computing techniques. For a more comprehensive characterization, see Table A.1.

Table 1.1: Characterization of Representative Near-Data Approaches

NDC Work	Yr.	ABST.	Near Where	Substrate	Domain	Program	Data Layout
Goal		**Unified**	**All**	**All**	**General**	**Trans.**	**Automatic**
CDCS [1]	'15	Thread	Core	Local Core	General	Trans.	Limited[1]
Dist-DA [2]	'22	DFG	LLC	Core/CGRA	General	Trans.	Oblivious
Omni-Compute [3]	'19	Inst.	GPU LLC	FU	General	Trans.	Oblivious
SnackNoC [4]	'20	Kernel	NoC Router	FU	Regular	API	Scratchpad
Fafnir [5]	'21	Kernel	DRAM[2]	FU	Gather	API	Manual
GenASM [6]	'20	Kernel	DRAM	ASIC	Genomic	API	Scratchpad
EMC [7]	'16	μop Seq.	DRAM	FU	Prefetching	Trans.	Oblivious
To PIM or Not [8]	'22	Thread	DDR Bank	Core	General	Trans.	Oblivious[3]
MeNDA [9]	'22	Kernel	DDR Rank	ASIC	Sparse LA[4]	API	Manual
Tesseract [10]	'15	Thread	HMC	Core	Graph	API	Manual
TOM [11]	'16	Thread	HMC	GPU Core	General	Trans.	Limited[5]
GPU-PIM [12]	'16	Thread	HMC	GPU Core	General	Trans.	Oblivious
ABNDP [13]	'23	Thread	HMC	Core	General	API	Oblivious[6]
PIM-Enabled Inst. [14]	'15	Inst.	HMC	FU	General	Trans.	Oblivious
IMPICA [15]	'16	Kernel	HMC	ASIC	Ptr-Chasing	API	Oblivious
Active Routing [16]	'19	Packet	HMC	FU	Aggregation	API	Oblivious
Gearbox [17]	'22	Kernel	HMC Bank	ASIC	Sparse LA[7]	API	Manual
FANS [18]	'21	Kernel	SSD	FPGA	Sorting	API	Manual
ASSASIN [19]	'22	Kernel	SSD	ASIC	General[8]	API	Oblivious
Neural Cache [20]	'18	Kernel	In-LLC	Bitline	ML	API	Scratchpad
Duality Cache [21]	'19	SIMT	In-LLC	Bitline	General	Trans.	Oblivious
DUAL [22]	'20	Kernel	In-DRAM	Bitline	Clustering	API	Manual
Ambit [23]	'17	Inst.	In-HMC	Bitline	General	API	Scratchpad

[1] Only at page granularity.

[2] In interconnects of DDR ranks and channels.

[3] Customized physical address layout in DRAM.

[4] Sparse matrix transposition.

[5] Specific to strided patterns on GPU.

[6] With DRAM-based cache to capture locality.

[7] Sparse linear algebra, mainly SpMV and SpMSpV.

[8] Programs need to be transformed into streaming computing.

SISA [24]	’21	Set	Multi.[9]	Multi.[10]	Graph Mining	API	Manual
MLIMP [25]	’22	DFG	Multi.[11]	Multi.	GEMM/GNN	API	Scratchpad
NDC Compiler [26]	’21	Inst.	Multi.[12]	FU	General	Trans.	Oblivious
Livia [27]	’20	Kernel[13]	Multi.[14]	Core/FPGA	General	API	Oblivious

- **Offloading Location:** There are two possible offloading dimensions for NDC: *vertically* across memory hierarchy levels, and *horizontally* among units within the same level Many works focus on the vertical approach near the controller of cache, DRAM, HMC, or SSD. Emerging technologies such as in-situ bitline operation turn the SRAM/DRAM arrays into massive vector units (labeled as `In-X` in Table 1.1). As for the horizontal dimension, many multi-core works targetting non-uniform memory accesses (NUMA) and non-uniform cache accesses (NUCA) do not offload computations from the core, but try to schedule and migrate computations horizontally among the cores to improve data locality (labeled as `Core` in Table 1.1). Notice that most vertical offloading approaches have a horizontal dimension within the offloaded level.

  Most existing works focus on a single offloading location. However, depending on the application, different near-data computations may favor different offloading locations, and the optimal offloading level for a single near-data computation can also change depending on the input size and runtime behavior.

  Goal: An ideal NDC system should intelligently schedule computations to a single or multiple efficient levels with minimal synchronization.

[9] In-situ DRAM (SISA-PUM) and near DRAM controller (SISA-PNM).

[10] In-situ DRAM bitline for SISA-PUM, and small in-order cores for SISA-PNM.

[11] In-situ LLC, In-situ DRAM and ReRAM.

[12] NoC routers, LLC controllers, DRAM controllers, inside DRAM.

[13] Can only process a single cache line.

[14] LLC controllers and DRAM controllers

- **Computing Substrate:** Besides the offloading location, different substrates to perform the near-data computation also present different tradeoffs. A small in-order core offers maximal generality but still incurs all the overheads of a general-purpose pipeline, e.g. instruction decoding, register reads and writes, etc. The opposite extreme is fix-function ASIC units, which are highly specialized for certain operations by sacrificing generality. Reconfigurable hardware like FPGA and CGRA, as well as tailored function units (FUs) that can be programmed to perform some predefined primitives, tries to balance programmability and efficiency. There are also many emerging computing substrates, e.g. bit-serial in-memory computing, resistive random-access memory (RRAM), non-volatile memory (NVMe), high-bandwidth memory (HBM), etc. All these techniques pose different tradeoffs.

  Goal: An ideal NDC system should offload computations to the suitable computing substrate, or even split between them to combine their strengths.

- **Application Domain:** Many prior near-data computing techniques are limited to a specific application domain: graph processing, linear algebra, DNA sequencing, database acceleration, sorting, etc. These applications are usually memory intensive and can benefit the most from the improved memory bandwidth of near-data computing. Others try to target general computation by directly offloading an entire thread or providing a general programming interface. We argue that a general near-data computing system is more favorable in the future to handle more and more diverse applications and amortize the costs.

  Goal: An ideal NDC system should be general enough to cover a wide range of application domains.

- **Programming Model:** Another key design choice is how to program such a complex near-data computing system, as much essential information is required to efficiently coordinate various system components. Some works simply program a low-level interface, e.g. intrinsics or assembly instructions, while others provide special APIs in domain-specific or general-purpose languages. All these approaches require manually rewriting the program. Another approach is reusing a general-purpose language and automatically extracting near-data opportunities with minimal programmer hints. This significantly eases the overheads to adopt near-data computing on existing applications with less programmer intervention, but requires advanced compiler analysis and careful microarchitecture optimization to maintain consistency.

  Goal: An ideal NDC system should remain almost transparent to the programmer without sacrificing performance.

- **Data Layout:** A common oversight in designing a near-computing system is data layout. Simply pushing computing into the memory hierarchy does not guarantee that computation is now closer to the data, especially when the computation accesses more than a single piece of data. For example, stencil workloads compute on multiple arrays, and graph workloads require indirect access to neighboring vertices. Without a suitable data layout, the required operands may be scattered far away from each other, and naïvely offloading computation near data may yield no data movement reduction or even hurt the performance. However, existing NDC work either relies on manual coarse-grained data partition, or is simply oblivious to the data layout and falls back to in-core computing when near-data computing is not profitable.

  Goal: An ideal NDC system should automatically optimize data layout to improve data affinity.

On top of Table 1.1 lists the goal of an ideal near-data computing system. Unfortunately, existing NDC works fall short of reaching these goals. They are often limited to certain offloading location, computing substrate and application domain, requires enormous efforts to program, and are oblivious to data layout. We need a general, flexible and unified NDC system that fully realizes the potential of near-data computing.

1.2 Near-Stream Computing

A fundamental limitation of existing NDC techniques is that they still focus more on *computing* than *data*, i.e. they simply shift the conventional compute-centric view closer to the data, but lack of holistic data-centric view. The data is still treated as a small contiguous chunk identified by its address and size, and served by a memory subsystem reacting to individual memory requests. Such primitive memory abstraction completely misses diverse and rich memory behaviors. For example, do memory accesses follow a specific pattern such as linear, affine, or pointer-chasing? What is the computation performed on the fetched data? Are multiple data structures involved in the computation? If so, how should the system optimize their affinity? Without a rich semantic abstraction to efficiently capture this information, it is infeasible for near-data computing to reach the desired level of generality, flexibility and unification.

This dissertation replaces the primitive "data" abstraction with "stream" – a general abstraction with high-level data access patterns and near-data computations – to form a new paradigm called near-stream computing. For example, the above vector addition example `C[i]=A[i]+B[i]` can be represented as three streams: two load streams `A[]` and `B[]`, and a store stream `C[]`. The addition can be associated with the store stream `C[]`, forming a near-stream computation that is scheduled along with the stream. Specifically, near-stream computing presents the following unique opportunities.

- Streams capture broad access patterns, and computations consuming or producing stream data can be associated with the stream, forming near-stream computations. Moreover, streams still preserve the original sequential ordering, and can be automatically recognized by compiler analysis from general-purpose languages without sacrificing programmability or transparency.

- Streams and associated near-stream computations are inherently decouplable from the remainder of the program, making them a natural match for distributed near-data computing. The embedded high-level access patterns also enable proactive and highly efficient data orchestration and synchronization between the core and offloaded streams across the entire memory hierarchy.

- The dependency relationship between streams and near-stream computations precisely represents the affinity relationship between data structures, enabling automatic data layout optimization to make near-data computations *truly* near the data.

Overall, near-stream computing fundamentally takes a data-centric view: it employs a *unified* data abstraction that captures *general* data movement and near-data computations (§2), orchestrates data and computation *flexibly* across the entire memory hierarchy (§3, §4, §5, §7), and automatically optimizes the data affinity to balance spatial locality and parallelism (§6).

1.3 Contributions

This dissertation identifies streams as the fundamental near-data computing abstraction and explores many unique opportunities for near-stream computing. The potential impact of this is to enable general, flexible and unified near-data computing that eliminates communication bottlenecks without sacrificing programmability or transparency. The specific contributions are in terms of stream characterization/ISA extension, unified execution model/architecture design for near-stream computing, and automatic data layout optimization and data structure codesign for data affinity.

Stream Characterization and ISA Extension We characterize various stream patterns in a variety of representative workloads from simple strided linear accesses on an array, to complex data-dependent accesses such as irregular indirect pattern `A[B[i]]` and pointer-chasing `p=p.next`. Streams are prevalent, covering on average more than 60% memory accesses [28]. Based on these observations, we extend a general-purpose ISA (we use x86) with stream abstraction to capture the high-level access pattern and potential near-data computations in the presence of control flow and indirect memory, This serves as the missing cornerstone of next-gen near-data computing. We also implement the automatic compiler analysis and transformation to directly recognize streams from plain-C programs.

In-Core Stream Specialization With streams explicitly embedded in the ISA, we can leverage the inherent structure of streams to specialize the core pipeline, cache interface and cache policies. We find that streams can be decoupled, providing a semi-binding interface that does not require stream data to be consumed. Our stream-specialized microarchitecture benefits from stream-based prefetching, decoupling of address computation, and stream-awareness in prefetch throttling and cache bypassing. Broadly, this paradigm of encoding rich memory access semantics could open up new opportunities for specialization of access and communication at even higher levels within the cache and memory hierarchy.

In-Cache Near-Stream Computing We then explore the idea of decoupling long-term access patterns, i.e. streams, with computations into on-chip caches. Instead of always bringing data to the core, computations are offloaded along with streams and are performed near the data. We propose microarchitecture extensions that can recognize near-stream computing opportunities and offload computations to remote cache banks while still maintaining the precise state with low overhead. It dramatically reduces the network traffic and improves core utilization, all without requiring programmer involvement. More importantly, this work breaks with the core-centric view and explores using memory streams as the basic unit for near-data scheduling. This hybrid approach is key to enabling high-performance and energy-efficient execution in future large-scale multi-core systems.

Software Co-Design for Data Layout We propose the first general and programmable framework that automatically optimizes data layout for any near-data computing technique. The idea is to capture the essential data affinity requirement, i.e. X should be close to Y, in a lightweight memory allocator interface. For example, when allocating the vectors `A[N]`, B[N] and C[N], the programmer can specify that they should be element-wise aligned for near-data computing. Another example is pointer-based data structures, e.g. linked lists, with the new node allocated closer to the previous one. Complemented by co-optimized data structures, runtime libraries, minimal OS extensions and microarchitecture tables, it achieves a clean layered design that systematically captures data affinity information and optimizes data layout throughout the system stack. Evaluated with critical dense scientific computing and irregular graph processing workloads, it achieves 2.36× speedup and 1.82× energy efficiency over a state-of-the-art near-data computing technique. More importantly, it also yields 75% traffic reduction, making near-data computations *truly* near the data.

Fusing Near-Cache and In-Cache Computing While this new stream abstraction captures the essential program semantics, it remains neutral to the underlying hardware details. This makes it possible to apply it to improve the programmability and usability of other emerging computing paradigms, or even to fuse multiple paradigms. A particularly interesting case is bit-serial in-cache computing, in which each cache bitline is a vector lane, forming a massive vector unit (e.g. a 64MB L3 cache with 256x256 SRAM arrays has 2M vector lanes). Due to the massive parallelism, it is especially effective for large dense computations, e.g. matrix operations, but is not as efficient for irregular computations. Hence we need a hybrid paradigm.

Our stream abstraction captures essential program semantics to effectively fuse both paradigms. First, dense regular operations are often represented as computation involving multiple affine streams. By relaxing the sequential semantics of stream and allowing all stream elements to be processed in parallel, we can easily exploit the massive parallelism provided by in-cache computing without introducing another set of abstractions. Second, irregular sparse operations can still be handled as normal near-cache computing using irregular streams. We performed an end-to-end case study on the state-of-the-art point cloud application: PointNet++, in which the dense multi-layer perceptron (MLP) is handled by in-cache computing, while irregular operations, e.g. sampling centroids, gathering neighbor-

Chap.	Topic	Author's Related Prior Work
2	Stream Characterization and ISA	ISCA 2019 [28], HPCA 2022 [31]
3	In-Core Stream Specialization	ISCA 2019 [28]
4	Stream-based Proactive Cache	HPCA 2021 [32]
5	In-Cache Near-Stream Computing	HPCA 2022 [31]
6	Automatic Data Affinity Optimization	MICRO 2023 [33]
7	In-/Near-Cache Computing Fusion	ASPLOS 2023 [30], CAL 2022 [29]
8	Conclusion	

Table 1.2: Dissertation Organization and Relation to Author's Prior Work

ing vertices' feature vectors, are left as near-cache computing. This unifies two computing paradigms using a *single* abstraction, and achieves 1.92× speedup over the baseline, and 1.20× over the best of individual paradigm [29, 30]. This demonstrates the potential of a unified abstraction to fully unleash the benefit of data-computation orchestration.

1.4 Organization

Table 1.2 summarizes the organization of the dissertation and the author's related prior works. In the rest of this dissertation, we first introduce stream definition, stream characterization as well as the proposed stream ISA in Chapter 2. Following that, we discuss how to exploit streams and support in-core stream specialization in Chapter 3. Then we move on to how to leverage streams to enable proactive and decentralized cache optimizations (Chapter 4), and how to schedule computation along with streams (Chapter 5). Chapter 6 enables automatic software co-optimization for data affinity, and Chapter 7 leverages streams as the unified abstraction to fuse in-/near-memory computing. We conclude in Chapter 8.

CHAPTER 2

Stream Basics

In this chapter, we first define and characterize streams in representative applications to demonstrate that streams are suitable candidates for next-gen memory abstraction (§2.1). Then we move to explore how to embed streams in the ISA (§2.2) and extend them to cover near-data computing (§2.3). We also introduce necessary compiler support to automatically recognize streams and transform the program.

2.1 Stream Characterization

A foundational question for a stream-specialized system is whether *programs* exhibit enough streaming behavior to take advantage of. We define four key questions:

Q1 - Coverage: Do streams cover program access?

Q2 - Pattern: What are their access patterns?

Q3 - Length: Are streams long enough to be meaningful?

Q4 - Control: Are they entangled with the core's control flow?

This section attempts to answer the above questions through a trace-based analysis of streams. The observations both justify our motivation and provide insights for the ISA and microarchitecture.

Stream Definition For this analysis, we empirically characterize patterns that may eventually be capturable by an instruction as streams. where the longest extent is defined as the entry and exit of the outermost containing loop.

Desirable Properties Extracting access patterns from memory streams and decoupling them from the von Neumann order of the program is key to achieving high performance and energy efficiency. However, certain memory streams are more amenable than others for specialization. In particular, certain properties are desirable, so we consider streams with these properties to be "qualified":

- **Within an inlinable loop:** This is because saving and restoring streams at function-call boundaries would be more expensive than for a scalar register.
- **Address is Control Independent:** We intend to leave control decisions within the non-stream portion of the program, so that traditional speculative execution may take advantage. We disqualify control-dependent address computation, as supporting this would simultaneously eliminate the benefit of decoupling (close interaction with non-stream instructions), and make analysis for cache specialization more difficult.
- **Affine Strides:** This restriction keeps the hardware for streams trivial (an integer ALU is sufficient) and also enables simple analysis by cache hardware.

Three Clarifications First, these properties only need to hold up to some loop nesting level, because they can be considered to start at that level. Second, *data-dependent streams* (indirect and pointer-chasing) are still potentially quite profitable to target, as their addresses are still control independent. Third, it can still be profitable to target streams where not all elements of the stream's data are guaranteed to be used – i.e. the memory access *can be* control dependent. Note that the access being control dependent is orthogonal to the address being control dependent, and control-dependent access is not disqualified.

Methodology We profile SPEC CPU 2017 [34] to capture general application behavior, as well as CortexSuite [35, 36] to reflect the importance of data processing. To analyze the workloads, we use dynamic instrumentation and trace analysis. We exclude stack spilling accesses as it would inflate the number of affine stream accesses.

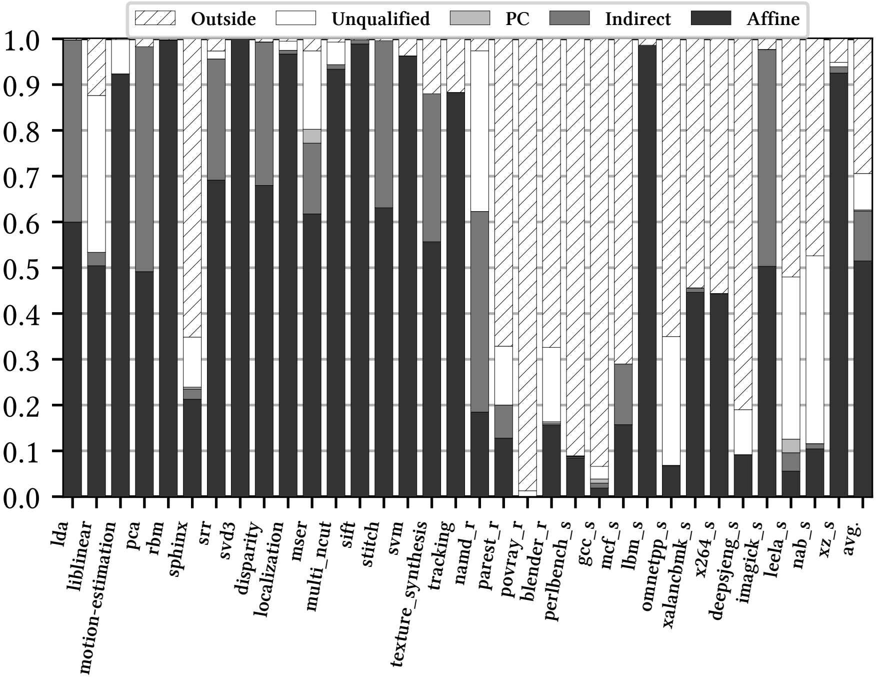


Figure 2.1: Stream Breakdown (PC: Pointer-Chasing)

Q1 and Q2: Coverage and Pattern Fig 2.1 shows the breakdown of dynamic memory accesses. Memory accesses outside of inlinable loops are labeled “outside”. Depending on its access pattern, each qualified stream is further classified as affine, indirect or pointer-chasing (PC). On average, 51.49% dynamic memory accesses belong to affine streams, while 10.90% come from indirect streams and 0.3% from pointer-chasing streams. Although on average indirect streams contribute less than 12%, for some benchmarks more than 40% of stream accesses are indirect, e.g. `namd_r`. These benchmarks require efficient support for dependence between streams to achieve high performance.

Observation 1: More than 60% of dynamic memory access instructions belongs to a stream with specializable properties.

Observation 2: Affine streams are the most common, while indirect streams are also common for some benchmarks.

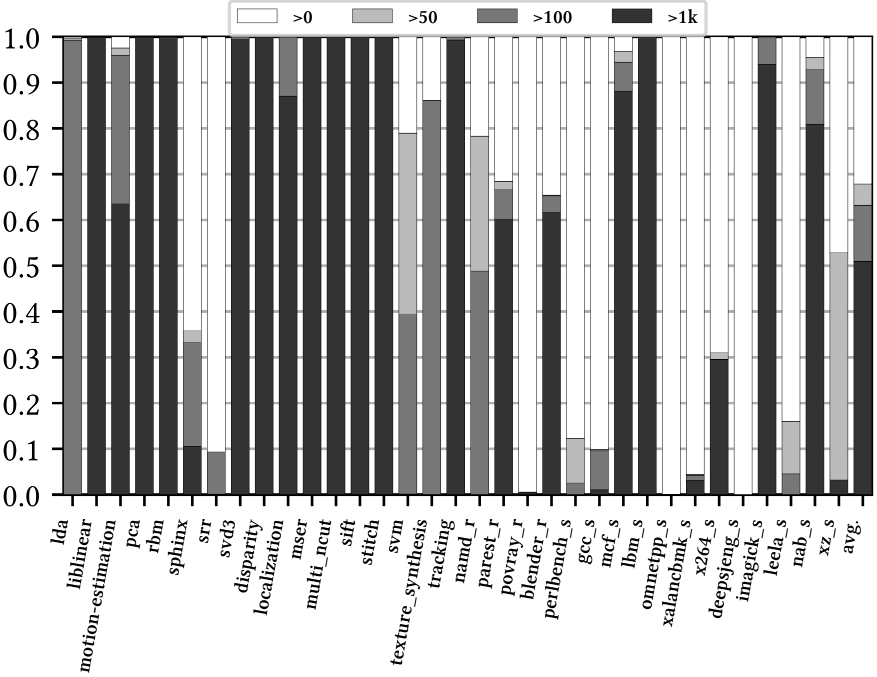


Figure 2.2: Average Stream Length

Q3: Length Fig 2.2 shows the breakdown of stream accesses by the stream length. 51% of stream accesses belong to a stream of length at least 1000, and 62.1% come from streams with length at least 100. Notice that a stream of length N represents at least N loop iterations (greater if we consider the reuse of stream elements). Therefore, even a shorter stream may span across a long instruction window if the loop body is large.

Observation 3: Streams are generally long enough to convey meaningful patterns, while shorter streams are also common, requiring low initialization overhead.

Q4: Interaction with Control For general-purpose workloads, it is common for streams to coexist with the core's control flow. To characterize the degree of this interaction, Fig 2.3 shows the accumulated distribution of stream accesses, grouped by the number of control paths within the loop containing that static memory access instruction. Loops with 3 or more control paths contribute 27.7% of dynamic stream access.

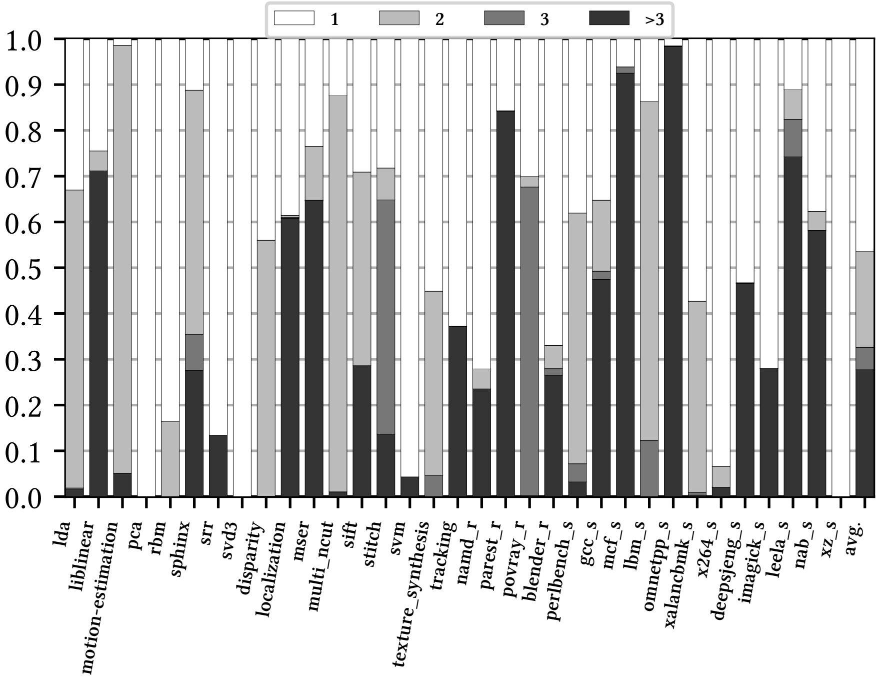


Figure 2.3: Number of Control Paths

Observation 4: Because many stream accesses coexist with control flow, it is essential for the ISA to decouple control flow.

2.2 Decoupled-Stream ISA

In this section, we first make an argument for the requirements of a stream-specialized interface. We then define the decoupled-stream ISA informed by these requirements.

Stream ISA Requirements

In §2.1, we find that streams are common (> 50% of dynamic access), which is promising. However, some streams are shorter (37% less than 100 accesses), streams often have indirect access (about 11%), and streams often coexist with control flow (> 50% of stream accesses). Therefore, we argue that a decoupled-stream ISA interface should have five qualities:

❶ **Integration-simplicity** It should be lightweight and not require excessive core modification, while also efficiently conveying stream patterns to hardware with low overhead for short streams.

❷ **Generality** It should be able to capture both regular and irregular (indirect, pointer-chasing) memory access patterns.

❸ **Pattern-simplicity** The stream definition should be analyzable by hardware (for stream-aware cache policies).

❹ **Control under streaming** It should enable control-dependent access, without interfering with the core speculation.

❺ **Abstract** It should not expose the underlying microarchitecture.

Decoupled-stream ISA Approach Streams are initialized through a configuration instruction that defines the pattern. To communicate with the core pipeline, each stream is assigned a pseudo-register, which is a register implicitly mapped to stream data. This means that instructions which consume/produce stream data remain unmodified, keeping the integration simple (req.❶). Streams may specify other streams as dependences, which enables generality across irregular types (req.❷). Streams are simple to analyze because there are only a handful of common patterns (req.❸).

Streaming under control (req.❹) is possible because of how we update the meaning of each pseudo-register, i.e. the data item a pseudo-register corresponds to within the stream. Specifically, our approach is to add a "step" instruction to the core, which indicates the advancement of the stream from the core's perspective. This implies that data within the stream may be used multiple times, or even ignored if not needed depending on the core control flow.

Streams are general and ubiquitous, and therefore useful across subsequent ISA generations (req.❺). The only aspect of the microarchitecture exposed is the number of pseudo-registers, which represents the number of streams supported simultaneously.

Decoupled-Stream Concepts

The following are the essential components of the ISA extensions:

- **Streams:** Streams are decoupled portions of the program which together generate memory accesses. They are explicitly constructed and deconstructed (`stream_cfg` and `stream_end` instructions), and their data can be accessed by traditional instructions through pseudo-registers.

- **Stream Types and Dependence:** There are two stream types: memory streams describe a memory access pattern; and induction streams define a repeating pattern of values. Memory streams are dependent on either 1. induction variable streams (affine access patterns), 2. other memory streams (indirect access patterns), or 3. themselves (pointer-chasing).

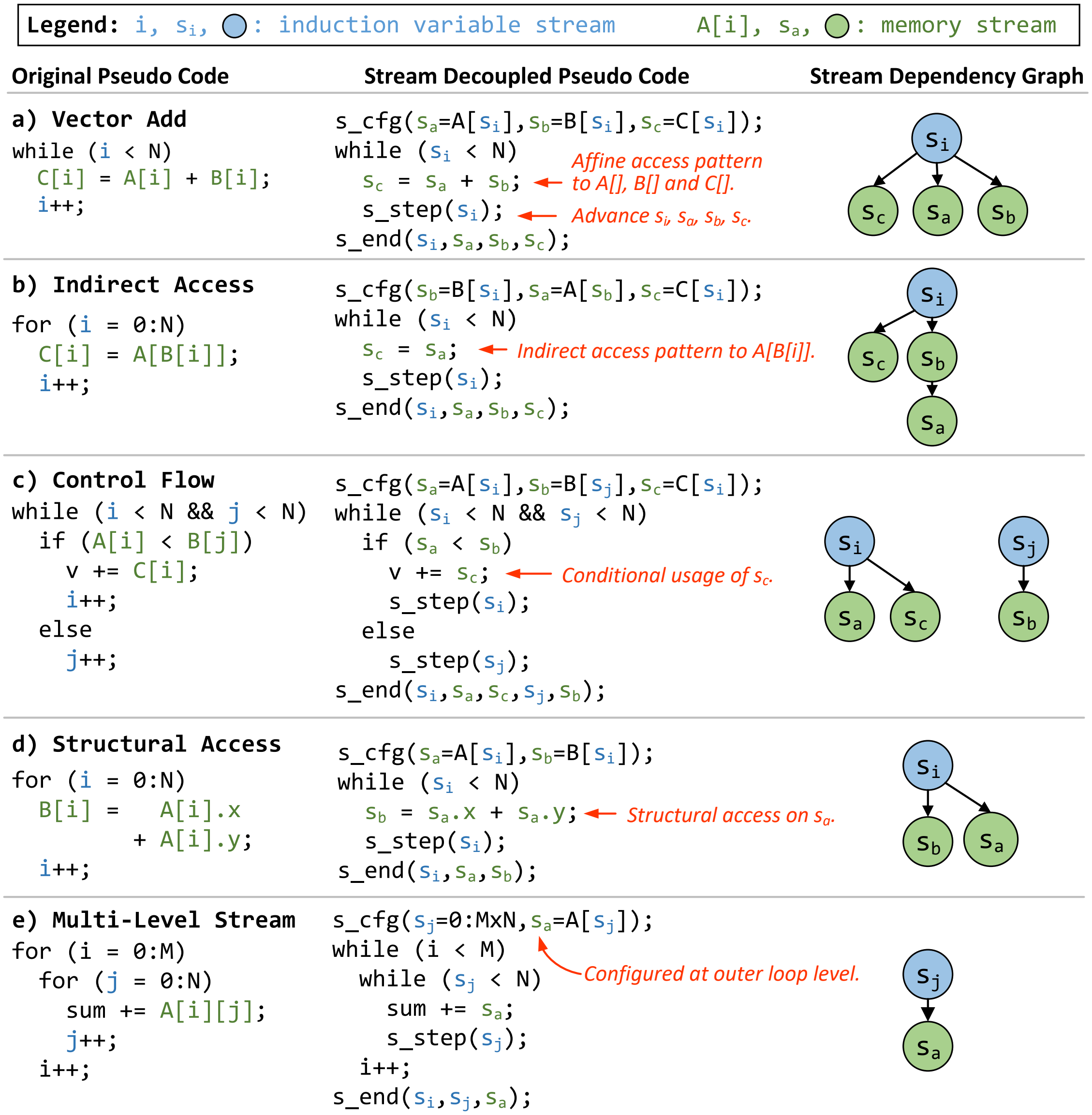
- **Pseudo-registers and Stream Stepping:** A pseudo-register is a register which refers to a stream's data. The meaning of the register, the position into the stream, is updated by a `stream_step` instruction to the associated induction variable stream. In other words, a `stream_step` advances the pseudo-register position of all dependent streams.

- **Memory Semantics and Architecture State:** Semantically, a load occurs at the point of the first use of a pseudo-register corresponding to a load stream after stepping or configuring, while a store happens at every write to a pseudo-register of a store stream. Pseudo-registers become part of the architecture state after their first use and are removed from the architecture state after stepping the corresponding induction variable stream.

- **Pseudo-register Width:** Pseudo-registers have a definable width, which determines the amount of data read by each step instruction. Instructions that access narrower portions of the register specify an offset.

Stream-ISA Extensions

To explain the ISA intuitively, we describe its principles and potential through a series of examples that stress its different aspects. Fig 2.4 shows five examples, with the decoupled-stream pseudo code, and the stream dependence graph.



Legend: i, s_i, ●: induction variable stream   A[i], s_a, ●: memory stream

Original Pseudo Code | Stream Decoupled Pseudo Code | Stream Dependency Graph

a) Vector Add

```
while (i < N)
  C[i] = A[i] + B[i];
  i++;
```

```
s_cfg(s_a=A[s_i],s_b=B[s_i],s_c=C[s_i]);
while (s_i < N)
  s_c = s_a + s_b;
  s_step(s_i);
s_end(s_i,s_a,s_b,s_c);
```

Affine access pattern to A[], B[] and C[].

Advance s_i, s_a, s_b, s_c.

b) Indirect Access

```
for (i = 0:N)
  C[i] = A[B[i]];
  i++;
```

```
s_cfg(s_b=B[s_i],s_a=A[s_b],s_c=C[s_i]);
while (s_i < N)
  s_c = s_a;
  s_step(s_i);
s_end(s_i,s_a,s_b,s_c);
```

Indirect access pattern to A[B[i]].

c) Control Flow

```
while (i < N && j < N)
  if (A[i] < B[j])
    v += C[i];
    i++;
  else
    j++;
```

```
s_cfg(s_a=A[s_i],s_b=B[s_j],s_c=C[s_i]);
while (s_i < N && s_j < N)
  if (s_a < s_b)
    v += s_c;
    s_step(s_i);
  else
    s_step(s_j);
s_end(s_i,s_a,s_c,s_j,s_b);
```

Conditional usage of s_c.

d) Structural Access

```
for (i = 0:N)
  B[i] =    A[i].x
         + A[i].y;
  i++;
```

```
s_cfg(s_a=A[s_i],s_b=B[s_i]);
while (s_i < N)
  s_b = s_a.x + s_a.y;
  s_step(s_i);
s_end(s_i,s_a,s_b);
```

Structural access on s_a.

e) Multi-Level Stream

```
for (i = 0:M)
  for (j = 0:N)
    sum += A[i][j];
    j++;
  i++;
```

```
s_cfg(s_j=0:MxN,s_a=A[s_j]);
while (i < M)
  while (s_j < N)
    sum += s_a;
    s_step(s_j);
  i++;
s_end(s_i,s_j,s_a);
```

Configured at outer loop level.

Figure 2.4: Decoupled Stream ISA Examples

Basic Operation – Fig 2.4(a) The example is a dense vector addition, using three affine streams. There are two load streams `A[i]`, `B[i]` and a store stream `C[i]`, which are all dependent on an induction variable stream `i`. Each stream is assigned a pseudo-register, which is used by the traditional instructions to interact with streams. Next, we explain the use of stream instructions.

`s_cfg`: A `s_cfg` instruction is inserted before entering the loop which uses the stream's data. It defines all of the streams within this loop level, including their type (induction/memory), pattern (stride, width, and optional length), dependences, and starting address[1]. This interface conveys stream information at a coarse granularity, using a stable interface.

In practice, after the configuration is complete, the hardware may begin fetching data ahead of the core's requests based on the program. Also, note that the stream's data never needs to be consumed, though an unused stream would occupy a pseudo-register.

`s_step`: As described earlier, the `s_step` instruction advances the pseudo-register position of the induction variable and dependent streams. In this example, stepping `s_i` will also advance `s_a`, `s_b` and `s_c` by one element. This highlights how the approach of implicitly stepping dependent streams avoids redundant step instructions.

An alternative decoupled ISA could have used a "destructive read" interface, where a read of a pseudo-register implicitly advances the state. This would have worked well in this example, eliminating the need for step instruction. However, this would not allow control-dependent access, as described shortly.

`s_end`: The `s_end` instruction deallocates a set of streams from the corresponding pseudo-registers. Generally, this happens after the loop in which the stream use occurred, but an explicit `s_end` enables the termination of a stream to be data-dependent.

[1] This can be implemented with a series of instructions for each stream. While this is shown abstractly in the figure, in our implementation, it is an instruction cache load of configuration data, interpreted by the hardware.

Indirect Memory Access – Fig 2.4(b) Indirect memory access is supported by making the address of one memory stream dependent on the value of another. In this example, `s_a` is dependent on `s_b`. We also refer to `s_b` as the base stream of `s_a`. Note that `s_a` is also stepped with the `s_step` of `s_i`.

Control Flow – Fig 2.4(c) The `s_step` interface enables the ISA to specify control-dependent access, meaning that a stream element may be used 0 times, once or many times. This example iterates over the elements of `a[i]` and `b[j]`, but their relative ordering is data-dependent. This is implemented by conditionally stepping stream `s_i` and `s_j` depending on the outcome of the comparison. Having a `s_step` instruction makes it trivial to support such a scenario, by simply replacing the increment instruction with a corresponding `s_step`.

Notice that here not every element of `s_c` will eventually be used. In a traditional ISA, such unused elements make it harder for the prefetcher to figure out the access pattern and prefetch for future elements. With the help of the compiler and the support of explicit control on when to step the stream, we effectively decouple the access pattern from the control flow. This also enables a new opportunity for the hardware, as now it knows the addresses and can speculate whether the stream element will be used and whether it should prefetch.

Coalescing Streams – Fig 2.4(d) In some situations, memory access patterns become more regular when coalescing from two static instructions. A common scenario is iterating through an array of structs, as shown in the example. Here the structural accesses on `x` and `y` fields can be coalesced into a single stream, where the pseudo-register width is now doubled. This reduces the total number of streams and also makes the access pattern contiguous.

To support this, the user of a pseudo-register may add an immediate offset parameter to specify the offset from the head of the pseudo-register[2]. In this example, `s_a.y` has an offset of 4 bytes (assuming `int32_t` data type).

[2] In theory, this support could be added to the ISA by extending each instruction or adding a header byte to specify the offset. In our implementation, we add this information to the stream configuration.

Multi-Level Streams – Fig 2.4(e) It is sometimes advantageous to configure a stream at an outer loop level to increase the length. This example iterates over a 2D array and is transformed into a single memory stream. Because N is known and there is no conditional stepping, the affine access pattern can be determined before entering the outer loop. The induction variable `i` is not specialized as a stream in this example, while the induction variable stream `s_j` iterates from 0 to $M \times N$.

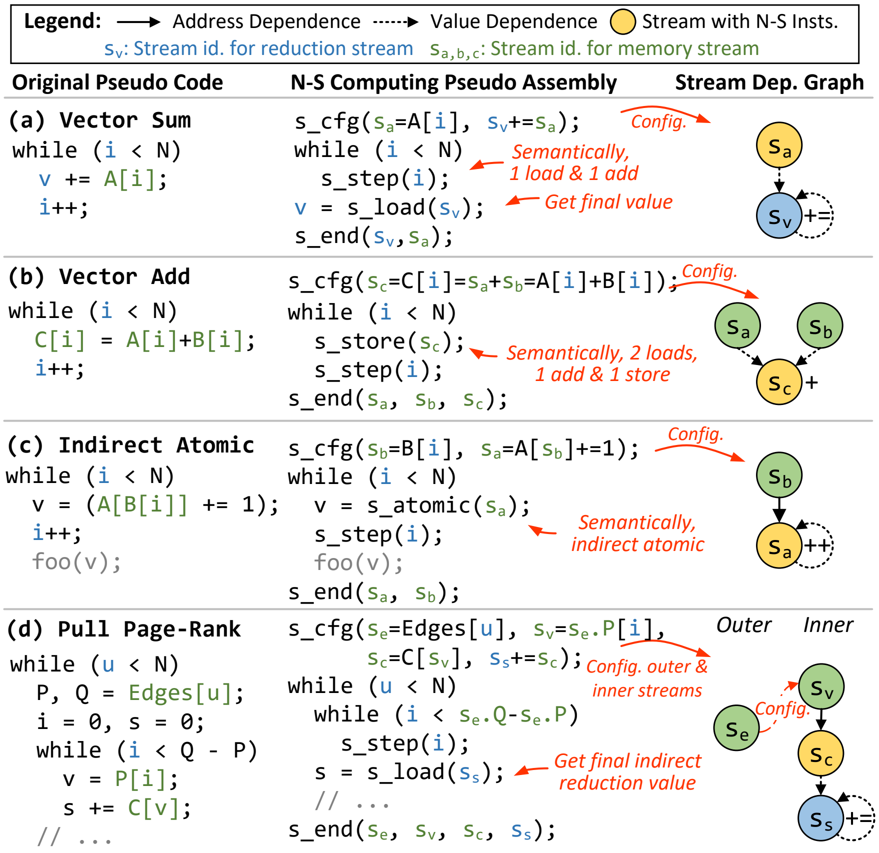
Pattern Limitations and Speculation

The address patterns that we support are limited to those that are decouplable, i.e. determined at the point of configuration. There are two relevant caveats: 1. data may be conditionally used, and 2. the outermost dimension of the pattern can have an unknown length. This corresponds to the two forms of speculation that we allow for address patterns: that cache lines in the pattern are likely useful, and that streams are long enough that the overhead of loading a few extra items is acceptable.

This has implications for how many loop levels we can hoist up the configuration of a stream. If at a given outer level either the trip count of the inner loop becomes unknown, or the induction variable becomes conditionally stepped, then the decoupling invariant can no longer be maintained.

2.3 Near-Stream Computing Abstraction

Here we discuss extensions to decoupled stream ISAs [28] to associate streams with computation, as well as the compiler support.



Legend: Address Dependence Value Dependence Stream with N-S Insts.
s_v: Stream id. for reduction stream s_a,b,c: Stream id. for memory stream
Original Pseudo Code
N-S Computing Pseudo Assembly
Stream Dep. Graph

```
(a) Vector Sum
while (i < N)
  v += A[i];
  i++;
```

```
s_cfg(s_a=A[i], s_v+=s_a);
while (i < N)
  s_step(i);
v = s_load(s_v);
s_end(s_v,s_a);
```

```
(b) Vector Add
while (i < N)
  C[i] = A[i]+B[i];
  i++;
```

```
s_cfg(s_c=C[i]=s_a+s_b=A[i]+B[i]);
while (i < N)
  s_store(s_c);
  s_step(i);
s_end(s_a, s_b, s_c);
```

```
(c) Indirect Atomic
while (i < N)
  v = (A[B[i]] += 1);
  i++;
  foo(v);
```

```
s_cfg(s_b=B[i], s_a=A[s_b]+=1);
while (i < N)
  v = s_atomic(s_a);
  s_step(i);
  foo(v);
s_end(s_a, s_b);
```

```
(d) Pull Page-Rank
while (u < N)
  P, Q = Edges[u];
  i = 0, s = 0;
  while (i < Q - P)
    v = P[i];
    s += C[v];
  // ...
```

```
s_cfg(s_e=Edges[u], s_v=s_e.P[i],
      s_c=C[s_v], s_s+=s_c);
while (u < N)
  while (i < s_e.Q-s_e.P)
    s_step(i);
  s = s_load(s_s);
  // ...
s_end(s_e, s_v, s_c, s_s);
```

Figure 2.5: Near-Stream Computing ISA Examples

Near-Stream Computing ISAs

We extend streams to define co-located computation, or *near-stream instructions*. Many types of streams can have near-stream instructions: load streams (computing using loaded values), store streams (computing values to be stored), atomic streams (the atomic function is defined by the stream), and special reduction streams (computing using its previous result and loaded values). When there is a choice of associating an instruction with one or another stream, the compiler decides based on heuristics to optimize data traffic and reuse (see §2.4).

Besides the normal address dependence for the access pattern, streams with computation may also have value dependencies if they take other streams' data as their computation inputs. We define the user as the value-dependent stream and the provider as the value-base stream. 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 in a separate function, with the pointer in the stream configuration. 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 the 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 Fig 2.5 shows four examples in the near-stream computing ISA, each demonstrating a specific feature.

Reduction - 2.5(a): A reduction stream $\mathtt{s}_\mathtt{v}$ sums a load stream $\mathtt{s}_\mathtt{a}$. $\mathtt{s}_\mathtt{v}$ has value dependencies on $\mathtt{s}_\mathtt{a}$ 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 - 2.5(b): A store stream $\mathtt{s}_\mathtt{c}$ has two value dependencies on load streams $\mathtt{s}_\mathtt{a}$ and $\mathtt{s}_\mathtt{b}$. The `s_store` recieves *both* the address and stored value from the store stream $\mathtt{s}_\mathtt{c}$, and semantically it completes several near-stream operations: 2 loads, 1 addition and 1 store.

Atomic - 2.5(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 - 2.5(d): To avoid frequent configuration of short inner loop streams, we extend the stream ISA [28] 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.

2.4 Compiler Support

We implement compiler support to identify streams and transform the original program to decouple streams with near-stream computations. We implement the compiler transformations using LLVM IR [37]. Programs are transformed and compiled to an extended x86 backend with new stream instructions.

Decoupled-Stream ISA

To support decoupled-stream ISA, there are three phases: recognizing stream candidates, selecting qualified candidates, and code generation.

Recognizing Stream Candidates The compiler treats every static memory access instruction in a loop as a candidate for a memory stream, and every ϕ node in the loop entrant basic block as an induction variable stream. ϕ nodes not in the loop entrant basic block represent other control-dependent values and are not considered as candidates. Starting from the candidate instruction, the compiler performs a backward search on its operands, gathering instructions until it encounters a loop-invariant, a constant, or another candidate instruction. It will also record dependencies between stream candidates.

Selecting Stream Candidates After finding the candidates, the compiler identifies all candidates qualified for stream decoupling. First, a candidate can only be qualified if it has a simple enough pattern to match the supported affine, indirect, and pointer-chasing patterns. Specifically, it can not contain any ϕ node, which represents control-dependent address generation. Also, it should not contain any unsupported operations, e.g. floating point operations.

Second, the compiler checks the dependencies between streams. A trivial constraint is that if any of its base streams within the same loop level is unqualified, the stream is unqualified. A more sophisticated case is to handle multiple induction variables. To support configuring streams in outer loops, if the address pattern limitations in 2.2 are satisfied, we remove the dependency on any outer loop induction variable so that the memory stream depends on only one innermost induction variable (the iteration domain is incorporated into the inner loop variable). If this is not possible, the stream becomes unqualified.

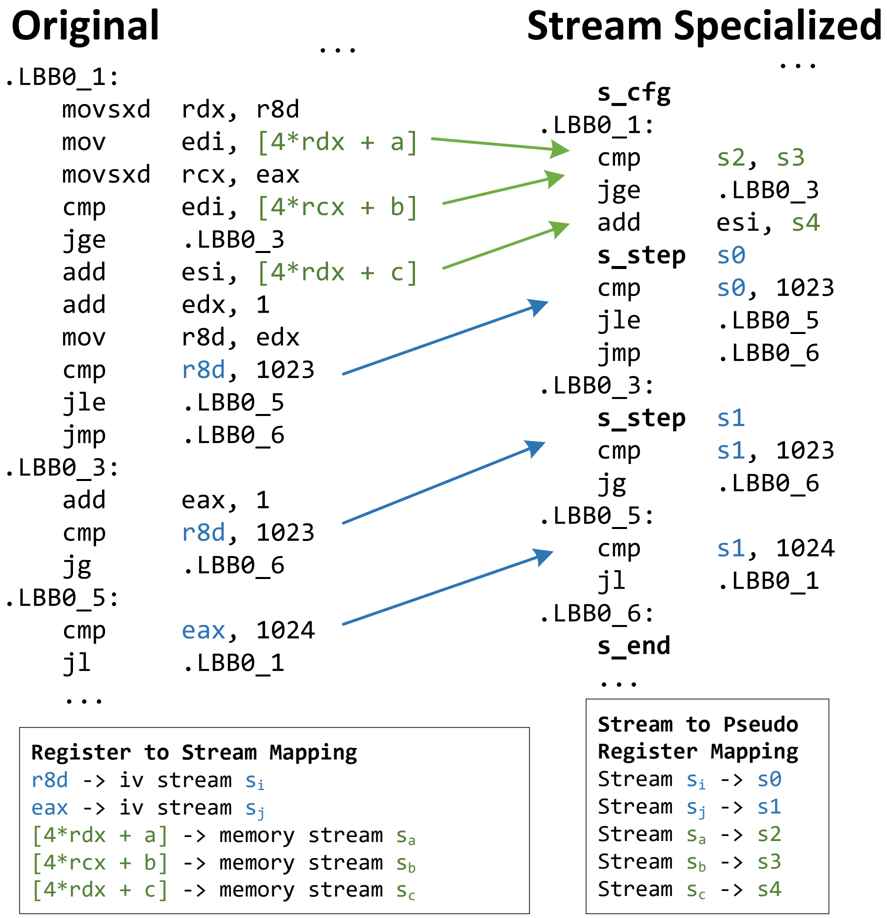
During this phase, the compiler coalesces affine streams with the same induction variable and small offsets between their elements. The compiler also drops some qualified streams if the total number of streams exceeds the maximum. The compiler prioritizes memory streams with no dependent streams to drop, as they are less likely on the critical path.

Similar to some prior work [38, 39, 40], we take a hardware/software codesign approach to memory aliasing. The compiler records which loads and stores may alias, so that non-aliasing streams can bypass the core's LSQ.

Code Generation During the code generation phase, the compiler first generates the stream configuration for the selected candidates. The configuration specifies 1. which pseudo-register to represent the stream; 2. the type of the stream (induction, load, store); 3. loop invariant values (stride, width); and 4. stream dependencies.

The compiler transforms the loop by 1. inserting `stream_cfg`, `stream_step` and `stream_end` instructions; 2. replacing the operand of a user instruction with the corresponding pseudo register, along with the offset within the element (for a coalesced stream); 3. removing the memory access instruction for a memory stream, and possibly insert a dummy user instruction to ensure the original program order is preserved; and 4. if there are no other users, remove the address computation instructions.

Fig 2.6 shows both the original and transformed X86 assembly code for example in Fig 2.4(c). The stream operands are replaced by the corresponding pseudo-registers.



```
                             ...
.LBB0_1:
    movsxd   rdx, r8d
    mov      edi, [4*rdx + a]
    movsxd   rcx, eax
    cmp      edi, [4*rcx + b]
    jge      .LBB0_3
    add      esi, [4*rdx + c]
    add      edx, 1
    mov      r8d, edx
    cmp      r8d, 1023
    jle      .LBB0_5
    jmp      .LBB0_6
.LBB0_3:
    add      eax, 1
    cmp      r8d, 1023
    jg       .LBB0_6
.LBB0_5:
    cmp      eax, 1024
    jl       .LBB0_1
    ...
```

```
                   ...
    s_cfg
.LBB0_1:
    cmp      s2, s3
    jge      .LBB0_3
    add      esi, s4
    s_step   s0
    cmp      s0, 1023
    jle      .LBB0_5
    jmp      .LBB0_6
.LBB0_3:
    s_step   s1
    cmp      s1, 1023
    jg       .LBB0_6
.LBB0_5:
    cmp      s1, 1024
    jl       .LBB0_1
.LBB0_6:
    s_end
    ...
```

```
Register to Stream Mapping
r8d -> iv stream s_i
eax -> iv stream s_j
[4*rdx + a] -> memory stream s_a
[4*rcx + b] -> memory stream s_b
[4*rdx + c] -> memory stream s_c
```

```
Stream to Pseudo
Register Mapping
Stream s_i -> s0
Stream s_j -> s1
Stream s_a -> s2
Stream s_b -> s3
Stream s_c -> s4
```

Figure 2.6: Decoupled Stream Assembly Example

Support Near-Stream Computing

Ideally, we could formulate this as an optimization problem to find the best slicing between computations and streams. However, we find a heuristic-based approach is capable of handling the existing workloads. Specifically, the compiler tries to recognize load computation, store computation and reduction one by one:

Load For each load stream, the compiler performs a BFS on its user instructions, and checks if visited instructions form 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 the visited instructions, with the final instruction as the return value. The compiler iterates to find larger closures with fewer bits total in live outputs. Algorithm 1 covers the details.

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 backward 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 guarantees 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/results. It also inserts necessary stream instructions in the original program to communicate with streams, e.g. `s_store`.

Algorithm 1: Find Load Computation

```
Result: ComputeInsts, FinalInst
ComputeInsts ← ∅, FinalInst ← Load;
Visited ← Frontier ← {Load}, NextFrontier ← ∅;
while !Frontier.empty() do
    forall Inst : Frontier do
        forall User : Inst.users() do
            if User in Visited or !isComputeInst(User) or !isSameBasicBlock(User) then
                continue;
            NextFrontier.insert(User)
    if NextFrontier.size() == 1 then // check closure
        FormClosure ← true;
        CandidateFinalInst ← NextFrontier.front();
        forall Inst : Visited do
            forall User : Inst.users() do
                if not User in Visited and User != CandidateFinalInst then
                    FormClosure ← false;
                    break;
    if FormClosure and CandidateFinalInst.size() ¡ FinalInst.size() then
        ComputeInsts ← Visited;
        FinalInst ← CandidateFinalInst;
    Visited ← Visited ∪ NextFrontier;
    Frontier ← NextFrontier;
    NextFrontier ← ∅;
```

CHAPTER 3

Stream-Specialized Processors

Even without near-stream computations, streams by themselves are powerful abstractions. In this chapter, we develop stream-specialized processors (SSP), which leverage the high-level semantics exposed by *only* streams to improve performance and energy efficiency. Specifically, we first overview the proposed stream-specialized processor (§3.1) with detailed microarchitecture design (§3.2) and stream-aware policies (§3.3). Then we describe the methodology in §3.4 and evaluate stream-specialized processors in §3.5. Finally, we discuss related works in §3.6.

3.1 Stream Specialization Overview

Given the premise of explicitly encoding streams in the ISA, the core pipeline can be extended to explicitly manage the stream context and the relative index in each stream. We first overview our approach in terms of the microarchitecture of the stream-specialized processor (SSP), and then discuss how it unlocks the following opportunities depicted in Fig 3.1.

Microarchitecture Approach We modify the core pipeline's front end to track the position within each stream based on interpreting step instructions. We add a *stream engine* (SE) to generate addresses and interact with the memory system. Finally, we add a load-stream FIFO (and store-stream FIFO), which core instructions may access when loading (and storing) pseudo-registers. Streams may be configured and accessed speculatively; and a simple protocol enables the rollback of stream positions and configurations on misspeculation.

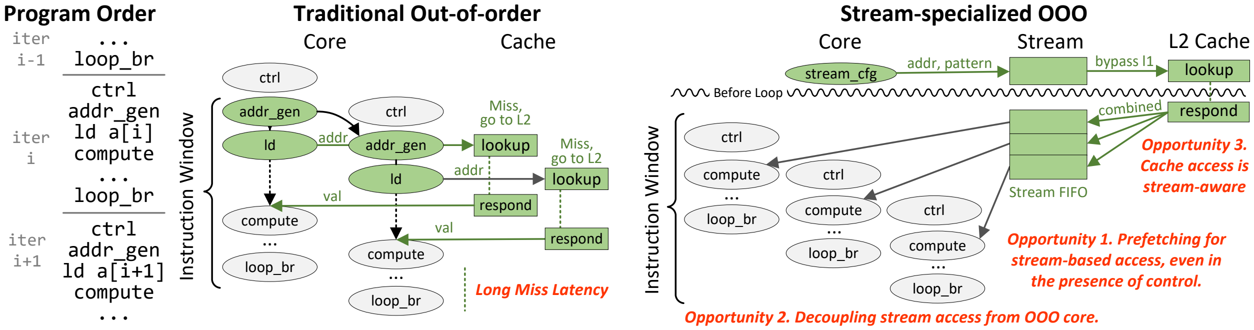


Figure 3.1: Overview of Stream Specialization Paradigm vs Traditional Out-of-Order

Opportunity 1: Stream-based prefetching Knowing access patterns and their relationship to the core's control flow can lead to a very effective stream-based programmable prefetcher. As shown in Fig 3.1, the prefetcher can understand when exactly to make a request based on how far ahead of the core the prefetch is. Similar to other programmable prefetchers, this would enable the scheduling of memory requests far past the limits of a traditional OOO core processor's instruction window.

Our Approach: Stream requests are decoupled from the core's instruction window, enabling deep prefetching for regular and irregular memory access. The primary benefit of decoupling is reducing the negative impact of long-latency memory accesses, without requiring a large instruction window. Maintaining the relationship to the control flow of the core through the "step" instruction enables the prefetcher to keep an accurate distance without running ahead and polluting the cache.

Opportunity 2: Stream decoupling Stream primitives can further be incorporated into the functional semantics of the program to enable what we refer to as *semi-binding* prefetching. Following the principle of decoupled access execute [41], a specialized memory access engine would generate requests corresponding to streams, and ordinary core instructions can access stream data through registers that are mapped to this data (we call these pseudo-registers). There are several potential advantages, including the removal of address-generation instructions from the general core pipeline, coalescing accesses from related streams into a smaller number of requests to cache, and also reducing the possibility of cache pollution through timely, semi-binding prefetch.

Our Approach: The principle of stream decoupling is to create a direct interface between data which is stream-prefetched, and the core instructions. This eliminates redundant address generation which is typical of programmable prefetchers, and simultaneously reduces instruction pressure on the core pipeline. Besides, the benefits of vectorization of memory are brought to traditionally non-vectorizable code; stream loads fetch data in units of the L1 bandwidth, even though a particular code may have too much control flow to be otherwise vectorized, and our design requires no vector shuffling.

Decoupled streams are what we call *semi-binding.* They are binding in that they are obligatory and consume registers. However, they are non-binding in that *not* all data must be consumed, and so the hardware can ignore memory protection faults for non-consumed data. Therefore, stream-decoupling keeps the benefits of binding prefetch, even in the presence of control flow and indirect access. Also, the prefetch distance can be controlled through dynamic throttling, reducing the negative impact of being obligatory.

Opportunity 3: Cache Awareness Streams are precise definitions of an access pattern. Various cache policies could take advantage of advanced knowledge of these patterns: replacement policies, dead-block prediction, cache bypassing etc. One specific idea is to let the cache bypass streams based on the expected footprint of the stream.

Our Approach: The stream engine has access to high-level information regarding streams, through stream configuration instructions. Using this information, and supplemented by the access pattern, the stream engine can make requests to the cache in a way that is aware of stream behaviors.

We specifically explore the idea of exposing the *footprint* of the stream to the cache. A footprint is an under-approximation of the total number of cache lines accessed. Knowing

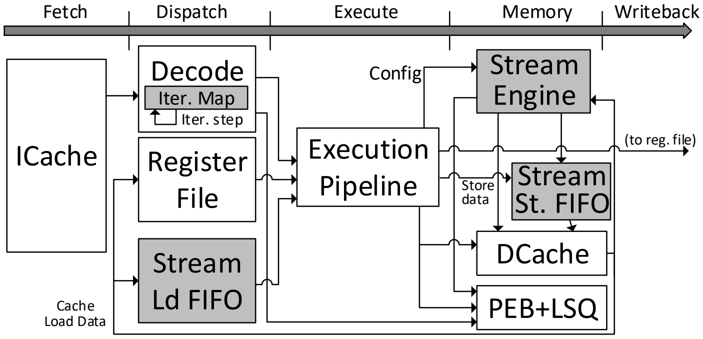


Figure 3.2: Stream-specialized Pipeline

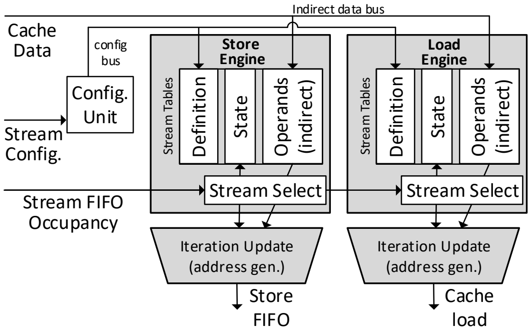


Figure 3.3: Stream Engine

the footprint in advance can lead to an enhanced cache bypassing policy, where requests from a high-footprint stream (that would not fit in e.g. an L2 cache) with low temporal reuse are bypassed to larger caches so that they do not evict useful data.

3.2 SSP Microarchitecture Extensions

A traditional processor can be extended with a small number of relatively simple structures to create a stream-specialized processor (SSP), as we depict at a high level in Fig 3.2. SSP extensions have four basic responsibilities: 1. Maintain the *core's view* of stream position based on configuration and stepping instructions; 2. Maintain the *streams' decoupled view* of their state, and allow streams to issue memory requests; 3. Maintain the data that decoupled between the core's view and the streams' view, and enable core instructions to access this data; 4. Keep the above consistent during misspeculation and exceptions. We overview each of the corresponding components:

Core's view – Iteration Map The front end of the pipeline maintains the iteration map (Fig 3.4), which counts iterations of induction variable streams, as seen by dispatch. A `s_cfg` instruction updates the mapping from the stream index to the iteration count table. A `s_step` increments the iteration count table. User instructions of a stream access the table to ascertain the current iteration, which is used to index into stream FIFOs.

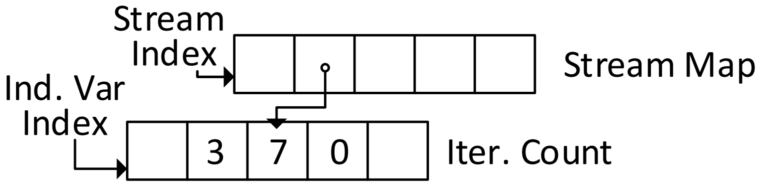


Figure 3.4: Iteration Map

Streams' View – Stream Engine The stream engine (SE) is the central component of a stream-specialized design, as shown in Fig 3.3. It contains an induction table to hold iterator parameters, and a load engine and store engine, which generates load and store requests to memory. Multiple streams may be mapped to each engine.

To explain the operation, first, a `s_cfg` instruction will load data to the stream engine's configuration unit. This will initialize the designated streams and parameters on the load and store engines. When the unit receives notice of a committed `s_end`, the associated stream is deallocated from the load or store engine.

The load and store engine maintains three tables describing the state of any stream. The first is the stream's definition, containing the pattern (affine, indirect, linked) and parameters (stride, width, optional length). The second is the stream's state, essentially the memory-side view of the induction variables. This is where the current address is stored. Finally are operands, which store any dependences on the data of other streams (for indirect streams and pointer-chasing streams). Each stream can have up to two dependencies on other streams, and for each dependence, we keep enough space for four iterations worth of storage for any given dependence to run ahead.

Each cycle, the stream select unit picks a stream based on the readiness of corresponding operands (if any) and whether remaining FIFO entries are allocated to the stream (see Section 3.3 on page 39 for allocation policies). Requests are in units of per-port L1 cache bandwidth (64 bytes in our design). In parallel with sending the request, the stream's state is updated for the next iteration.

Decoupled Data – Stream FIFOs The stream FIFOs are responsible for holding decoupled state either from or to memory (load and store FIFO). We use an implementation similar to the dynamically partitioned queues of Outrider [42], which uses a pointer table to virtualize a single wide buffer into multiple FIFO queues (in our case, one for each concurrent stream). For core instructions which consume stream data, they would access the load stream FIFO instead of the register file[1]. For stores to streams, core instructions only produce values, and addresses are produced by the stream engine. These are combined at the store stream FIFO before being sent to the memory system.

Control Mispeculation Stream requests and uses are speculative to avoid pipeline serialization. We discuss implementation in the context of an R10K [43] style merged register file. To maintain the core's view, during misspeculation rollback while the map table is being reverted using register mapping information stored in the core's reorder buffer, the iteration map is also similarly decremented for each misspeculated step instruction. If no `s_cfg` instruction is misspeculated, only the core's view of the stream is reverted, because the addresses for streams are control-independent. This means we achieve a low-cost form of selective replay [44] by virtue of semi-binding prefetch.

When reverting a `s_cfg` instruction, both the core's view and streams' view is reverted. On the core side, the stream map entries are freed, and the corresponding streams are deconfigured within the stream engine. Data stored within decoupled FIFOs corresponding to these streams is flushed.

Precise State and Context Switch Precise states and exceptions can be supported using the same speculation recovery mechanisms as above. Because the stream configuration and pseudo-register values (specifically, those that have not been stepped since the last use) are part of the architecture state, they must be saved on the context switch. These items amount to less than 1KB for our design.

[1] An alternate design could partition the physical register file for use as a stream-FIFO.

Interaction with Memory Before issuing a stream request, the virtual address is translated by the core's MMU. Access to TLB can be delayed to favor core loads, but address translation needs only occur once per page for affine streams with low strides, reducing TLB access in the common case.

Because stream loads effectively aggressively reorder loads, may-alias streams require memory disambiguation and recovery. For this, the stream engine relies on the core's LSQ to perform memory disambiguation, along with its memory dependence predictor (similar to MAD [38]). When dispatching a core instruction that semantically triggers the memory access, it is inserted into the core's LSQ as a normal load/store. To detect RAW dependence between a store and a prefetch stream element, the stream engine also maintains a prefetch element buffer (PEB). The PEB can be considered a logical extension of the LQ, which contains the prefetched elements by the stream engine. Elements in the PEB are freed when the first use is dispatched, or when the element is released as unused. Traditional memory order checking is performed between SQ and LQ + PEB. Hitting in the PEB indicates a misordered stream access, and the streams' view should be reverted. Overall, may-alias streams can still be aggressively reordered, but do not reduce the LSQ-energy.

To implement a non-relaxed memory model, SSP needs to be integrated with the core's memory-consistency speculation mechanism (e.g. if relevant coherence state changed, flush core pipeline and roll back streams' view).

Finally, because the ISA semantically only performs memory operations if a pseudo-register is accessed, memory faults from prefetching stream requests are delayed until the execution of the corresponding user instruction. Faults are silently ignored if the FIFO entry is unused.

3.3 Stream-Aware Policies

Now with the stream-specialized microarchitecture aware of high-level stream information, we can leverage it to design effective prefetching and stream-aware cache bypassing policies.

Stream Prefetch Distance and Throttling

One common problem for prefetching is to determine a suitable prefetch distance. An ideal prefetcher would bring in the data precisely when the user instruction is ready to be issued. Thus, a waiting user instruction is an accurate signal that the prefetcher is falling behind. It is straightforward to leverage this information within a decoupled-stream microarchitecture, as the user instruction checks the readiness of the FIFO entry before issuing. Since the data is prefetched into the FIFO, allocating a different number of FIFO entries to a stream will effectively change its prefetch distance.

A simple policy would be to split the FIFO evenly for all stream pseudo-registers. This reduces the hardware complexity of managing the FIFO. However, this leads to low utilization, as FIFO entries for unassigned pseudo-registers will be wasted. Also, streams with different memory footprints may hit different cache levels and require different prefetch distances to hide the memory latency. A better policy is to dynamically allocate FIFO entries on demand.

Dynamic Throttling We implement a stream-aware dynamic throttling policy. Each stream is assigned a FIFO occupancy N. Associated with each FIFO entry is a 1-bit `late` flag, which is set by the issue logic when the stream operand is the last operand to be ready. Each stream is assigned a 3-bit `late_counter`. When releasing a FIFO entry, the `late_counter` is incremented if `late` is set, and decremented otherwise. When the late counter reaches a threshold (currently 7), the stream is considered lagging behind the core and its N is increased (by 2) if N is smaller than a maximum threshold T and the sum of all configured streams' N does not exceed the total FIFO size. Having a maximum size T

avoids the pathological case when a stream occupies most of the FIFO. N is initialized to a small value when configuring the stream, which helps capture different behaviors of the same stream during different phases.

Possible Extensions The compiler could provide a suggested initial value for N when generating stream configurations, by leveraging the information of stream memory footprints, profiled latency, etc. Another opportunity is to use the dependencies between streams to prioritize those with dependent streams, as they are more likely on the critical path. These are left to future work.

Stream-Aware Cache Bypassing

Caching data with low temporal locality unnecessarily wastes the cache capacity and hurts the performance. It is beneficial to identify and bypass such requests.

Our insight is that streams inherently contain useful information for the cache to make such a bypassing decision, e.g. memory footprint, stream length, reuse distance, etc. Ideally, the cache should bypass a stream when the storage required to achieve temporal reuse is beyond its capacity. Bypassing correct streams brings two major benefits: 1. It avoids polluting the cache with data that will not be reused; and 2. Since a bypassed stream is not cached, the cache can speculate that a request from that stream will miss and immediately forward the request to the next level of cache without waiting for the tag lookup or allocating an MSHR. Tag lookup is still necessary to detect misspeculation, but it is removed from the critical path for the common miss case. Not allocating an MSHR increases memory parallelism by allowing more misses to be handled simultaneously.

To better understand how stream information can help cache bypassing, consider the following examples:

Example 1: Repeatedly iterating over two affine streams, where the cache can hold only one stream. Without bypassing, the cache tries to keep both streams and results in a 0% hit

Field	Description	Field	Description
`sid`	Stream id	`miss`	# cache misses
`footprint`	Est. mem. footprint	`reuse`	# cache reuses
`request`	# stream requests	`bypass`	Whether to bypass

Table 3.1: Fields of Stream Table

rate. With the footprint of the stream, the cache can reason that the total storage required to cache both streams is beyond its capacity, and thus bypass one stream. The other stream now can be fully cached, which improves the hit rate to 50% and reduces the bandwidth pressure to lower cache levels.

Example 2: Iterating over one large affine stream that can not fit in the LLC. In such a case, there are no benefits to caching it. Bypassing it will increase the memory parallelism. The benefit of stream-awareness is knowing the footprint at the time of stream configuration.

While useful, stream information is not sufficient to handle all situations. For example, it is impossible to accurately estimate the memory footprint of an indirect stream. Also, there may be some temporal reuse from non-stream requests, and bypassing the cache for such a stream hurts the performance.

To mitigate this, a hybrid policy is used to leverage both the stream information and dynamic statistics. In the cache, a stream is identified by the `s_cfg`'s PC and pseudo-registers (`sid`). Some lower bits of the PC are used to distinguish streams with the same pseudo-register from different regions. We augment the cache with a stream table. Table 3.1 gives a basic description of each field of the stream table. An `sid` field is also added to the tag representing which stream brought in this cache line.

Stream Configuration: After configuration, the stream engine will send a request to the cache, which contains all configured streams' `sid`s and their memory footprints. For affine streams with known length at configuration time, their memory footprint can be estimated

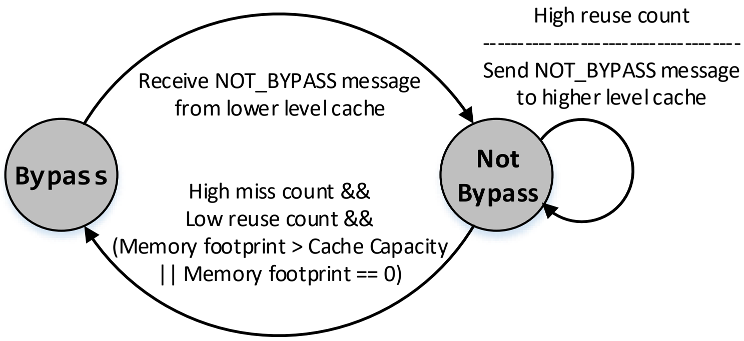

High reuse count
Send NOT_BYPASS message to higher level cache
Receive NOT_BYPASS message from lower level cache
Bypass
Not Bypass
High miss count && Low reuse count && (Memory footprint > Cache Capacity || Memory footprint == 0)

Figure 3.5: State Transition for Cache Bypassing

by the stream engine from the configuration. If not, it sets the memory footprint to 0, and the cache will exclude this information when making bypassing decisions. The cache fills in the corresponding stream table entry when receiving this request. Requests generated by the stream engine contain the stream's `sid`. The cache looks up the stream table to check if it should bypass.

Non-Bypass Stream Requests: If `bypass` is not set, the request is treated as a normal request. The cache updates the stream's dynamic information by: 1. Incrementing the `access` counter. 2. If missed, incrementing the `miss` counter. When the cache line is brought in from the lower level cache, it sets the `sid` field of the tag to the request stream's `sid` so that reuse information can be tracked. 3. If hit, and the `sid` of the tag is valid, incrementing the `reuse` counter of that stream.

Bypass Stream Requests: If the stream is bypassed, i.e. `bypass` is set, the cache will forward the request directly to the lower level cache without waiting for its tag lookup or allocating MSHRs. 1. If missed, the cache forwards the future response from the lower-level cache without caching it. 2. If hit, the cache responds normally and drops the future response from the lower-level cache.

Bypassing Decision: Fig 3.5 shows the FSM making bypassing decisions. The cache reconsiders its decision for a stream when its `access` counter hits a threshold. A stream

Core	2.0GHz 8-Way fetch/issue/commit OoO Cores 64 IQ, 32 LQ, 32 SQ, 192 ROB, 256 Int RF, 256 FP RF
Function Units	6 Int ALU (1 cycle), 2 Int Mult/Div (3/20 cycles) 4 FP ALU (2 cycles), 2 FP Mult/Div (4/12 cycles), 4 SIMD (1 cycle)
Priv. L1 ICache	32KB, 8-way, LRU, 8 MSHRs, 2-cycle latency
Priv. L1 DCache	32KB, 8-way, LRU, 8 MSHRs, 2-cycle latency, LRU
Priv. L2 Cache	256KB, 16-way, 16 MSHRs, 15-cycle latency, LRU
To L3 Bus	16-byte width
Shared L3 Cache	8MB, 8-way, 20 MSHRs, 20-cycle latency
DRAM	2 channel, 1600MHz DDR3 12.8 GB/s

Table 3.2: Simulation Parameters for Baseline

satisfies the condition in Fig 3.5 is marked as bypassed. On the other hand, streams with a high `reuse` count may also have a high reuse rate at the higher-level cache. In such a case, the cache will send a `NOT_BYPASS` message to the higher-level cache to cancel its bypass decision. The LLC never bypasses any stream. The cache also clears the `access`, `miss` and `reuse` counter after reconsidering the bypassing decision. This is to ensure that the stream table captures the changing dynamic behavior of the stream at run time.

3.4 Methodology

Simulation and Compilation For the simulation, we model an out-of-order processor with a modified version of gem5 [45], extended with support for decoupled-stream ISA extensions and the proposed microarchitecture. As described, we use an LLVM-based compiler to identify streams and transform the program. The simulation is carried out with an approach similar to Aladdin [46, 47] and TDG [48, 49], where compiler transforms are applied to a dynamic dependence graph (DDDG) of LLVM IR operations. We generate wrong-path addresses of streams in the DDDG to ensure fair accounting of unused elements.

Common Parameters Table 3.2 summarizes the parameters of the baseline system. We use McPAT [50] for energy estimation, extended to model the stream engine. For the number of pseudo-registers, we choose 24, as it is sufficient to cover most of the hot regions in the benchmarks we simulated.

Baselines/Configurations We compare against the following:

- **Stride Prefetching (Pf-Stride):** In this configuration, we add a PC-based stride prefetcher to all three cache levels. The prefetcher takes 1 cycle to generate the prefetch request and it prefetches for 8 requests ahead.
- **Ideal Helper Thread Prefetching (Pf-Helper):** As discussed earlier, helper-thread approaches [51, 52, 53, 54, 55, 56, 57, 58] are a form of aggressive execution-driven prefetching. We evaluate against an ideal SMT-based helper-thread approach, which consumes no core resources (e.g. ROB, RF)[2]. The helper thread is fixed to run k dynamic instructions ahead of the main thread to prefetch the data. We experimentally found $k = 1000$ is sufficient to bring significant speedup for the main thread.
- **Non-Binding Stream Prefetching (SSP-Non-Bind):** This configuration is a limited version of SSP, where the compiler only recognizes the stream and inserts stream instructions, i.e. `s_cfg`, `s_step` and `s_end`. This configuration only uses the stream engine as a prefetcher and the data fetched is stored in cache. If not specified, we use a 192-entry FIFO for this configuration, with 8 entries per stream. Since most

[2] Properly allocating resources and choosing an instruction slice for a helper thread is the subject of much research, so we abstract here.

streams will have element size less than or equal to 8 bytes, we set the FIFO entry size to 8 bytes. Note that throttling is not possible in SSP-Non-Bind as there are no user instructions.

- **Semi-Binding Stream Prefetching (SSP-Semi-Bind):** This configuration is the same as SSP-Non-Bind except that we use the full decoupled-stream ISA, which has the additional benefits of semi-binding streams and address-computation specialization. If not specified, the dynamic throttling policy from Section 3.3 is enabled.

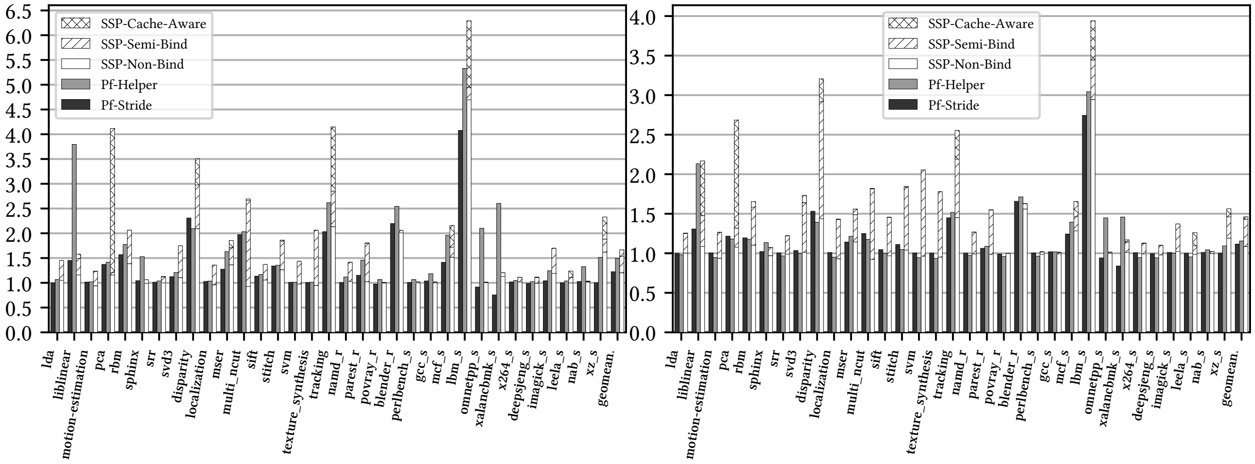
- **Stream-Aware Cache (SSP-Cache-Aware):** This configuration is built upon SSP-Semi-Bind, but with the stream-based cache bypassing policy described in Section 3.3.

We simulate 33 benchmarks from the SPEC CPU 2017 and CortexSuite [35, 36]. We exclude all Fortran benchmarks from SPEC CPU 2017 [34] due to incompatibilities with our current framework. We use the reference input set for SPEC and the largest provided input set for CortexSuite. We use SimPoint [59] to select multiple representative simpoints for simulation from the first 10 billion dynamic instructions. Each simpoint contains 10 million dynamic instructions, and on average 10 simpoints are selected for each benchmark. After cache warm-up, we simulate the simpoints and compute the total execution time and energy based on each simpoint's weight.

3.5 Evaluation

Our evaluation studies the benefits of the three potential opportunities: stream-prefetching, stream-decoupling, and cache-awareness. We first analyze the overall benefit, then discuss each aspect, and end by discussing the integration with different cores.

Overall Benefit Fig 3.6a shows the speedup of all the configurations over the baseline OOO core. Stride prefetching achieves 1.22× speedup, while ideal helper thread yields 1.50× speedup. For SSP, non-binding stream prefetching achieves 1.20× speedup, which is



(a) Overall Speedup

(b) Energy Efficiency

Figure 3.6: Overall Speedup and Energy Efficiency

similar to stride prefetching. Semi-binding stream prefetching achieves 1.53× speedup, which outperforms even the ideal helper thread. The main reason is that semi-binding removes the instruction overhead for address computation. It also does not generate duplicate memory requests to L1. Finally, stream-aware cache results in 1.67× speedup over the baseline OOO core.

Fig 3.6b shows the overall energy efficiency of all the configurations over the baseline OOO core. Stride prefetching improves the energy efficiency by 1.12×, while ideal helper thread achieves 1.16×. Non-binding stream prefetching slightly increases the energy efficiency by 1.09×, while semi-binding stream prefetching gives a significant improvement to 1.47×, as semi-binding removes much instruction overhead. Finally, making the cache stream-aware achieves 1.53× energy efficiency.

Benefits of Semi-Binding Stream Prefetching

The major benefit of semi-binding stream prefetching versus non-binding stream prefetching comes from a combination of removing address computation from the pipeline and reducing traffic to the L1 cache. Fig 3.7 shows the performance of semi-binding stream prefetching,

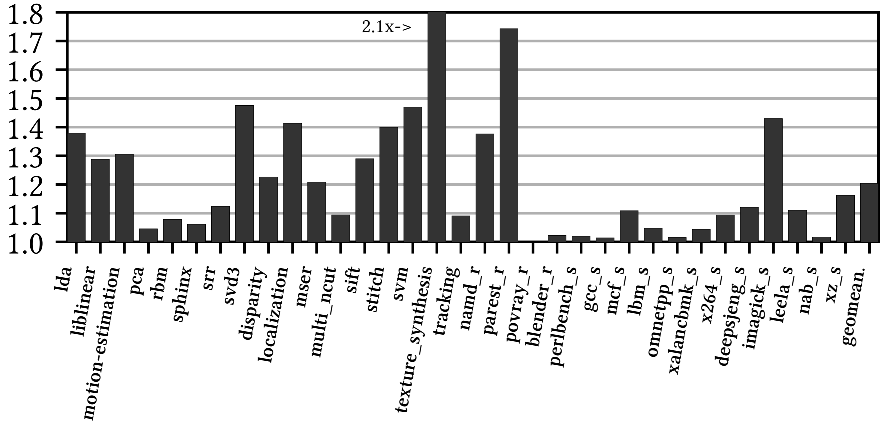


Figure 3.7: Speedup of SSP-Non-Bind. vs. SSP-Semi-Bind

normalized over non-binding stream prefetching. Both configurations use a 192-entry FIFO without throttling. Overall, compared to non-binding prefetching, semi-binding prefetching achieves 1.26× speedup.

Fig 3.8 shows the number of dynamic instructions committed in semi-binding prefetching, normalized to the original program. On average, semi-binding prefetching removes 35% of the dynamic instructions from the original program, while adding back only 5.6% to control the stream engine. Most of the new instructions added are `s_step` instructions that advance the stream FIFO – in most cases one per loop iteration. An extreme case is `svm` from CortexSuite. The hot regions of this benchmark involve small matrix multiplication, which has a small memory footprint. The L1 data cache has less than 1% miss rate, and this explains why neither stride prefetching nor ideal helper thread can improve the performance. When cache is not the bottleneck, semi-binding stream prefetching achieves 1.48× speedup over non-binding stream prefetching for `svm`. A similar analysis also applies to `texture_synthesis`.

Dynamic Throttling Fig 3.9 shows the performance of semi-binding stream prefetching with various FIFO sizes and throttling policies, normalized to the configuration with a 72-entry FIFO, non-throttling configuration. Compared with an evenly distributed policy, dynamic throttling improves the performance the most when the FIFO is small, as it achieves

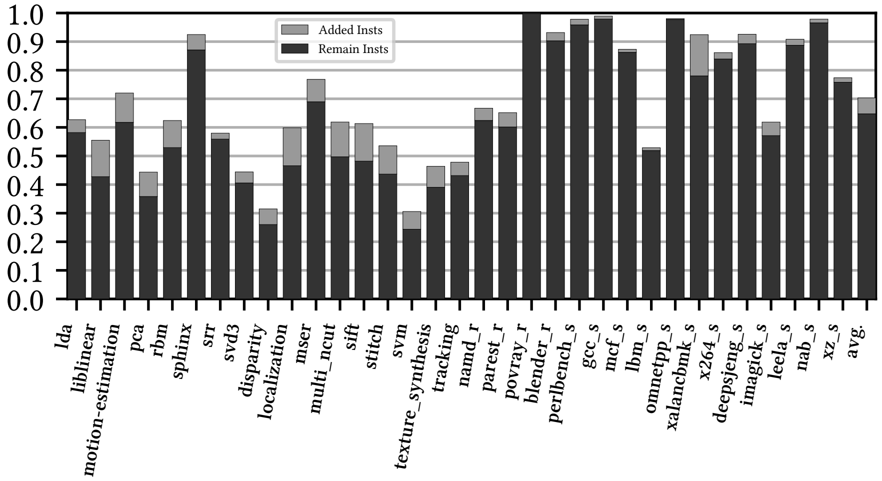


Figure 3.8: Dynamic Instructions in SSP-Semi-Bind

a better utilization for the FIFO by allocating more space to streams lagging behind the core. With a 72-entry FIFO, dynamic throttling improves the performance by 13%, while for a larger 192-entry FIFO, it yields a marginal improvement of 5%.

An extreme case is `multi_ncut`, where most of the execution time is spent on a simple loop that iterates through a matrix and generates sorted indexes. The matrix is too large to be cached in L2, and 68.7% of the memory accesses in this loop go to the L3 cache. Since one stream FIFO entry corresponds to one loop iteration when it is unconditionally stepped, the maximum effective prefetch window measured in dynamic instructions is the number of dynamic instructions per iteration times the FIFO entries allocated for that stream. As the loop body contains only 9 static instructions, the effective prefetch window achieved by a non-throttling policy is not large enough to fully hide the L3 cache latency.

Unused Stream Requests Since we are decoupling the stream pattern from the control flow, the stream elements may be fetched from the cache into the stream FIFO but never used by the core. Fig 3.10 shows the percentage of unused requests issued by the stream engine to the L1 cache in SSP-Semi-Bind. The average unused stream requests is 11.1%. An extreme case is `namd_r`, which has 66% of unused requests. This is partly because some

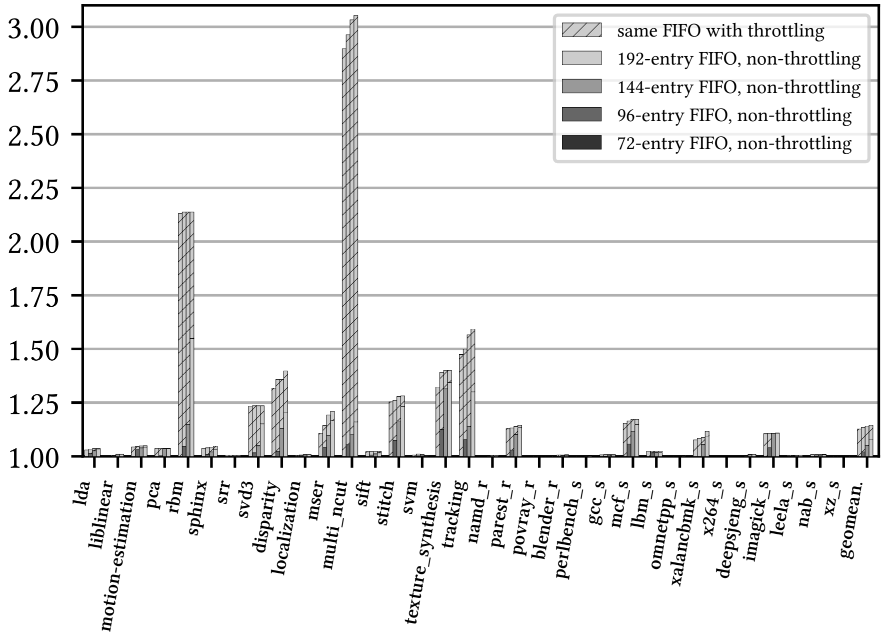


Figure 3.9: Speedup with Dynamic Throttling

stream elements are unused due to control flow, but also because some stream elements are prefetched beyond the termination of the stream (`stream_end`); this is more common for shorter streams. However, these unused stream requests may still be useful as the fetched data may be used by future accesses. It is also possible that the unused stream requests may hit in the L1 cache and not increase the overall memory traffic. The overhead is mainly the extra pressure on the bandwidth between the core and the L1 cache.

Stream-Aware Cache Fig 3.11 shows the performance of stream-aware cache, normalized by the performance of semi-binding stream decoupling. Stream-aware cache supports stream-based bypassing (see Section 3.3). Both configurations use a 192-entry FIFO with dynamic throttling.

Stream-aware cache improves the performance from 1.53× to 1.67× (9%), with the highest peak of 3.4× on the `pca` benchmark. For `pca`, the key kernel (based on our simpoints) is computing the correlation matrix, which contains a 3-level nested loop (i, j, k), and the

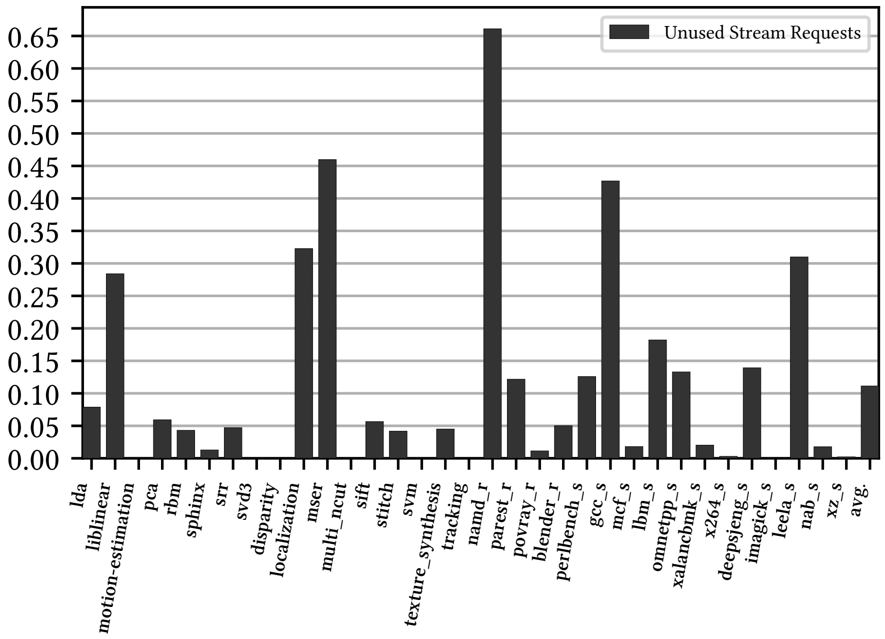


Figure 3.10: Unused Stream Requests

innermost loop accesses two matrix columns ($a[k][i] \times a[k][j]$). The reuse distance is k for the first column and $k \times j$ for the second one. To make things worse, the matrix is accessed in column order, meaning that most data within the cache line goes unused. Without cache awareness, we constantly miss in the L2 cache. Notice that semi-binding stream prefetching here can not effectively hide this latency as we are bound by the L1 cache MSHRs, while the stride prefetcher in the L2 cache does not face this constraint. In a stream-aware cache, the L2 cache bypasses the second column, which releases enough space to fully cache the first column. This increases the L2 cache hit rate and saves bandwidth on the bus to the L3 cache, and leads to 3.4× speedup over semi-binding stream prefetching and 4.1× over the baseline OOO core.

Another case is `lbm_s`, whose memory footprint is too large to be cached in L3 and is memory-bound. In such a case, the stream-aware cache can forward requests to the L3 cache when all MSHRs of the upper-level cache are used. This effectively increases the total number of parallel misses that can be handled by the cache system and improves the performance of

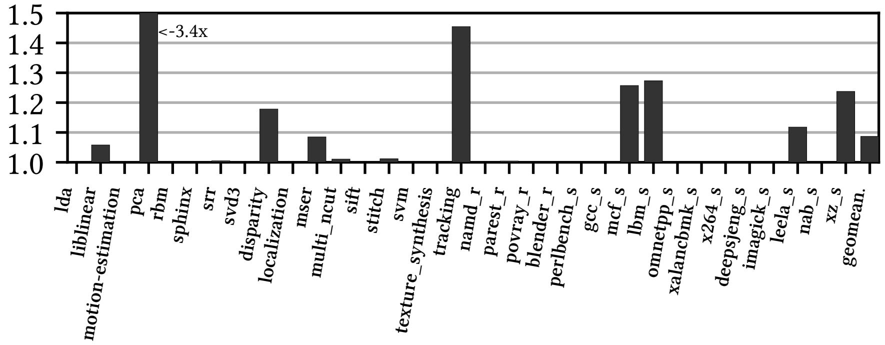


Figure 3.11: Speedup with Cache Awareness

`lbm_s` by 1.3×.

Design Space Interaction To understand the tradeoffs and interaction with different OOO cores, we simulate several configurations from dual-issue up to 8-issue. Fig 3.12 shows the relative speedup and energy efficiency of the baseline OOO processor, stride prefetching, ideal helper thread and SSP with stream-aware cache, normalized to a dual-issue OOO core. Compared with traditional prefetching, stream decoupling can greatly improve both the performance and energy efficiency in both SPEC CPU 2017 and CortexSuite. Notably a 6-issue SSP can surpass an 8-issue OOO in both energy efficiency and performance.

Compared to an ideal helper thread, SSP is much more effective on CortexSuite. This is because most accesses are streams, which can decoupled from the core, and also because SSP can intelligently reason about the cache behavior of streams. SSP only sees similar benefits to the ideal helper thread on SPEC CPU, because its advantage of decoupling is offset by its disadvantage in coverage against non-streaming access.

3.6 Related Work

While the idea of stream-specialized general-purpose processors itself is novel, it derives inspiration from and has an intimate relationship with at least four main areas of architecture

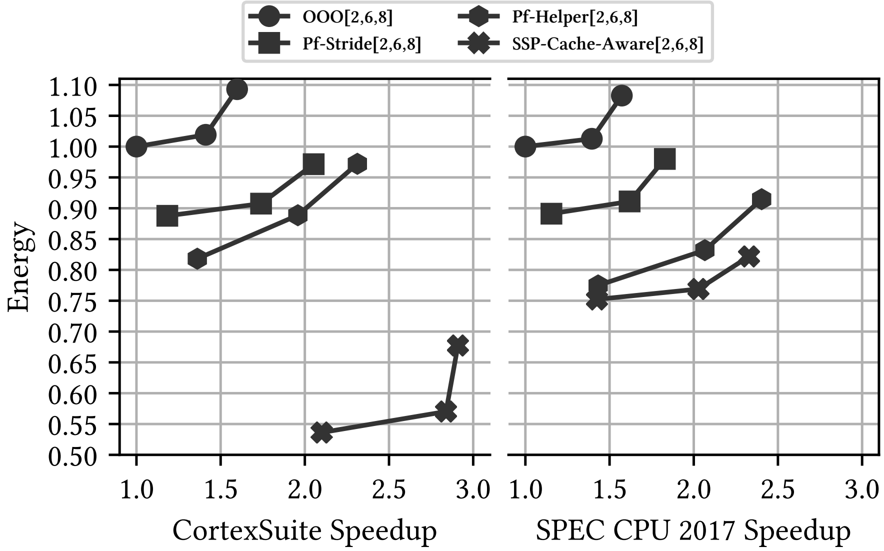


Figure 3.12: Relative Speedup and Energy Efficiency for Various OOO Processors

research: specialization of address generation, decoupled access-execute, prefetching, and cache policy enhancements.

Memory Interface Specialization The concept of exposing patterns of memory access as “streams” within an ISA perhaps originated with the Imagine Stream Processor [60], designed for media processing. Following in their footsteps, a variety of specialized architectures have employed stream abstractions, like RSVP [61], Q100 [62], Softbrain [63], VEAL [64] and CoRAM++ [65]. None of the above target a traditional general-purpose out-of-order core (e.g. no control speculation) or make a general cache stream-aware.

Memory Access Dataflow (MAD) [38] is a reconfigurable front-end/memory-fetch engine for accelerators and SIMD units, but does not use stream abstractions. MAD powers down the OOO core pipeline while it is active, and also does not support exceptions or control speculation. On the other hand, our approach extends the OOO core and does not interfere with its capabilities.

A philosophically similar approach is XMem [66] and the locality descriptor [67], which are cross-layer programming abstractions for conveying memory semantics. The key difference is that our ISA conveys semantics about each access at the instruction level, rather than describing a memory region. This gives our ISA a more fine-grain view of memory patterns.

Decoupled Access Execute (DAE) By encoding and performing streaming memory operations separately from the Von Neumann order of the program, we are implementing a limited form of DAE [41] which is tailored to certain common access patterns. From that perspective, other DAE architectures exploit similar parallelism within programs and can also hide memory latency [68, 69].

One example is Outrider [42], which supports multiple simultaneous decoupled inorder threads; our stream generator supports multiple concurrent streams. DeSC [70] is a recent example that couples an OOO core with either a second OOO core or an accelerator for the computation. DeSC adds compiler/architecture support to break dependencies for certain control-dependent and indirect memory access patterns, which we also address in our work.

In the accelerator space, several designs decouple the datapath, like DySER [71], CCA [72], Chainsaw [73] and ASIC accelerators [74, 75, 76]. However, they are fundamentally limited by the instruction window of the general-purpose core for latency-hiding. A recent work in this space is Buffets [77], which is a storage idiom for decoupled access-execute accelerators, enabling fine-grain synchronization, flexible data reuse and composability.

Runahead/Prefetching Similar to DAE, prefetching also hides memory latency. Stream-specialized processors have an advantage over traditional hardware prefetchers (e.g. stride-based [78] and indirect [79], spatial/temporal memory streaming [80, 81, 82], and irregular correlating prefetchers [83]), in that the data they prefetch is guaranteed to be accurate. Also, the stream-FIFOs can be seen as software-exposed stream-buffers [84, 85], eliminating the overhead of dynamic prediction as well as tag-checking in caches.

The type of prefetching performed with SSP is more similar to software/execution-driven prefetching. For example, The stream-generator can be viewed as a highly-specialized helper thread [51, 52, 53, 54, 55, 56, 57, 58]. Software prefetching [86] also exposes access patterns through the ISA, and some recent proposals are highly programmable [87] and can be compiler-directed [88].

SSP is different in two key ways: There is no redundant address generation, and there is little potential for cache pollution. These are due to SSP's semi-binding prefetch, which eliminates the problems with traditional binding using regular registers (too much register pressure, cannot prefetch under faults or control flow).

Cache-Policy Our cache-policy enhancements are inspired primarily by prior works in cache bypassing, like those based on reuse count [89, 90, 91, 92]. Using the footprint for modifying the cache replacement policy is inspired by the prior cache insertion policy techniques [93, 94], which are designed to dynamically detect behavior that we have available statically in the stream definition. We also combine static and dynamic information about memory accesses for cache bypassing, as was previously explored in the GPU space [95].

3.7 Summary

This chapter explores the concept of leveraging the inherent structure of streams to specialize the core pipeline, cache interface and cache policies. We find that streams can be decoupled, providing a semi-binding interface that does not require stream data to be consumed. Our stream-specialized microarchitecture benefits from the proposed decoupled-stream ISA extension and enables stream-based prefetching, decoupling of address computation, and stream-awareness in prefetch throttling and cache bypassing. Broadly, this paradigm of encoding rich memory access semantics could open up new opportunities for the specialization of communication and computation at even higher levels within the memory hierarchy.

CHAPTER 4

Proactive and Decentralized Stream-Aware Cache Optimizations

While stream-specialized processors leverage explicit stream information *in the core*, the cache hierarchy is still unaware of the stream[1]. In this chapter, we develop extensions to a tiled multicore's memory system to allow decoupled streams to be offloaded into the shared last-level (L3) cache banks. We name our approach *stream floating*, as decoupled streams *float* between L3 cache banks automatically following the access pattern, and proactively generate read requests for the requesting core.

We view stream floating as a new avenue for achieving less request and response traffic, lower effective access latency, and less L3 bandwidth demand. In the remainder of this chapter, we first motivate by discussing overheads in existing reactive caches, and overview how we address these with our three optimizations (§4.1). Following that, we develop the hardware extensions and policies necessary for stream floating (§4.2), as well as coherence considerations (§4.3). Finally, we present methodology (§4.4), evaluation (§4.5), and discuss related work (§4.6).

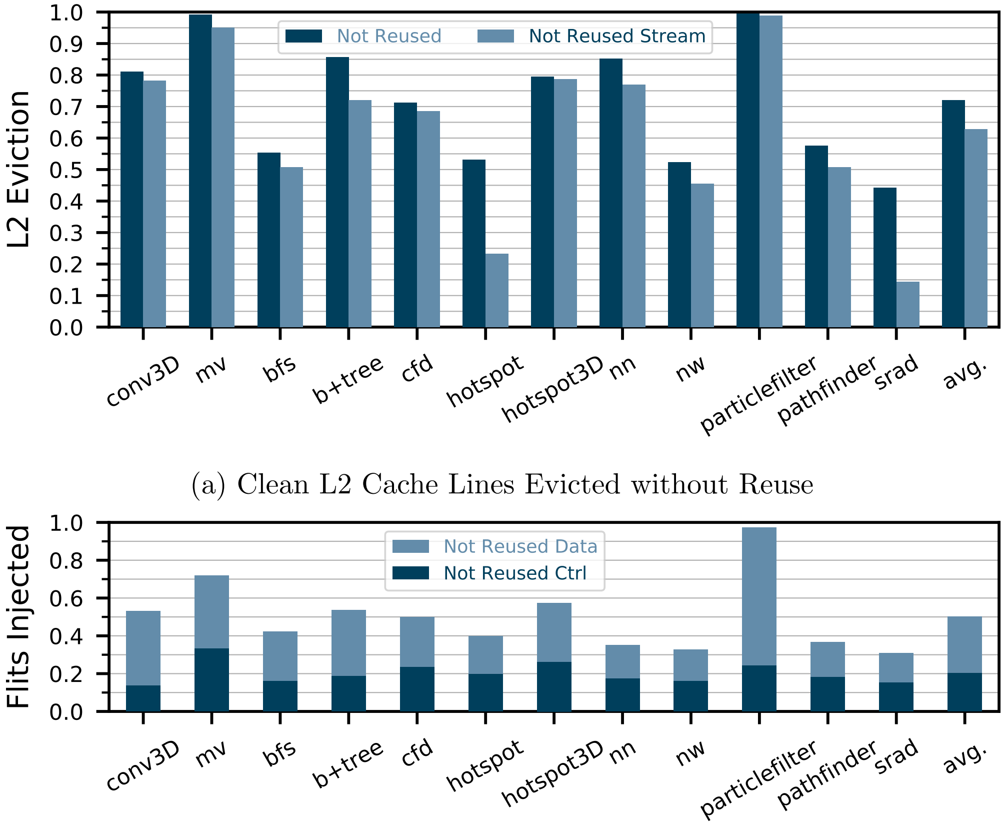
[1] except the stream-aware cache bypassing, in which the bypassing decision is still made by the stream engine in the core.

4.1 Motivation and Overview

As the system scales up, the memory system becomes more and more critical, as it incurs significant overheads to move data around the memory hierarchy and track the coherence state. On the other hand, this also means that there is still a lot of room for optimization, provided that the cache system is *proactive* and *decentralized*. For example, if the remote shared last level cache (henceforth L3) knew the long-term access pattern, it could proactively stream data back to the requesting core without the excessive control traffic. Furthermore, if the system could correctly recognize data without reuse, it could bypass the coherence protocol to avoid expensive invalidation and writeback.

Unfortunately, it is not obvious how to implement such a 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 [28, 96], 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.



(a) Clean L2 Cache Lines Evicted without Reuse

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

Figure 4.1: Overhead of Caching Data without Reuse

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 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 §4.4 for details). The results reveal three major overheads:

- **Cache Thrashing:** Fig 4.1a shows the ratio of L2 cache lines evicted in a clean state without being reused. Overall, 72% of evicted 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. Fig 4.1b 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 caches 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 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]]`. Fig 4.1a 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.

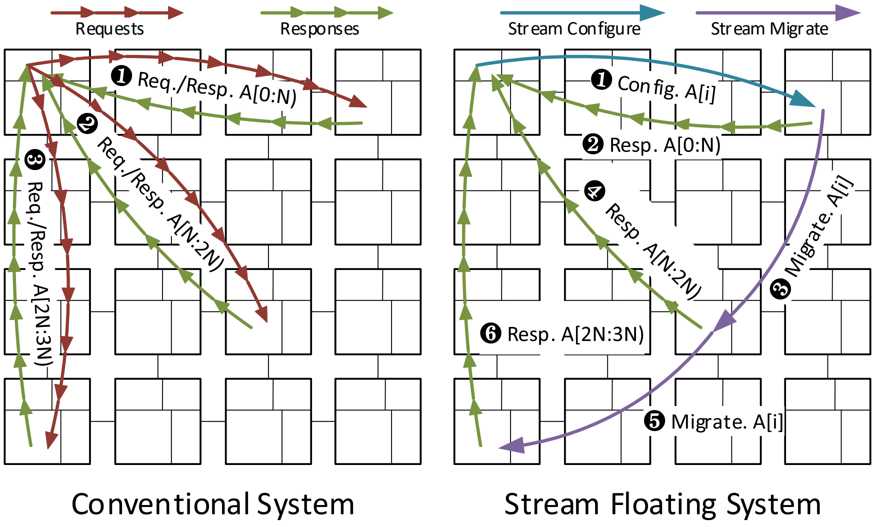


Figure 4.2: Affine Floating Optimization

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 Fig 4.2 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.

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 with a one-time configuration and a few migration messages and the cache proactively prefetches.

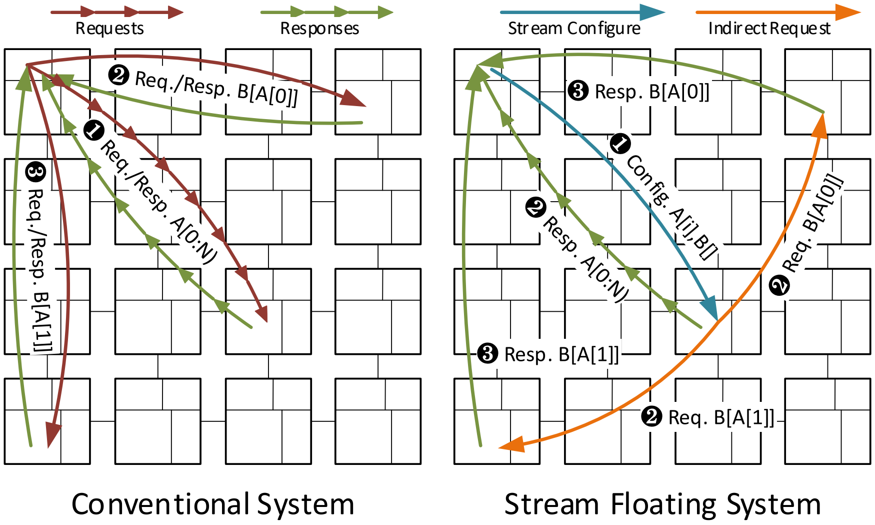


Figure 4.3: Indirect Floating Optimization

Indirect Floating Fig 4.3 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 Fig 4.3). This shortens the chain for indirect accesses.

Stream Confluence In multi-threaded workloads, it is common for different threads to request the same data. However, in existing systems, these accesses are independent of each other, as Fig 4.4 shows. The cache lacks sufficient information to detect and coalesce identical accesses, as 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 compare and coalesce. Streams accessing the same data tend to have the same parameters: e.g. start

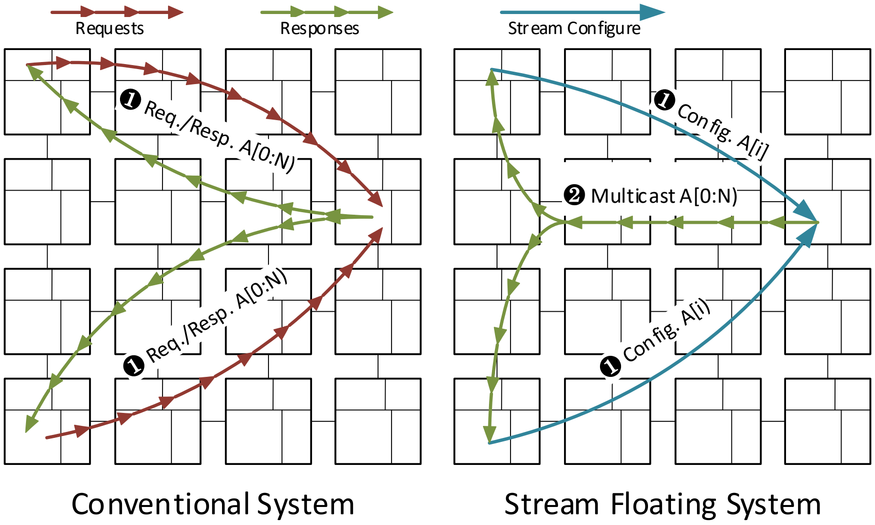


Figure 4.4: Stream Confluence Optimization

address and stride. Also, streams are generally long enough to describe long-term behaviors, which exposes more multicast opportunities. In Fig 4.4, streams accessing the same data can be transparently merged by the cache, turning them into one multicast stream and further reducing the NoC traffic.

Enabling these optimizations to work efficiently means overcoming several challenges that we address in this work. This includes: how to avoid the communication overheads of the 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. Overall, leveraging streams as a coarse grain unit of offloading can empower proactive and more intelligent caches.

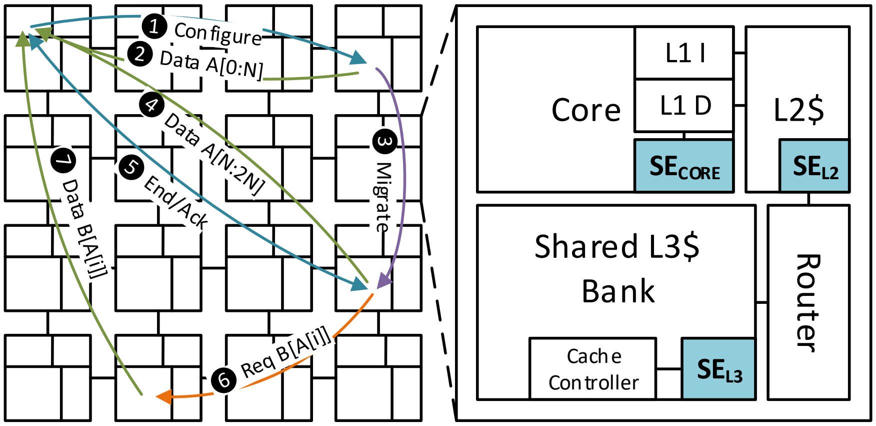


Figure 4.5: Stream Floating Overview

4.2 Stream Floating Design

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

Fig 4.5 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} (Fig 4.7) generates requests and sends stream data back to the requesting SE_{L2}. The requesting SE_{L2} (Fig 4.6) 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.

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 §4.2). If so, SE_{CORE} sends a stream configuration packet to SE_{L2}, containing the hardware context id (same as core id if no

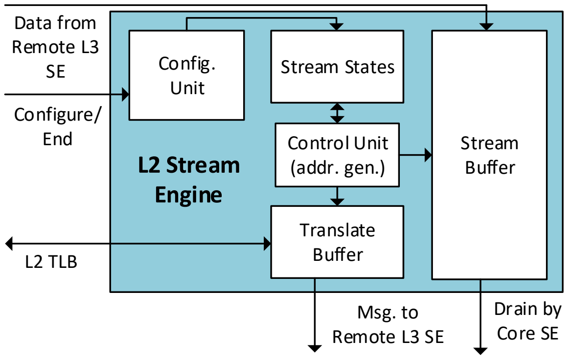


Figure 4.6: L2 Stream Engine (SE_{L2})

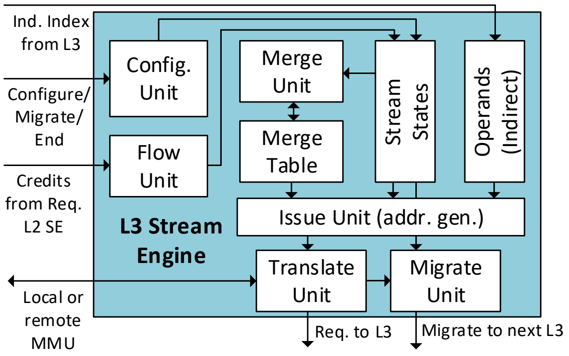


Figure 4.7: L3 Stream Engine (SE_{L3})

SMT), the stream id, and its pattern (i.e. base address, stride, etc.). This is ❶ in Fig 4.5.

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 §4.2). SE_{L2} sends a configuration message over the NoC to the remote L3 bank where the first element is mapped. 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 Fig 4.5). 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 §4.2).

Note that SE_{CORE} still generates requests to prefetch the stream data. These requests are also tagged with the 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$_{\text{L2}}$ will also be notified that the stream request is already served so that it can correctly advance the stream buffer.

Stream Migrate As the stream iterates in the SE$_{\text{L3}}$, eventually, the next element will be mapped to another L3 bank. At this point, the migrate unit 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 Fig 4.5).

Stream End When a stream completes, the SE$_{\text{CORE}}$ constructs a "stream end" packet to terminate the floating stream. The SE$_{\text{L2}}$ uses the last allocated element's address to determine where to forward the packet, and the SE$_{\text{L3}}$ will ack once done (❺ in Fig 4.5). Floating streams with known length can be silently terminated with no stream end packets. Notice that the stream end packet also enables the SE$_{\text{CORE}}$ 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 §4.2).

Coarse-Grained Flow Control Since the stream data is buffered at the SE$_{\text{L2}}$, we need a flow control scheme to synchronize the SE$_{\text{L2}}$ and SE$_{\text{L3}}$ and avoid overwhelming the SE$_{\text{L2}}$'s buffer. We use a credit-based flow control scheme, where the SE$_{\text{L2}}$ sends credits to the SE$_{\text{L3}}$ indicating the vacancy of the stream buffer. These credit messages are handled by the flow unit in Fig 4.7, and the issue unit stalls the stream when running out of credits. This scheme is coarse-grained, as SE$_{\text{L2}}$ only sends out credits when half of the allocated buffer is available; this helps amortize the overhead of flow control messages. SE$_{\text{L2}}$ computes the last allocated element's addresses to determine which L3 bank it should send credits to.

Configuration Size Table 4.1 summarizes the fields in a stream configuration packet. We

	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 4.1: Affine and Indirect Stream Configuration

assume a 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.

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 (Fig 4.7) 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 Fig 4.5).

Supported Patterns The general indirect access pattern is:

$$\underbrace{i_0^{\text{len}_i} j_0^{\text{len}_j} k_0^{\text{len}_k}}_{\text{any order}} w_0^{\text{size}} B[A[i][j][k]+w] \tag{4.1}$$

The i,j, and k iterators can be reordered (by changing the strides in Table 4.1), 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) [97]. If such reuse distance can fit in $\mathrm{SE}_{\mathrm{L2}}$'s buffer (after accommodating other streams), $\mathrm{SE}_{\mathrm{L3}}$ would only send the first K elements of the first stream `A[i]`, and $\mathrm{SE}_{\mathrm{L2}}$ would reuse data from the second stream `A[i+K]` for the following elements. This essentially provides the `A[i]` stream with a prefetch distance of K elements. This approach is compatible with the aliasing detection scheme in 4.2.

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 only need transmit the required portion of the cache line, which can further save network traffic.

Configuration Size Table 4.1 lists the fields of an indirect stream, which are appended to the base affine configuration and require 60 bits per indirect stream.

Stream Confluence

As an optimization, $\mathrm{SE}_{\mathrm{L3}}$ transparently merges the traffic for cores requesting the same streaming data. When adding a stream to the $\mathrm{SE}_{\mathrm{L3}}$ (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 the same address space and parameters, form a confluence group, which is recorded in the merge table. The issue unit records any merged information in the request and the response is multicast to the cores.

Although it is possible to merge streams from any two cores, it increases the hardware

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

Table 4.2: Stream History Table

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 can catch up and form a confluence request.

Policy for Floating and Sinking Streams

The SE_{CORE} decides whether to float a stream by considering both the current pattern as well as historical 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 4.2. 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 that 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.

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).

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 Fig 4.7, 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 [98, 99, 100]. Another option is to use the requesting core's MMUs for translation, as was explored in prior work for accelerators [101]. 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. Fig 4.8 visualizes the life of a floating stream load. Starting backward from commit, there are three windows where aliasing could happen:

After the SE_{CORE} issues the request, the floating load is protected by the PEB and LSQ,

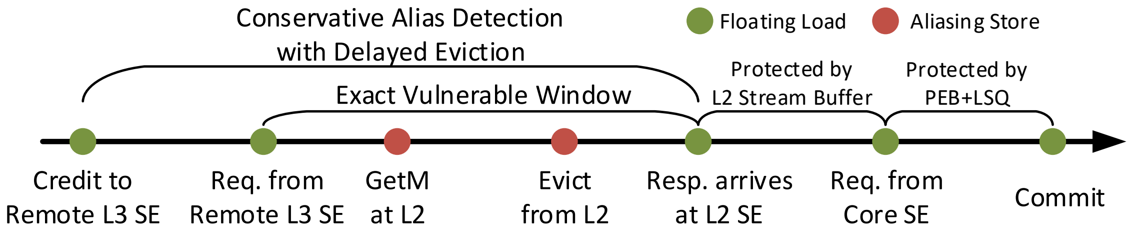


Figure 4.8: Detecting Aliasing to Floating Stream Load

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 the 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.

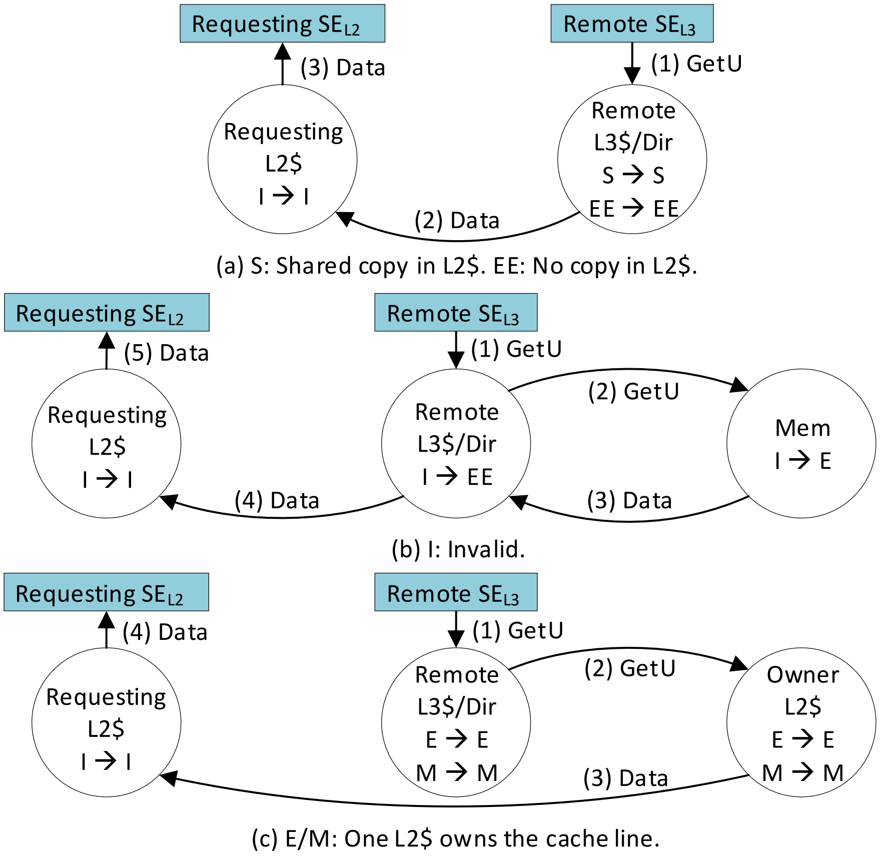


Figure 4.9: Coherence Protocol Interaction

To break the dependence cycle, the SE_{L2} will notify the SE_{CORE} to sink a floating stream if it causes a long delay.

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.

4.3 Coherence and Consistency

As discussed in §4.1, 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.

Our Approach: Uncached Stream Data

Our approach to avoiding clean-eviction traffic is to simply let the stream data reside in $\mathrm{SE_{L2}}$'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 the stream data that is not detected by the directory. Note that this is rare in data-processing workloads, as writes to streaming data are otherwise synchronized. Streams *are* guaranteed to see stores before 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 $\mathrm{SE_{L3}}$, 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. Fig 4.9 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.

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 `s_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 [102, 103, 104, 105, 106], 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[2]. 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 that 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.

[2] Also, the need to prevent cache line evictions for alias detection (see Section 4.2) becomes unnecessary, as the SE_{L3} can inform the requesting core if it is attempting write ownership of stream data (indicates alias misspeculation).

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	64-entry / 8-way
L2/SE_{L3} TLB	2k/1k-entry / 16-way, 8-cycle latency
L1 I/D Cache	32KB / 8-way, 2-cycle latency
Private L2 Cache	256KB / 16-way, 16-cycle latency
L1 Stride Pf.	16 streams, 8 pf. per stream
L1 Bingo Pf.	8kB PHT, 2kB region
L2 Stride Pf.	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 latency, 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 4.3: System and Microarchitecture Parameters

4.4 Methodology

Simulator We extend X86 gem5 [45, 107] with partial AVX-512 support for higher vector width and simulate all cores in execution-driven, cycle-level detail. The CPU is extended with a SE_{CORE} to support decoupled-stream ISA extensions. For the NoC, we use Garnet [108].

Compiler We develop an LLVM-based compiler to generate stream-specialized programs with X86 backend, similar to prior work [28]. One difference is that we use explicit load/store instructions (i.e. `stream_{load|store}`) , instead of pseudo-registers[3] to access stream data.

Benchmarks We simulate 10 OpenMP benchmarks (`-O3` and AVX-512) from Rodinia [109] and two tiled kernels: matrix-vector multiplication (`mv`) and 3d convolution (`conv3d`), as they

[3] 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.

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 4.4: Workload Datasets

are important workloads with stream behavior. Table 4.4 summarizes the parameters.

Systems and Comparison Table 4.3 summarizes the default system parameters including added hardware structures. We use McPAT [50] to estimate the energy at 22nm, and extended to model $\mathrm{SE}_{\mathrm{CORE}}$, $\mathrm{SE}_{\mathrm{L2}}$ and $\mathrm{SE}_{\mathrm{L3}}$.

We choose two different prefetchers to compare against: traditional strided, because they capture the streaming behavior of these workloads, and the Bingo spatial prefetcher [110], because it won 1st place for multi-core prefetching in DPC3 [111] 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 (Fig 4.12).

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 [110], and L2 stride prefetcher.
- **Stream Specialized Processor (SS)**: Stream-specialized core as described in [28]. 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.

4.5 Evaluation

Our evaluation attempts to address two main questions: First is how much potential exists in exploiting streaming patterns to reduce 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.

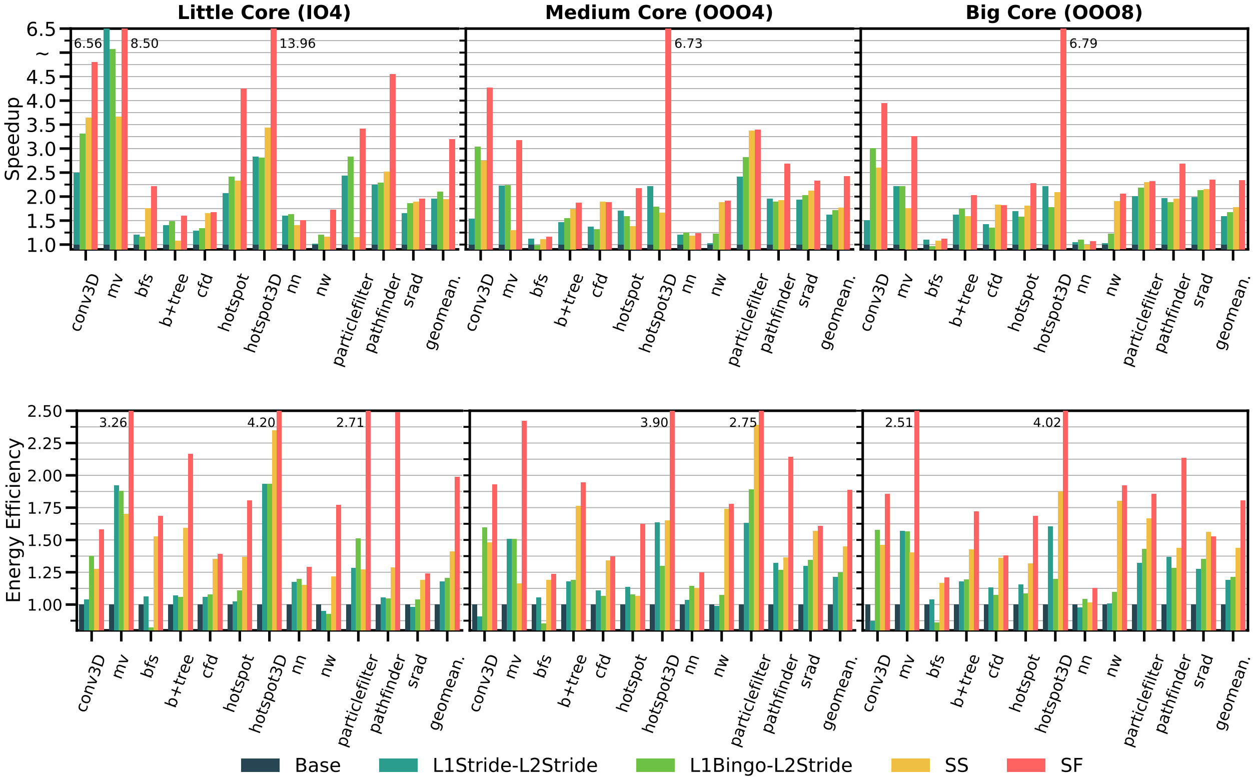


Figure 4.10: Overall Speedup and Energy Efficiency

Overall Speedup, Energy Efficiency and Area

Performance Fig 4.10 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.

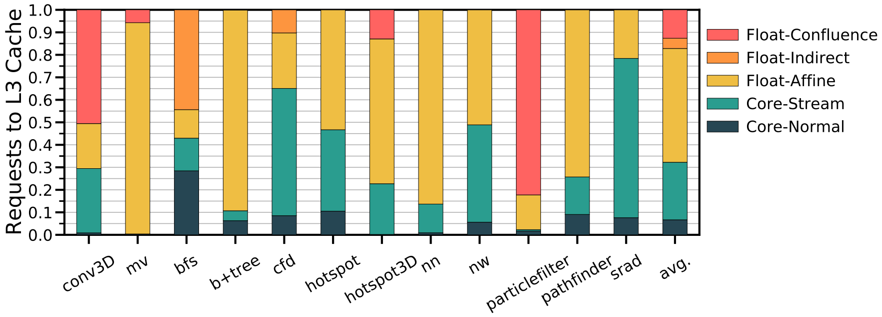


Figure 4.11: Requests to L3 of SF-OOO8

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`). 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 $\mathrm{SE_{L3}}$ 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 a 4-bit stream id and 12-bit sequence number (§4.2). 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 $\mathrm{SE_{CORE}}$).

Floating Requests

Fig 4.11 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 $\mathrm{SE_{L3}}$, 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

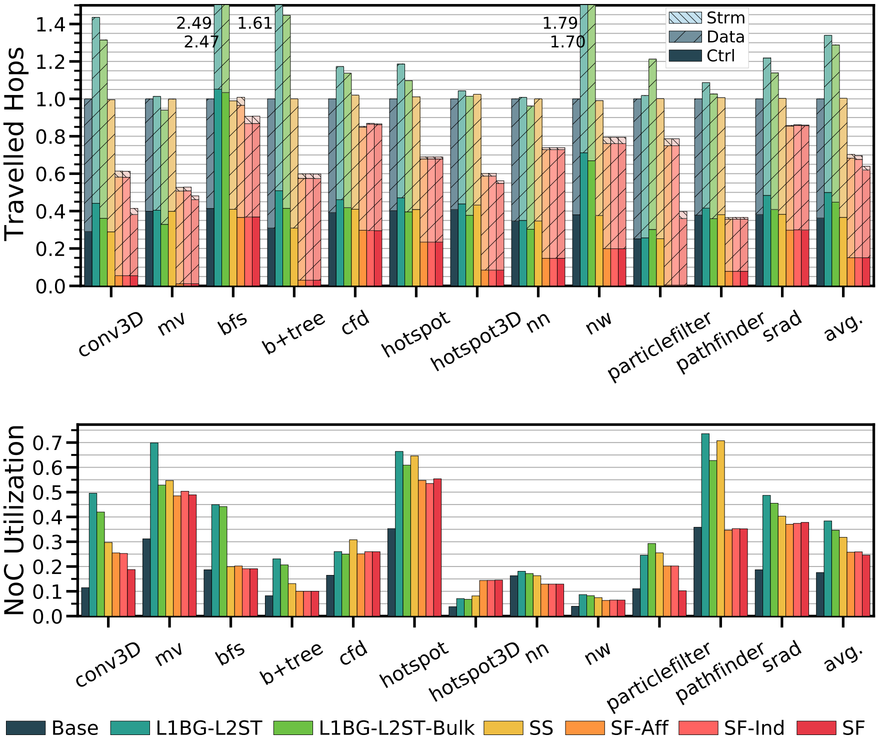


Figure 4.12: OOO8 NoC Traffic and Utilization

(`bfs` and `cfd`). For stream confluence, $\mathrm{SE_{L3}}$ can successfully recognize multicast, e.g. the input feature map in `conv3D` constituting 51% of requests.

Network Traffic

Fig 4.12 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 §4.4, which groups 4 contiguous prefetch requests.

Streams are accurate L1Bingo-L2Stride actually increases the NoC traffic by 34% due to

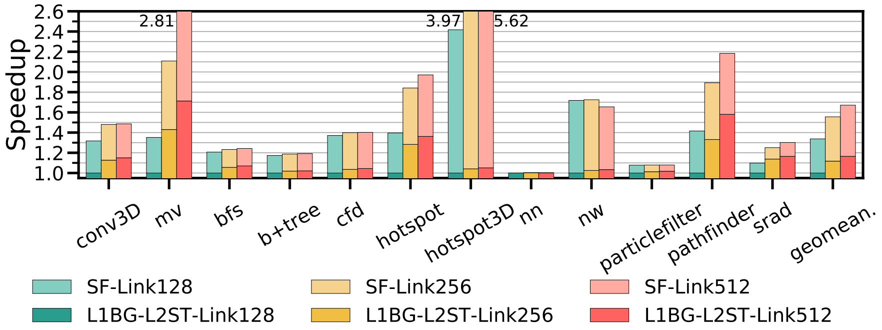


Figure 4.13: SF vs. Bingo with 128, 256, 512-bit link (OOO8)

inaccurate patterns 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 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%.

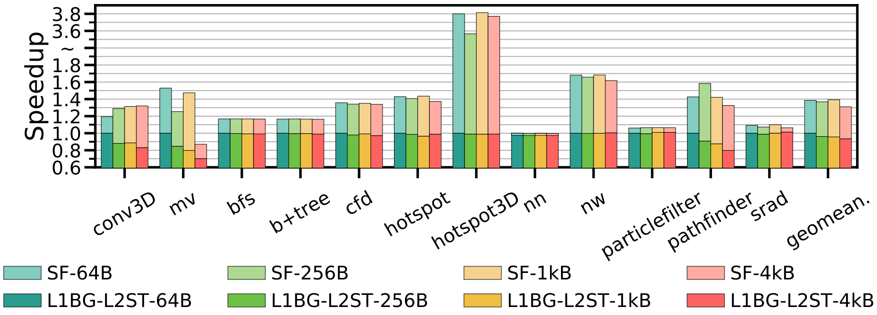


Figure 4.14: Effect of NUCA Interleaving Granularity (OOO8)

Sensitivity to NoC Bandwidth

Fig 4.13 shows the performance of SF and Bingo under different link widths normalized to Bingo with 128-bit links. For some benchmarks, increasing the 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 the link width increased from 128-bit (1.34×) to 512-bit (1.43×). This is because with 512-bit links, data messages are 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.

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. Fig 4.14 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

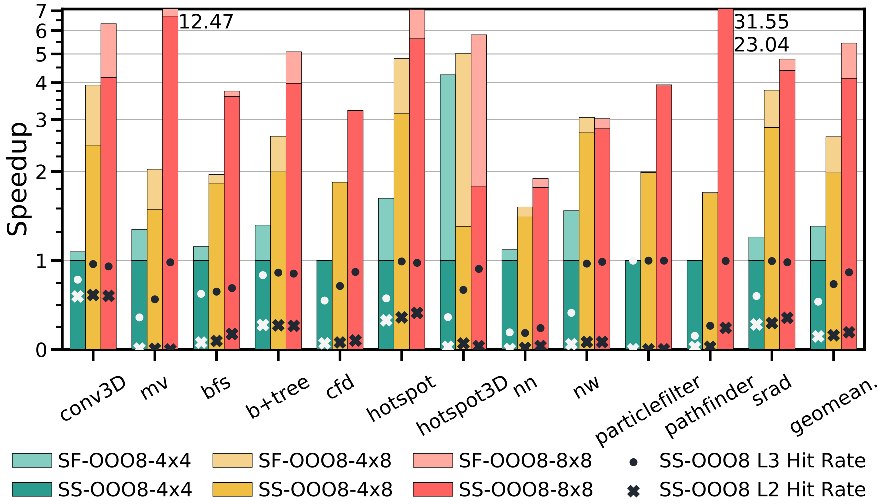


Figure 4.15: Core Scaling

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 reducing 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).

Sensitivity to Core Scaling

Fig 4.15 shows SF's speedup over SS with varying core counts, normalized to SS-4x4. Dots indicate L2 and L3 hit rates 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 for reused data. However, when the data cannot be cached on-chip, SF suffers from the same

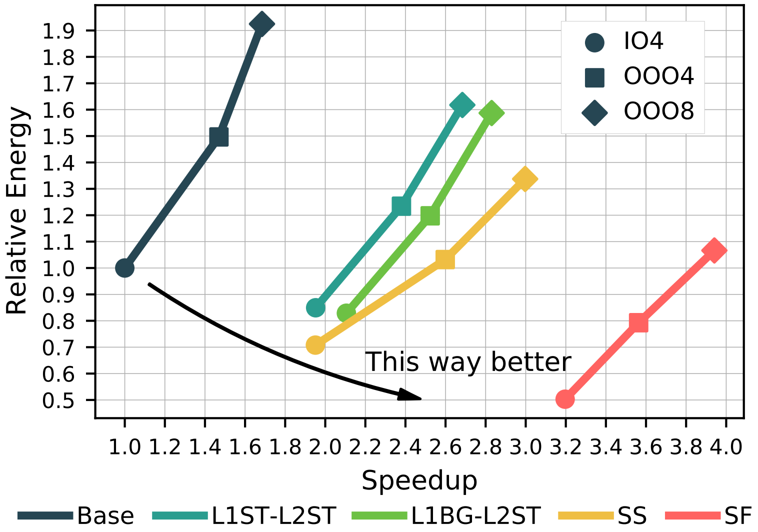


Figure 4.16: Energy vs. Speedup for IO4, OOO4, OOO8

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×).

Fig 4.16 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.

4.6 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 [14], which enables offloading remote memory instructions. Active Routing [16] 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 [4] leverages the idle time of the NoC router to perform computation. A centralized packet manager distributes computation to routers, which chains computation by forwarding. Livia [27] 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 [63, 28, 112, 113, 114]) 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.

EMC [115] augments a 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 [3].

Cache Policy Optimizations Dead-block techniques predict which cache lines have little reuse, and help quickly replace or avoid caching them [116, 117, 89, 118, 94]. Other bypassing techniques follow similar principles [90, 91, 92, 119, 95]). It is future work to selectively decide whether to bypass or allow floating stream data to enter the cache. Similarly, several cache replacement policies avoid thrashing by initially assuming little reuse [120, 121, 122].

Another body of work recognizes data-sharing patterns to simplify coherence operations (e.g. producer-consumer [123, 124], migratory [125, 126], false-sharing [127]), and also to enable forwarding to hide latency [128, 129, 130]. 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 This work is heavily inspired by many prior prefetching works. E.g. feedback directed prefetching [131] monitors prefetching usefulness, lateness and pollution to throttle the prefetcher; we also monitors the usefulness of floating and throttles based on timing.

Specifically for indirect (data-dependent) prefetching, prior work explored software techniques [132] and hardware techniques like IMP [79] and CATCH [133]. Our approach identifies similar patterns. An even more general approach is the event-triggered prefetcher [134], which allows specialized prefetching hardware to run simple programs that can respond to prefetch events. Our prefetcher is less general, focusing on common forms.

Buffets [77] 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.

4.7 Summary

This chapter 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, we believe that the concept of exposing higher-level abstractions like streams can help to enable new memory system optimizations, especially near-data computing.

CHAPTER 5

Near-Stream Computing

While streams mainly describe memory access behaviors, they can also be extended to include the computation producing or consuming the stream values. These computations now apply to the entire stream (subjected to predication). This enables a new paradigm, “near-stream computing”, where computations are flexibly scheduled along with the associated stream near the data.

In this chapter, we first make a case for near-stream computing in §5.1, and also overview our optimizations and compare against prior near-data works. Then we discuss the basic in-core operation in §5.2. We describe our offloading approach and range-based synchronization in §5.3, while §5.4 discusses programmer-exposed synchronization-free optimizations. We cover methodology and evaluation in §5.5 and §5.6, with additional related work in §5.7.

5.1 Motivation and Overview

We first discuss general tradeoffs for general near-data computing abstractions. Then, we introduce a taxonomy of sub-thread near-data patterns and the opportunity they provide. Finally, we overview our approach and compare it to other sub-thread near-data techniques.

Why Sub-Thread Near-Data Computing?

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 with efficient pipelines. This broad paradigm of near-data computing (NDC) includes near-memory techniques [135, 136, 137, 138, 139, 140, 15, 141, 142, 143, 144, 145, 146], as well as near-cache [27, 147, 148, 21, 20, 149, 3]; 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 [143, 12, 11]. 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 [14, 3, 141] or short program regions [16, 27, 4].

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 structures (arrays, lists, trees, etc.) and flexibly employ various computation strategies (near-data filtering, stores, atomics, reductions, etc.). However, prior works only support a subset [14, 3, 16, 27, 4, 141].

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 [14, 3]; 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 [16, 27, 4], 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 the detection of memory ordering violations using per-data-structure access summaries rather than individual accesses.

Sub-Thread NDC 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.

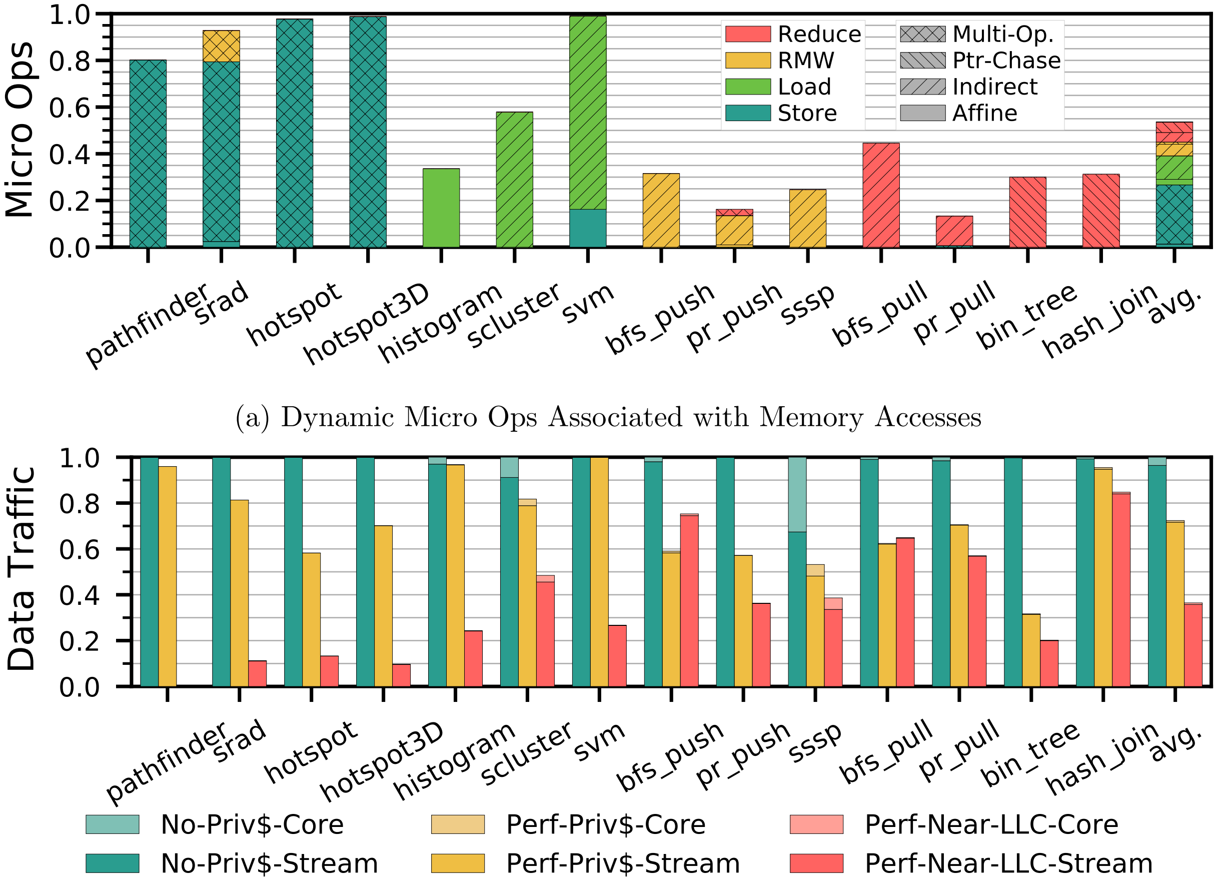
When such an address pattern is decoupled from the remainder of the program, we refer to it as a **stream**.

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 bit width 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 §5.5/§2.4 for workload/compiler details). Fig 5.1(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. Fig 5.1(b) shows the pure data traffic (bytes × 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%.



(a) Dynamic Micro Ops Associated with Memory Accesses

(b) Data Traffic Reduction after Offloading Computation

Figure 5.1: Potential of Sub-Thread Near-Data (View in Color)

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 Fig 5.2(a) shows the case of affine reduction (Σ`A[i]`). Conventional systems fetch all the data to accumulate the result (multiple request/response arrows), introducing

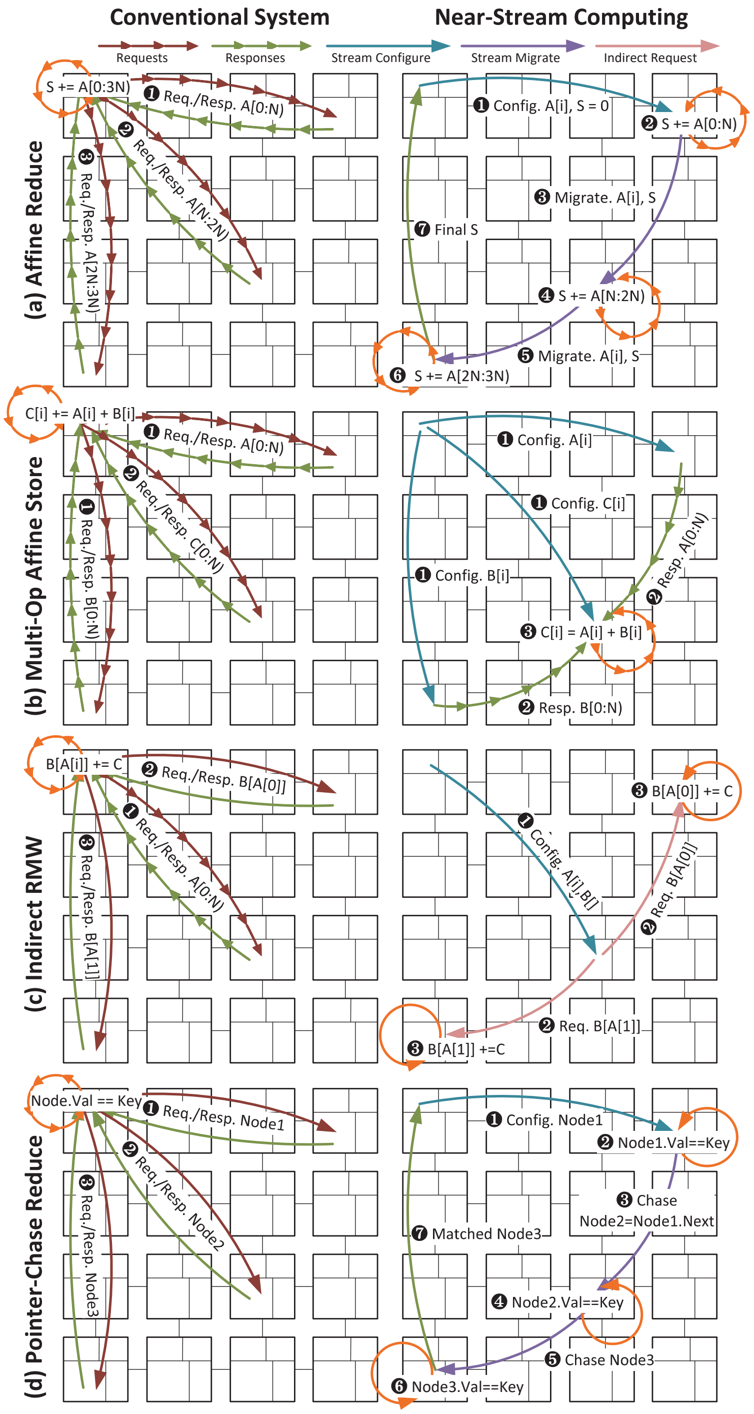


Figure 5.2: Near-Stream Computing Optimizations

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 writing 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 computing.

RMW RMW streams (e.g. `A[i]+=C`) are a hybrid case of both load and store computation. Semantically, they guarantee the 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. Fig 5.2(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.

Access Pattern: Indirection Computations can also be associated with indirect streams. Fig 5.2(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.

	Active Rtng [16]	Livia [27]	Omni- Comp. [3]	Snack -NoC [4]	PIM- En. [14]	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 5.2)	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 5.1: Capabilities of Sub-thread Near-data Approaches

Access Pattern: Pointer-Chasing Fig 5.2(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 to the core.

Unlike affine streams, the bank for 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 encountered 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.

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 5.1 summarizes comparison, and Table 5.2 compares supported address and compute patterns to prior works. We explain in detail below.

Active Routing [16] 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 [27] 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 [4] 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 [14] 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 5.2).

Omni-Compute [3] 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.

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 [28,

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

$\mathbb{A}$: Active Routing, $\mathbb{L}$: Livia, $\mathbb{O}$:Omni, $\mathbb{S}$: Snack-NoC, $\mathbb{P}$: PIM-en, $\mathbb{N}$: Near-stream

$\underline{\text{Underline}}$ indicates partial support through fine-grain offloading (high overhead).

Table 5.2: Address and Compute Patterns of Near-Data Works

32, 96, 150, 151], primarily in the context of prefetching. Table 5.3 compares their capabilities to generate various access patterns. Note that Prodigy [151] uses a different terminology of *Data Indirection Graph (DIG)* instead of *stream* dependencies.

	Addr. Pattern	Near-Data Compute?
Stream-Specialized Processor [28]	Affine, Indirect, Ptr.	No
Stream-Semantic Register [96]	Affine	No
Unlimited Vector Extension [150]	Affine, Indirect	No
Prodigy [151]	Affine, Indirect	No
Stream Floating [32]	Affine, Indirect, Ptr.	Address Only
Near-Stream Computing (this work)	Affine, Indirect, Ptr.	Addr. + Comp.

Table 5.3: Capabilities of Stream ISA Works

Unlike prior works focusing on address generation, this work extends streams with a new dimension: computation. With compiler and ISA support (see §5.2), 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 §5.3). When synchronization is not required (through explicit pragmas), near-stream computing introduces new aggressive optimizations (§5.4), e.g. embedding inner loop streams in outer-loop streams and completely removing the inner loop, which is not supported in SSP [28] or stream-floating [32].

5.2 In-Core Near-Stream Computing

Here we discuss the preliminary microarchitecture components of a near-stream system to enable in-core execution only (i.e. *not* offloading near-data). 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 computations. However, many important workloads require a vector unit that would be inefficient to replicate. Instead, our approach is to use light-weight thread contexts for executing more general near-stream computation.

The **stream computing manager (SCM)** manages the execution of near-stream function, 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) [152]. SCCs are lightweight, as near-stream instructions do not contain loads/stores and do not incur long 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 triggering a new computation. The SCM is responsible for scheduling 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 [28]). The SE configures the SCM with the function pointer and any loop invariant operands for new near-stream functions. 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 the stream-specialized processor, a prefetch element buffer (PEB) is added for memory disambiguation of prefetched data before it is ordered by core memory access instructions [28]; 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) [28].

Stream-floating described in Chapter 4 [32] is an alternate near-data approach that offloads *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.

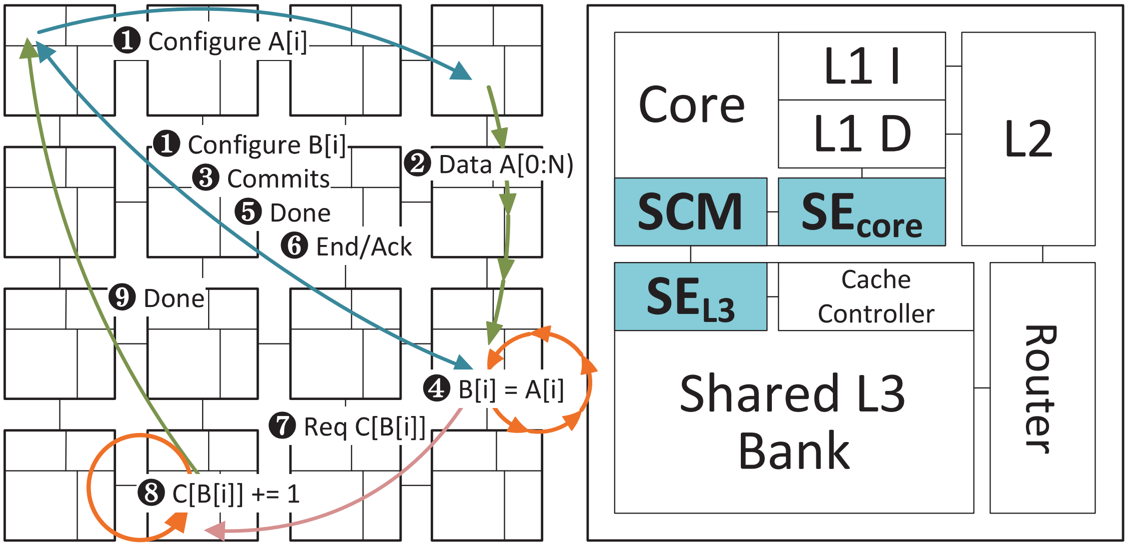


Figure 5.3: In-Cache Near-Stream Computing Overview

5.3 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.

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 to a limited range. In this work, we will further assume this observation extends to physical address ranges, due to the

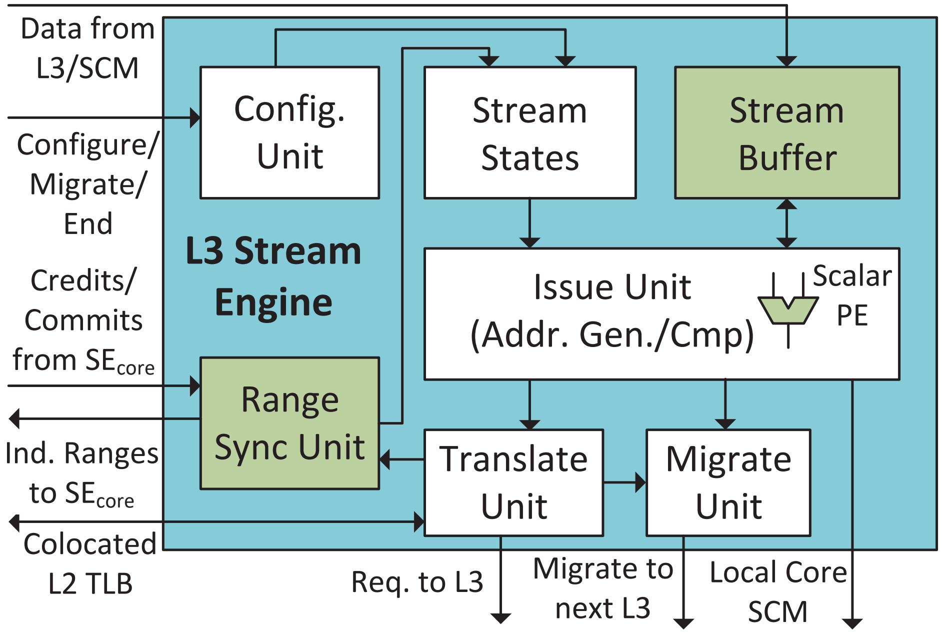


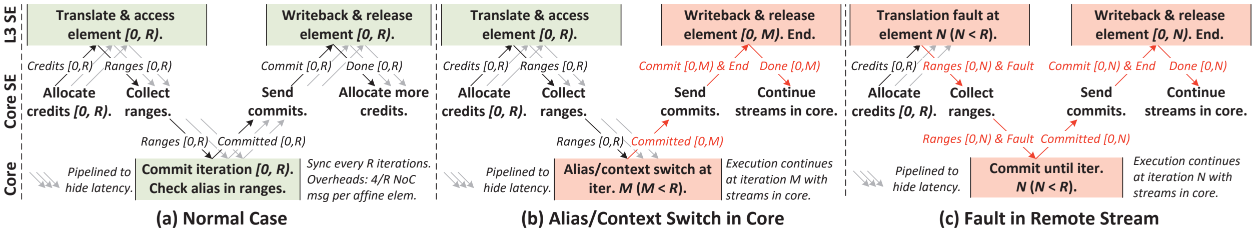
Figure 5.4: L3 Stream Engine (SE$_{L3}$)

use of large pages or the OS's support for transparently promoting continuous pages into huge pages to reduce fragmentation [153]. 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[1].

Proposed System Overview Fig 5.3 shows our proposed system. Besides the core stream engine (SE$_{CORE}$), we add an analogous SE to shared L3 banks (SE$_{L3}$) (Fig 5.4). The tile where the stream is offloaded is called the "remote" tile.

Near-stream operation begins when the SE$_{CORE}$ decides whether offloading would be profitable, and sends the request to the remote SE$_{L3}$ (Fig 5.4), which requests the stream

[1] Larger but more accurate approximation could also be used to reduce false positives, e.g. bloom filter used in BulkSC [154], and this would not require per-data structure physical address contiguity.


L3 SE
Core SE
Core
Translate & access element [0, R).
Writeback & release element [0, R).
Credits [0,R)
Ranges [0,R)
Commit [0,R)
Done [0,R)
Allocate credits [0, R).
Collect ranges.
Send commits.
Allocate more credits.
Ranges [0,R)
Committed [0,R)
Pipelined to hide latency.
Commit iteration [0, R). Check alias in ranges.
Sync every R iterations. Overheads: 4/R NoC msg per affine elem.
Translate & access element [0, R).
Writeback & release element [0, M). End.
Credits [0,R)
Ranges [0,R)
Commit [0,M) & End
Done [0,M)
Allocate credits [0, R).
Collect ranges.
Send commits.
Continue streams in core.
Ranges [0,R)
Committed [0,M)
Pipelined to hide latency.
Alias/context switch at iter. M (M < R).
Execution continues at iteration M with streams in core.
Translation fault at element N (N < R).
Writeback & release element [0, N). End.
Credits [0,R)
Ranges [0,N) & Fault
Commit [0,N) & End
Done [0,N)
Allocate credits [0, R).
Collect ranges.
Send commits.
Continue streams in core.
Ranges [0,N) & Fault
Committed [0,N)
Pipelined to hide latency.
Commit until iter. N (N < R).
Execution continues at iteration N with streams in core.

(a) Normal Case

(b) Alias/Context Switch in Core

(c) Fault in Remote Stream

Figure 5.5: Timeline of Range-Synchronization

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.

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, an 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 aliases. Remote streams' progress is either written back after the core commits the corresponding iteration without detecting aliases, 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 Fig 5.4). 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 the 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 [32]). 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 and reuse rate in the private cache as well as whether it has aliased with other streams or core accesses. Only streams with a 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[]` → `B[]`) to determine its current location. When received, SE_{L3} starts to generate stream requests and schedule computations (Fig 5.3 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 (Fig 5.3 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 $\textbf{SE}_{\textbf{L3}}$ 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 $\mathrm{SE_{L3}}$ 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. Fig 5.5(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, $\mathrm{SE_{CORE}}$ sends credits to $\mathrm{SE_{L3}}$, 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), $\mathrm{SE_{CORE}}$ 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 aliases and terminates the offloaded streams if it finds aliasing.

If there is no aliasing, $\mathrm{SE_{CORE}}$ sends commit messages to $\mathrm{SE_{L3}}$ for store and RMW streams; only then can streams write back to the cache. Subsequently, $\mathrm{SE_{L3}}$ will reply to $\mathrm{SE_{CORE}}$ with a "done" message, so that $\mathrm{SE_{CORE}}$ can allocate more credits (Fig 5.3 ❸-❺).

When $\mathrm{SE_{CORE}}$ detects an alias involving offloaded streams (e.g. a false positive due to the conservative range check), or when a context switch or exception happens, $\mathrm{SE_{CORE}}$ issues an end message to the remote $\mathrm{SE_{L3}}$ to write back committed iterations and release the stream (Fig 5.5(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 Fig 5.5(c).

Stream Migrate & End Similar to [32], streams automatically migrate to the next L3 bank as necessary due to address interleaving. To terminate a stream, SE$_{\text{CORE}}$ sends out an end message to SE$_{\text{L3}}$ (Fig 5.3 ❻). Streams with known length can be silently released in SE$_{\text{L3}}$ (after committing all work).

Coherence & Consistency SE$_{\text{L3}}$ issues requests to the L3 controller to collect and write back the 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 §5.3).

At the instruction level, near-data streams only support weak consistency, as remote stores/atomics are written out of order (serializing stores [155] is possible, but reduces near-data benefits). It is the compiler's 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$_{\text{L3}}$ among cores to avoid sharing. SE$_{\text{CORE}}$ keeps track of resource utilization and may pause issuing credits to avoid possible deadlocks. Another approach is to let SE$_{\text{L3}}$ dynamically allocate resources among streams, and have the SE$_{\text{CORE}}$ 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.

Efficient Indirection Support

Indirect computation can be offloaded along with the affine stream. Fig 5.3 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$_{\text{L3}}$, and reply to SE$_{\text{CORE}}$ (Fig 5.3 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 are 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 seen iteration 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 the precise state (similar to Fig 5.5(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

	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.			
Cmp.	`type`	4	Compute type.	`fptr`	48	Func pointer.
	`sid`	4	Arg. sid (8×).	`size`	3	Arg. size 2^n (8×).
	`ret`	3	Ret. size 2^n.	`data`		Const. arg.

Table 5.4: Near-Stream Computing Configuration

overheads.

To break the recursive dependence, we limit indirect reduction to associative operations, e.g. +, ×. When offloaded, partial results are reduced in each visited indirect banks, 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× # of banks) to avoid overheads of short indirect reduction.

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.

Stream Encoding

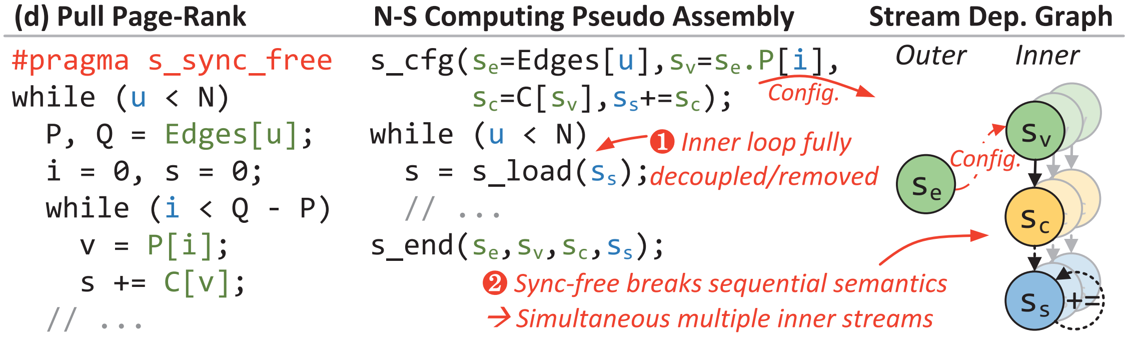
Table 5.4 lists fields of a stream configuration, separated as the access pattern and possible associated computation. We support up to 3-dimension affine patterns. For near-stream computing, we encode simple scalar computations directly in `type`, e.g. +, ×, RMW, etc., which can be executed by the ALU 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.

5.4 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 states 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 (Fig 5.6), indicating that streams in this region never alias. When offloaded, such streams 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 committing ahead of offloaded streams to avoid complete desynchronizing. This eliminates some control overhead, and importantly, shortens the dependence chain.


(d) Pull Page-Rank

```
#pragma s_sync_free
while (u < N)
  P, Q = Edges[u];
  i = 0, s = 0;
  while (i < Q - P)
    v = P[i];
    s += C[v];
  // ...
```

N-S Computing Pseudo Assembly

```
s_cfg(s_e=Edges[u],s_v=s_e.P[i],
      s_c=C[s_v],s_s+=s_c);
while (u < N)
  s = s_load(s_s);
  // ...
s_end(s_e,s_v,s_c,s_s);
```

Config.

❶ Inner loop fully decoupled/removed

❷ Sync-free breaks sequential semantics
→ Simultaneous multiple inner streams

Figure 5.6: Fully Decoupled Loop (Same Legend as Fig 2.5)

Coarse-Grain Context Switch Without synchronization, streams are free to commit until there are no remaining credits. Therefore, once the credits are sent out to the $\mathrm{SE}_{\mathrm{L3}}$, there is no sequential point in the original program order. However, a coarse-grain context switch is still possible by stopping credit issuing and collecting all the done messages. Offloaded streams' progress is 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 Fig 5.6, 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 $\mathrm{SE}_{\mathrm{CORE}}$, further reducing core instruction overhead.

More importantly, now $\mathrm{SE}_{\mathrm{CORE}}$ can *simultaneously* advance multiple instances of fully decoupled nested streams, increasing potential parallelism (3 concurrent instances in Fig 5.6).

5.5 Methodology

Evaluation Stack We use gem5-20 [107] for execution-driven, cycle-accurate simulation, extended with partial AVX-512 support, with Garnet [108] for the NoC and DRAMsim3 [156] for DDR4. We implement an LLVM-based compiler with x86 backed to recognize streams and associated computations as described in §2.4.

Benchmarks We simulate 14 OpenMP workloads from Rodinia [109], MineBench [157] and Gap Graph Suite [158], covering both affine patterns and irregular accesses (see Table 5.6). `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 5.5 lists system parameters. Energy consumption is estimated using McPAT [50] 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 [153].

The baseline core's L1 uses the Bingo [110] spatial prefetcher, the best multi-core prefetcher in DPC3 [111]. 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 [3]). Reduction cannot be supported due to fine-grained offloading.

- **Single-Line NDC (SINGLE)**: Single cache line accessing functions are offloaded to the cache, and may be chained to support pointer-chasing patterns. This resembles Livia [27] and has sync-free optimizations in §5.4 (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.

- **In-core Streams ($\mathbf{NS_{core}}$)**: $\mathrm{SE_{CORE}}$ is only used as an prefetcher (similar to SSP [28]).

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	64-entry, 8-way
L2/SE_{L3} TLB	2k/1k-entry, 16-way, 8-cycle latency
L1 I/D Cache	32KB, 8-way, 2-cycle latency
Private L2 Cache	256KB, 16-way, 16-cycle latency
Replace Policy	Bimodal RRIP, $p = 0.03$
L1 Bingo Pf.	8kB PHT, 2kB region
L2 Stride Pf.	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 latency, 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 latency to SCM
Stream Buf.	16kB FIFO
SE_{L3}	12 streams per core, 768 total 64kB stream buffer, 1kB per core 4-cycle latency to local SCM

Table 5.5: System and Microarchitecture Parameters

- **Address-only Near-Stream ($\mathbf{NS_{no\ comp}}$)**: Streams may be offloaded but *without offloading computation* (similar to Stream Floating [32])

- **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 §5.3.

- **Synchronization-Free Optimizations**: $\mathbf{NS_{no\ sync}}$ turns off range-sync as programmers guarantee alias-free. $\mathbf{NS_{decouple}}$ 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

Benchmark	Addr. Cmp	Parameters
pathfinder [109]	Multi-Operand Store	1.5M entries, 8 iters
srad [109]	Multi-Operand Store	1k×2k, 8 iters
hotspot [109]	Multi-Operand Store	2k×1k, 8 iters
hotspot3D [109]	Multi-Operand Store	256×1k×8, 8 iters
histogram	Affine Load	12M 32b value, 8b key
scluster [109]	Indirect Load	768k×64B, 5 iters
svm [157]	Indirect Load	384k×64B, 2 iters
bfs_push [158]	Indirect Atomic	Kronecker generated
pr_push [158]	Indirect Atomic	256k nodes
sssp [158]	Indirect Atomic	3.6M edges
bfs_pull [158]	Indirect Reduce	A/B/C: 0.57/0.19/0.19
pr_pull [158]	Indirect Reduce	weight [1,255]
bin_tree	Pointer-Chasing Reduce	128k nodes, 8B key, 512k uniform lookups
hash_join	Pointer-Chasing Reduce	8B key, 256k $\bowtie$ 512k, Hit Rate 1/8

Table 5.6: Workloads (MO: Multi-Op)

for $\text{NS}_{\text{decouple}}$ is SINGLE (both programmer-exposed).

5.6 Evaluation

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

Overall Performance/Energy/Area

Fig 5.7 presents the speedup relative to the baseline OOO8 core. Near-stream computing (NS) significantly outperforms prior prefetching and near-data techniques, achieving 3.19×

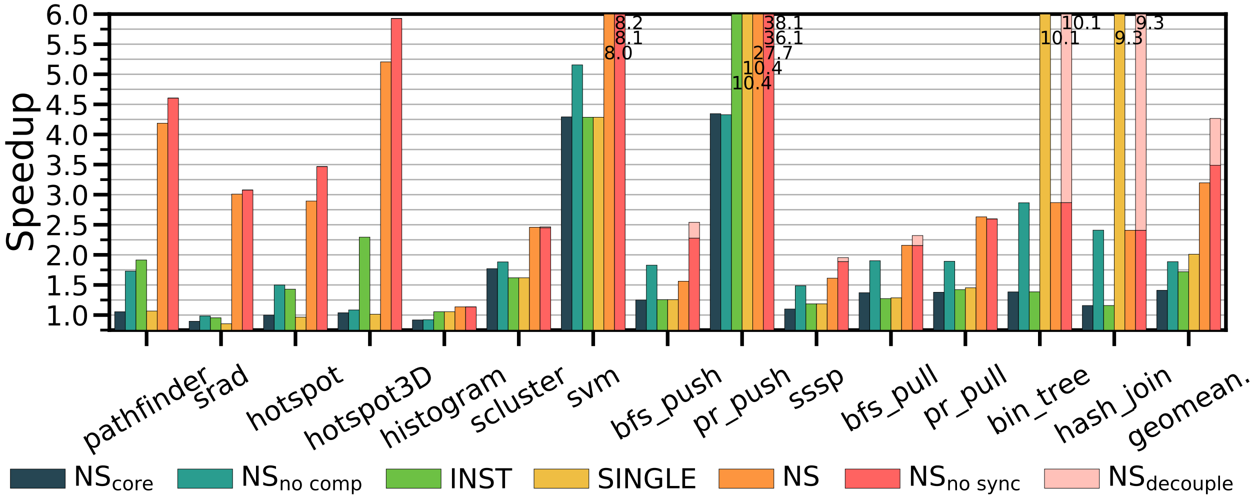


Figure 5.7: Overall Speedup over Base OOO8 Core

speedup over the OOO8 core, 1.69× over NS$_{\text{no comp}}$, and 1.85× over INST. With sync-free support, NS$_{\text{decouple}}$ achieves 4.27× speedup over the OOO8 core and 2.12× over SINGLE.

Fig 5.8 shows the normalized energy-performance tradeoff of different core sizes across workloads. All core types see similar speedups, with in-order cores benefiting the most (4.28× for NS over IO4). Because of the reduced communication and improved performance (less static energy), NS and NS$_{\text{decouple}}$ 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 states, and we estimate the area using CACTI and McPAT (22nm). SE$_{\text{CORE}}$'s stream buffer takes 0.09mm^2 [32]. SE$_{\text{L3}}$ requires a 64kB buffer to hold the stream operands and results, which takes 0.195mm^2. Adding the SE$_{\text{L3}}$'s stream configuration (48kB, 0.11mm^2) [32] and other components, the whole chip area overhead is 2.5% for IO4 (2.1% for OOO8 as SE$_{\text{CORE}}$ has larger FIFOs).

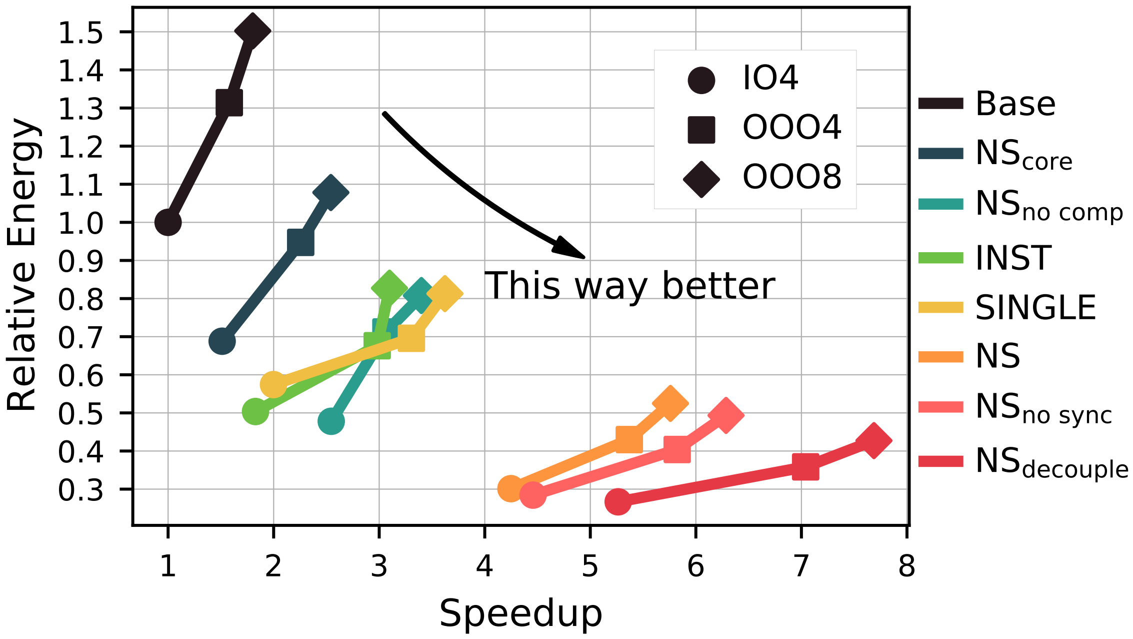


Figure 5.8: Energy vs. Speedup for IO4, OOO4, OOO8

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 Fig 5.9 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

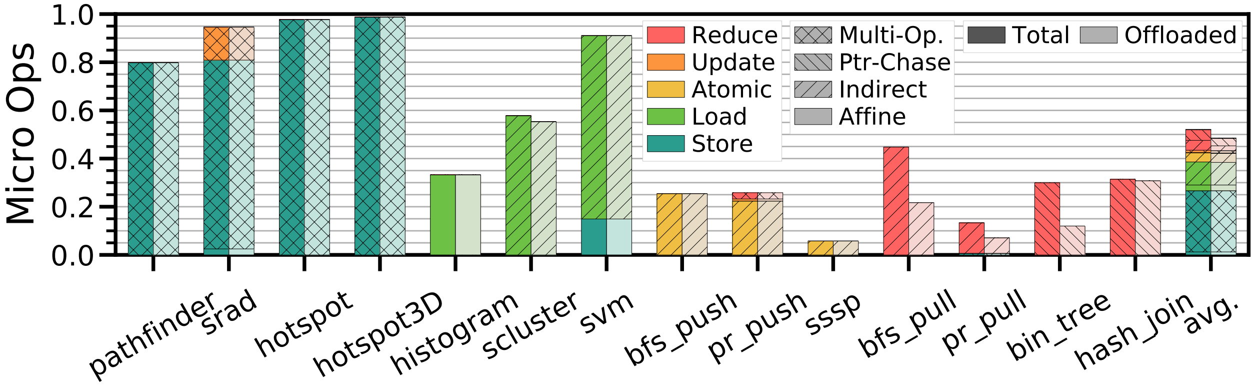


Figure 5.9: Breakdown of Dynamic Micro Ops

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 [79, 133], 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 Fig 5.10. 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 indirect atomics.

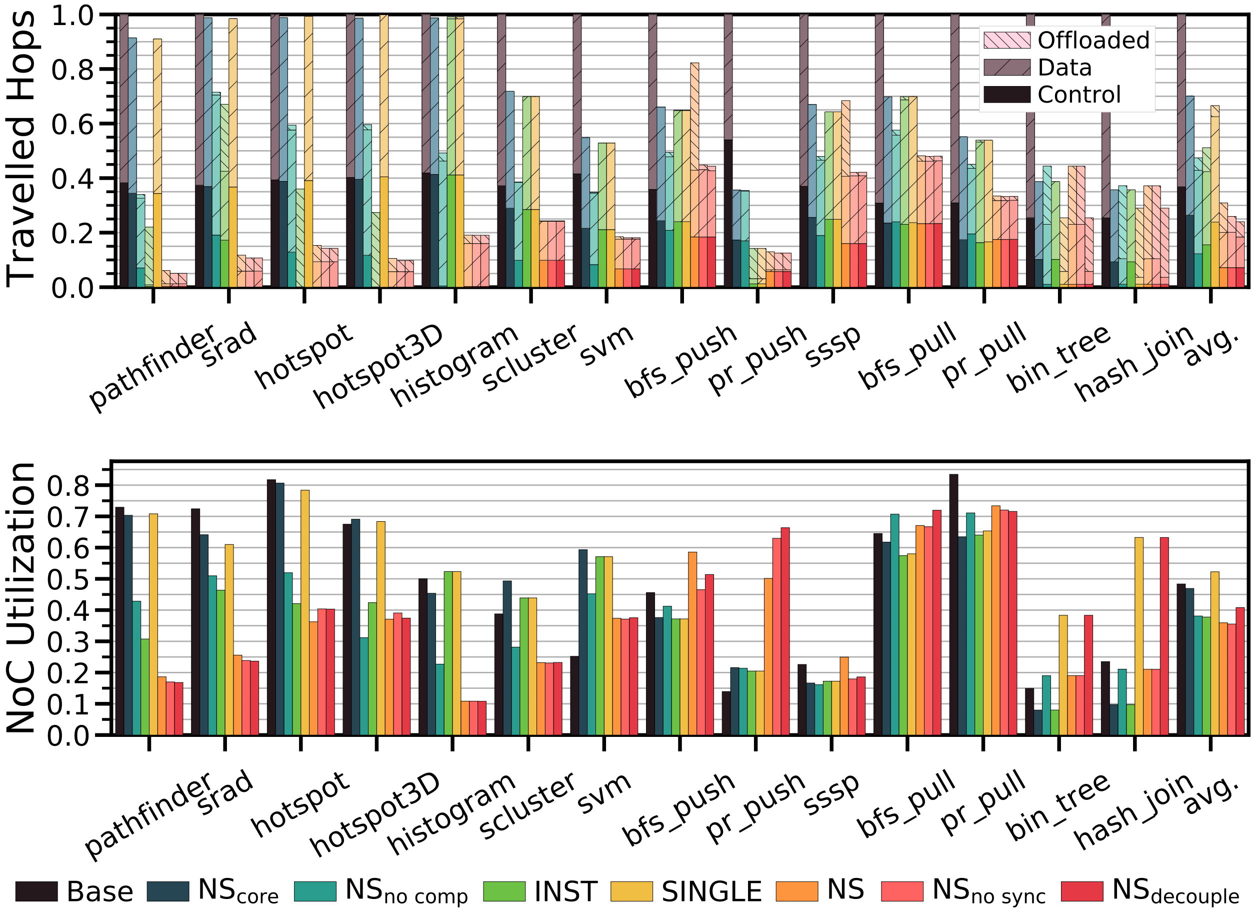


Figure 5.10: OOO8 NoC Traffic (top) and Utilization (bottom)

With synchronization eliminated in NS$_{\text{decouple}}$, 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$_{\text{decouple}}$ in many cases. The traffic is sometimes higher, as SINGLE cannot achieve autonomy on indirect atomics and falls back to iteration-level offloading.

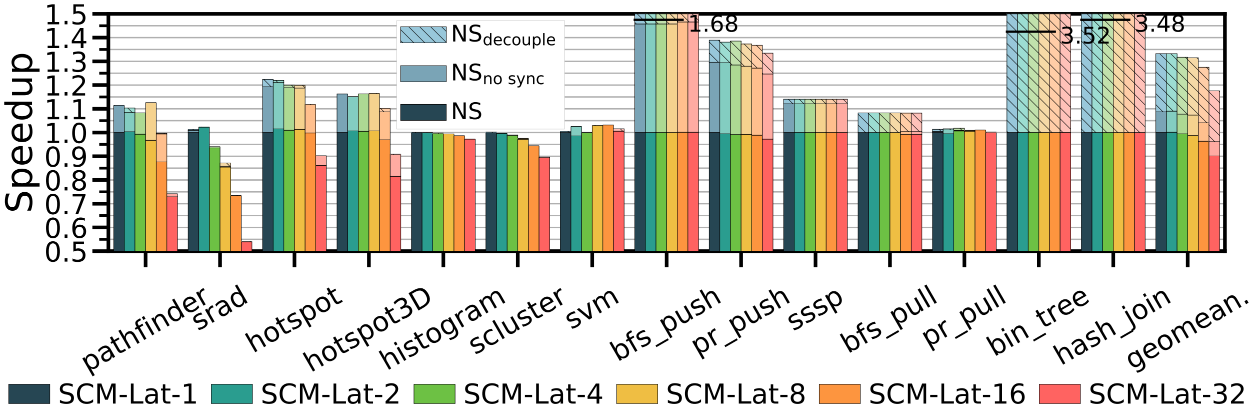


Figure 5.11: NS, $\text{NS}_{\text{no sync}}$, $\text{NS}_{\text{decouple}}$ with 1-32 SCM Latency

Sensitivity to Offload Capability

Fig 5.11 shows the performance of $\text{NS}_{\text{decouple}}$, $\text{NS}_{\text{no sync}}$ 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 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 contain 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 $\text{NS}_{\text{decouple}}$ 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 Fig 5.12. 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 a 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 real computation, which accounts for the significant speedup. We believe 2 SCCs is a reasonable

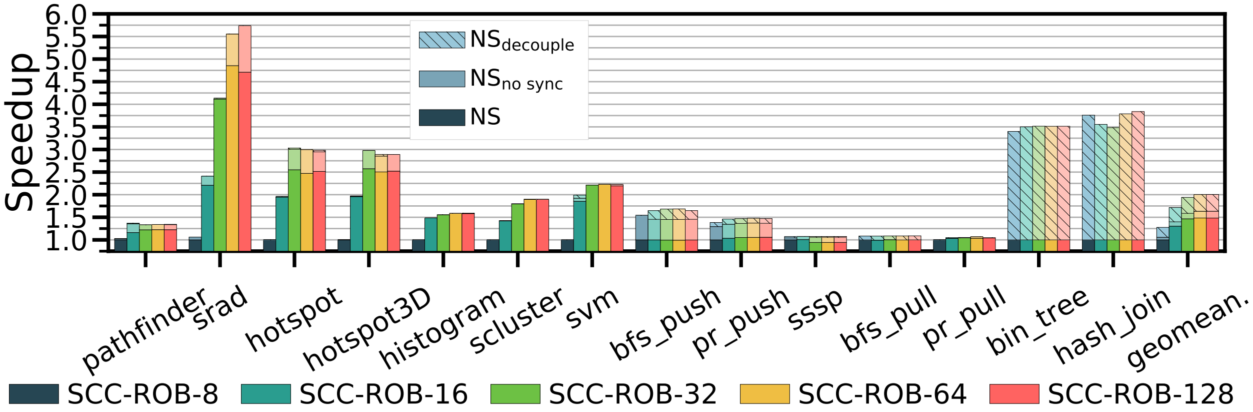


Figure 5.12: Sensitivity to 8-128 SCC ROB-Entry

choice for OOO cores since it requires fewer resources than a real hardware thread (less ROB/IQ and no LSQ entries). Overall, we set the default OOO8 configuration to 2 SCCs with a total of 64 ROB entries.

Other Sensitivity Studies

Affine Range Generation For affine streams, since the address pattern is known at configuration time, SE$_{\mathrm{CORE}}$ can generate the ranges to avoid the traffic of sending them from SE$_{\mathrm{L3}}$, at the cost of duplicate address generation and translation. Fig 5.13 shows the speedup and traffic of five affine workloads in NS with affine ranges sent by SE$_{\mathrm{L3}}$ or generated by SE$_{\mathrm{CORE}}$ (default behavior). The traffic data is classified as control, data and offloaded (same as Fig 5.10). For indirect workloads, SE$_{\mathrm{L3}}$ always sends the range as addresses are data-dependent. Overall, generating affine ranges at SE$_{\mathrm{CORE}}$ saves 15% traffic and achieves 5% performance improvement. NS$_{\mathrm{no\ sync}}$ and NS$_{\mathrm{decouple}}$ do not generate ranges as they require no range-based synchronization.

Lock Type Fig 5.14 shows the performance of exclusive and multi-reader single-writer lock (MRSW) on the three graph workloads with atomic operations (see §5.3). 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

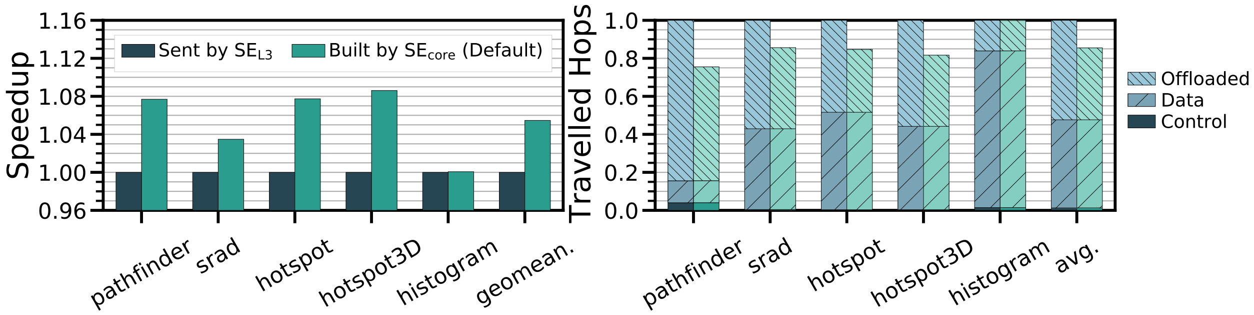


Figure 5.13: Sensitivity to Affine Range Generation (NS)

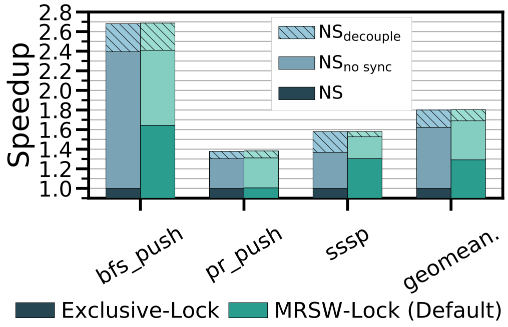


Figure 5.14: Exclusive vs. MRSW

with 1.29× speedup (NS). For $NS_{\text{no sync}}$ 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.

Scalar PE Both SE_{CORE} and SE_{L3} have a scalar PE to handle simple operations and avoid invoking SCM. Fig 5.15 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.

5.7 Additional Related Work

We discuss additional related work here; see §5.1 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 [5, 159, 22, 6, 160, 161, 162, 163].

Programmable architectures give varying degrees of control over how to schedule threads near data [137, 10, 164, 142, 165]. Thread-level offloading also enables programmer transparency. For example, in the context of GPUs, TOM [11] and Pattnaik et al. [12] 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 [143]. While transparent, they only make decisions at thread granularity.

Near What? Near-data computing is applicable in many contexts: in-cache [3], near mem-controller [27], near router [4], near-memory [10], near-storage [166], 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 [21, 149, 20] or DRAMs [167, 168, 23]. These techniques could provide highly parallel computation substrates for use in a near-stream system.

Speculative Multithreading Swarm [169, 170, 171] along with the T4 compiler [172], executes sequential programs speculatively in parallel as a series of tiny tasks. It supports scheduling such tasks near on-chip data [147]. In T4, near-data optimization is only applied for single cache line tasks.

Coherence and Synchronization Recent works provide better support for near-data

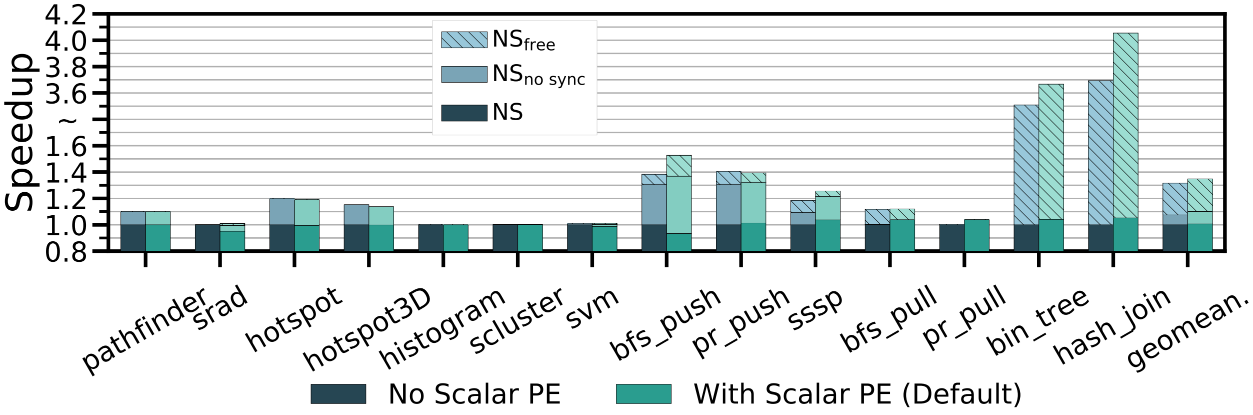


Figure 5.15: Sensitivity to Scalar PE in SE_{L3}

accelerator (NDA) coherence and synchronization. CoNDA speculatively executes NDA kernels while recording their memory accesses in bloom filters, condensing coherence traffic [165]. SynCron [164] 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 [102, 103, 104, 105, 106, 154, **?**].

Prefetching Prodigy [151] encodes indirect access patterns (similar to nested streams) in the program to efficiently prefetch into L1 cache. Event-triggered prefetcher [134] and Minnow [173] are programmable private-cache prefetchers for irregular accesses. However, prefetching-only techniques still suffer from the traffic overhead of fetching data into the core. Decoupled spatial architectures also leverage stream information for prefetching in accelerator designs [63, 113, 112, 114, 174].

EMC [115] 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.

5.8 Summary

In this chapter, we explore 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 computing 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 the 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.

CHAPTER 6

Affinity Alloc: Taming Not-So Near-Data Computing

With near-stream computing, we can flexibly offload computation along with streams to where the data is. However, the benefit of near-data computing heavily depends on spatial affinity, where all relevant data are in the same location, e.g. same cache bank. Existing NDC works lack a general and systematic solution: they either ignore the problem and abort NDC when there is no spatial affinity, or rely on error-prone manual data placement.

Our insight is that the essential affinity relationship, i.e. data A should be close to data B, is orthogonal to microarchitecture details and input sizes. By co-optimizing the data structure and capturing this general affinity information in the data allocation interface, the allocator can automatically optimize for data affinity and load balance to make NDC computations *truly* near data.

With this insight, in this chapter, we propose *affinity alloc*, a general framework to optimize data layout for near-data computing. It comprises an extended allocator runtime, co-optimized data structures, and lightweight extensions to the OS and microarchitecture. Evaluated on parallel workloads across broad domains, affinity alloc achieves 2.26× speedup and 1.76× energy efficiency over a state-of-the-art near-data computing technique with 72% traffic reduction.

Organization §6.1 discusses the data layout challenges and overviews our insight and proposed approach. §6.2 covers the basic interface and extensions to support affine layout, while §6.3 extends to irregular data layout. Methodology and evaluation are in §6.4 and §6.5. Further discussion and related work are in §6.6 and §6.7.

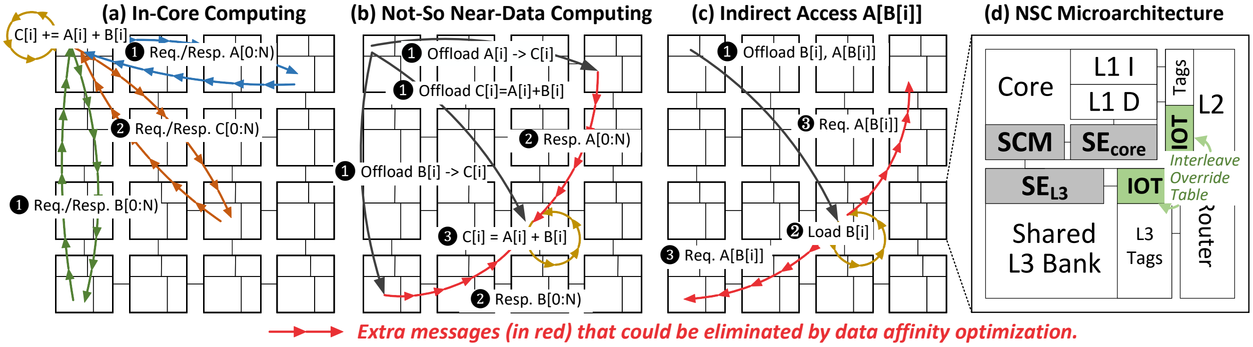


Figure 6.1: Affinity Optimization Opportunities in NDC (View in Color)

6.1 Motivation and Overview

Challenges and Insights

Simply pushing computing into the memory hierarchy does not guarantee that computation is now closer to the data, especially when the computation accesses more than a contiguous piece of data. Without a suitable data layout, the required operands may be scattered far away from each other. Fig 6.1 demonstrates, with Fig 6.1(a) depicting a conventional system. Fig 6.1(b) shows an NDC vector addition, where arrays not aligned in memory cause extra communication to collect operands. Fig 6.1(c) shows similar overheads for indirect accesses, which dominate graph processing workloads to access neighboring vertices. Naïvely offloading computation near data may yield no data movement reduction or even hurt the performance. Therefore, *an intelligent data layout decision is essential to fully realize the potential of near-data computing.*

Despite its importance, prior near-data computing work either relies on manual coarse-grained data partition on reserved scratchpad space using customized APIs [6, 175, 4, 137, 10], or requires domain-specific preprocessing (e.g. graph partitioning) [11, 145]. Other work simply is oblivious to the data layout, and falls back to the conventional computing paradigm when NDC is not profitable [27, 16, 3, 5, 31, 176]. They all fall short of providing a general and systematic solution to enabling guided and efficient data layout.

Challenges *We provide the first general and programmable framework that automatically optimizes data layout for near-data computing.* This is challenging as a hypothetical optimal data layout requires coordination of the entire system stack: to support customized data placement in the microarchitecture, to manage virtual to physical address translation, to expose network topology to the software, etc. Clearly, such a complex approach is not ideal.

This calls for *general* yet *concise* abstractions at each level of the system to efficiently convey the information required for intelligent data placement decisions. For *generality*, the interface should be expressive to specify broad data layout requirements, from simple strided layouts to complex fine-grained pointer-based alignment. For *simplicity*, the interface should only convey the minimal essential information across layers to maintain portability. This works in both directions: the software should be agnostic to the microarchitecture, while the hardware should be oblivious to the actual data structures. The interface should be compatible with general programming languages and be expressive enough to enable advanced layout optimizations for near-data computing.

Insight I To tackle these challenges, our first insight is that data placement *can* and *should* be optimized with data allocation. This is possible because most data layout requirements are known at allocation time [177], e.g. when allocating a linked list node, the previous node is already allocated, and if the new node can be placed closer to it, we can significantly reduce data movement when chasing the pointer. Also, picking the optimal data layout at allocation time saves the overhead of remapping later. Lastly, it incurs marginal programming complexity if the allocator can be reused as the new data placement interface. However, existing data allocators are either unaware of the data placement (e.g. `malloc`), or are imperative and opaque (e.g. `numa_alloc_onnode`), still leaving the placement decision to the programmer. We need a better allocator.

Insight II Secondly, instead of directly dictating the data placement, the new allocator interface should capture the essential data alignment constraints for efficient near-data computing. Such constraints are general to describe complex data affinity relationships, e.g. the new linked list node should be close to the previous one. Also, they are determined by algorithms and data structures, but orthogonal to the microarchitecture details. This is crucial to maintain transparency and portability, freeing programmers from the burden of manual placement for each hardware generation.

Insight III Perhaps most importantly, exposing a new allocator interface unlocks a variety of new opportunities to co-optimize the *data structure* to data affinity in NDC scenarios. For example, in graph algorithms, a global queue can be replaced by a spatially distributed queue to avoid remote accesses when pushing a new vertex into the frontier. Another example is using linked lists to replace the index array `B[i]` for indirect accesses `A[B[i]]`. Conventionally, traversing a linked list requires costly pointer chasing and is not as efficient as an array. However, it provides the flexibility to place the index closer to the destination data `A[B[i]]` and may yield higher performance in NDC. Such opportunities are impossible without the new allocator considering data placement.

Our Approach To summarize, we name our approach *affinity alloc*, as it systematically captures and optimizes data affinity for near-data computing. It contains a carefully designed allocator interface to capture the affinity information, a runtime library to lower the alignment constraints to an efficient data layout based on the underlying hardware details, and a lightweight yet general microarchitectural scheme to control the data layout. This design enables significantly more flexibility over manual data placement – instead of fixing data structure locations, we only describe how data structure elements should be kept close together. More importantly, it enables co-optimization between data structures and data layout to make NDC computations *truly* near the data.

We apply affinity alloc to optimize data placement for near on-chip SRAM computing, i.e. the last-level cache (LLC). The LLC level is promising because capacity continues to

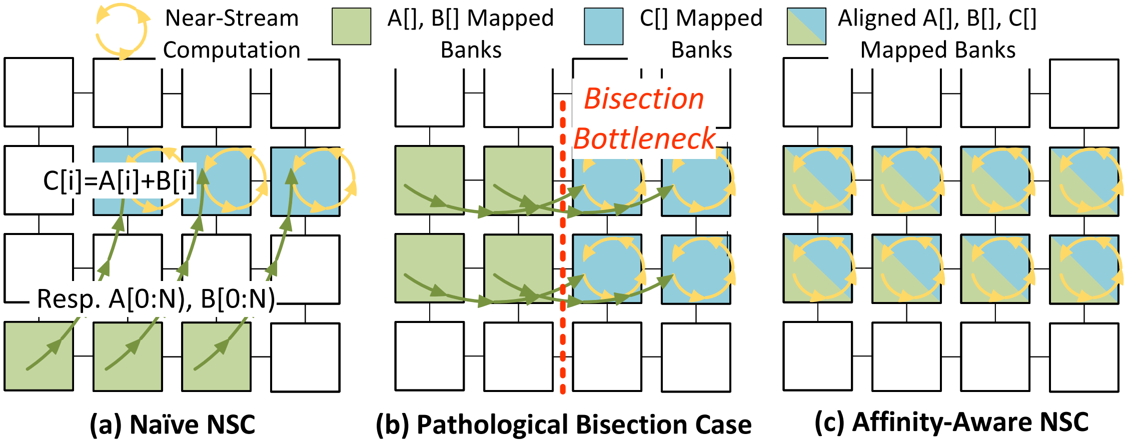


Figure 6.2: Affine Data Layout for Vec Add

scale in modern CPUs (768MB on AMD EPYC 7773X [178]), and many algorithms can be tiled for locality in the LLC. However, because affinity alloc addresses the fundamental data placement problem, its principles and implementation can be generalized to other near-data computing levels and techniques, e.g. near memory controller, in HMC die, near SSD, etc.

In the remainder of this chapter, we first quantify the potential benefits of having an optimal affine and irregular data layout, then overview our affinity alloc approach.

Affine Data Layout

We first consider a simple vector addition: `C[i]=A[i]+B[i]`. As shown in Fig 6.1(b) and Fig 6.2(a), When offloaded to the L3 cache, $\mathtt{s}_\mathtt{a}$ and $\mathtt{s}_\mathtt{b}$ forward the data to $\mathtt{s}_\mathtt{c}$, which writes back the added result. Intuitively, the placement of array `A[]`, `B[]` and `C[]` in the shared L3 banks directly affects the data forwarding traffic and performance.

Fig 6.2(a) shows a naïve affine data layout for the vector addition. For simplicity, we assume `A[]` and `B[]` are aligned in the shared L3 cache. However, since `A[]` and `B[]` are not aligned with `C[]`, we have to forward both operands through the network, leading to *not so* near-data computing. Such oblivious data layouts may even lead to pathological cases.

For example, in Fig 6.2(b), `C[i]` is mapped two banks behind `A[i]` and `B[i]`, causing a bisection bottleneck in the network and significantly reducing the effective bandwidth.

Therefore, an intelligent near-data computing system should be aware of the data affinity requirement and colocate all three arrays as shown in Fig 6.2(c). This eliminates the data forwarding traffic and fully unlocks the potential of near data computing.

To quantify the impact of affine data layout, Fig 6.3 shows the performance and network traffic of vector addition with various data layouts, normalized to baseline in core computing (no offloading). We use an 8x8 mesh NoC and control the data layout such that bank i always forwards to bank $(i + \Delta)$ mod 64 (methodology in §6.4). Although near-data computing always outperforms the baseline, its performance is very sensitive to the data layout (from 1.1× to 7.2×), as it dictates how much data traffic to forward the operands. A random data layout (i.e. each virtual page is mapped to a random physical page) avoids the pathological behavior, but only achieves 42% of the performance when data is aligned.

Challenges Even for this simple case, optimizing the data layout already requires optimizations across the whole system stack: to convey the data alignment requirement from the application, to translate virtual addresses in the OS, to control the physical cache line mapping in L3 banks, etc.

Irregular Data Layout

The analogous data layout problems for irregular data structures are even more complicated to solve. Fig 6.4(a) shows the baseline placement for a graph, using a compressed sparse row (CSR) format. We assume each cache line can hold two vertices (blue) or edges (green), and L3 banks are interleaved at cache line granularity. Many graph workloads (e.g. BFS, SSSP) scan edges and update pointed vertices. When offloaded in NSC, it takes 19 hops for indirect accesses to the vertices (green arrows) and 3 hops for stream migration (black arrows). However, as shown in Fig 6.4(b), if we can place the edges closer to the pointed-to

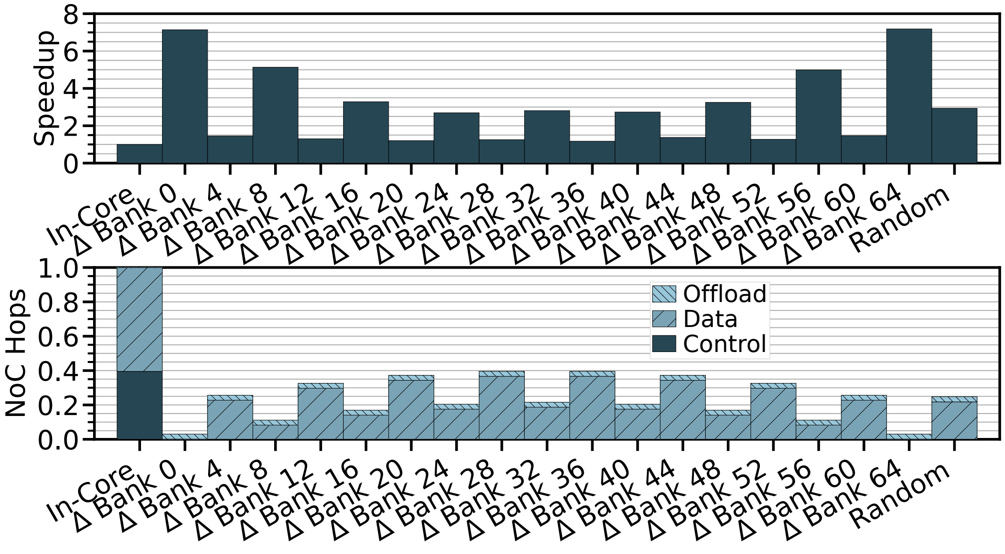


Figure 6.3: Impact of Affine Data Layout on Vec Add

vertices, we can significantly reduce the indirect access traffic to only 3 hops at the cost of a slightly longer migration distance.

To quantify such benefit, Fig 6.5 shows the speedup and traffic reduction if we can break the edge list in the CSR format into chunks of various sizes and freely map them to the L3 bank with minimal indirect traffic[1]. Smaller chunk sizes enable more fine-grained control on data layout. With 64B chunk (a cache line), irregular data layout optimization yields 60% traffic reduction and 2.14× speedup. An ideal configuration without indirect traffic achieves 4.1× speedup.

This demonstrates the potential of having an optimal data layout for irregular data structures, including other pointer-based data structures, e.g. linked lists, trees, etc. By optimizing the data layout, the overhead of irregular accesses can be significantly reduced.

Challenges Although promising, irregular data layout is even more challenging, as it requires fine-grained cache line layout and load balancing to ensure bank-level parallelism.

[1] Subject to a max 2% load imbalance between L3 banks, by moving chunks with the least traffic reduction to the least occupied bank.

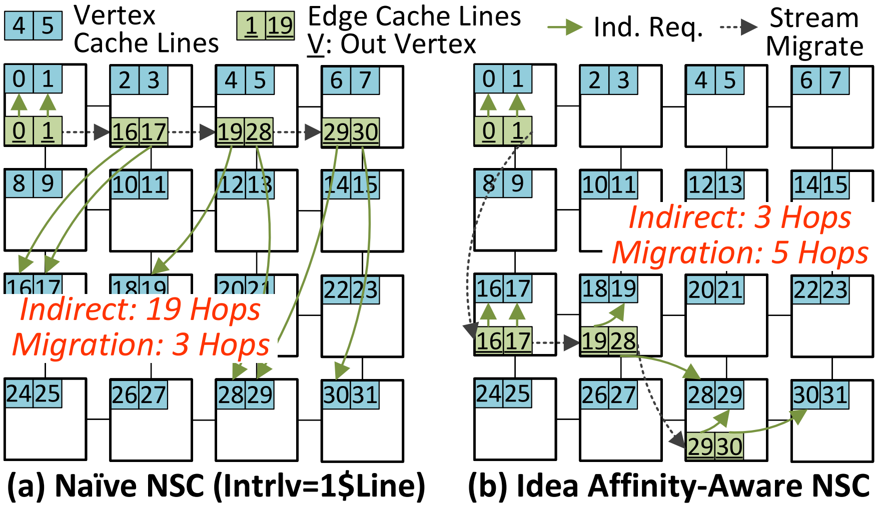


Figure 6.4: Irregular Data Layout for Graph Edge List

Affinity Alloc Approach Overview

To exploit these opportunities, we propose *affinity alloc*, a systematic data placement solution that optimizes data affinity during allocation for near-data computing. Fig 6.6 overviews the approach across different system levels.

Instead of having an imperative interface that exposes microarchitectural details and leaves the placement to the programmer (e.g. `libnuma`), an affinity alloc application only needs to convey the affinity information through the declarative allocator API. For example in Fig 6.6, when allocating a tree node, the pointer to the parent node is passed in so that the allocator can try to allocate the new node to the same bank as the parent node. Such affinity information is general enough to capture the essential relationship: that these pieces of data are used together and should be colocated.

To coordinate affinity information across all system levels, affinity alloc is designed by the divide and conquer principle: each layer tackles a simpler subproblem and only minimal information is exchanged between layers. Each layer is almost transparent to other layers. Specifically:

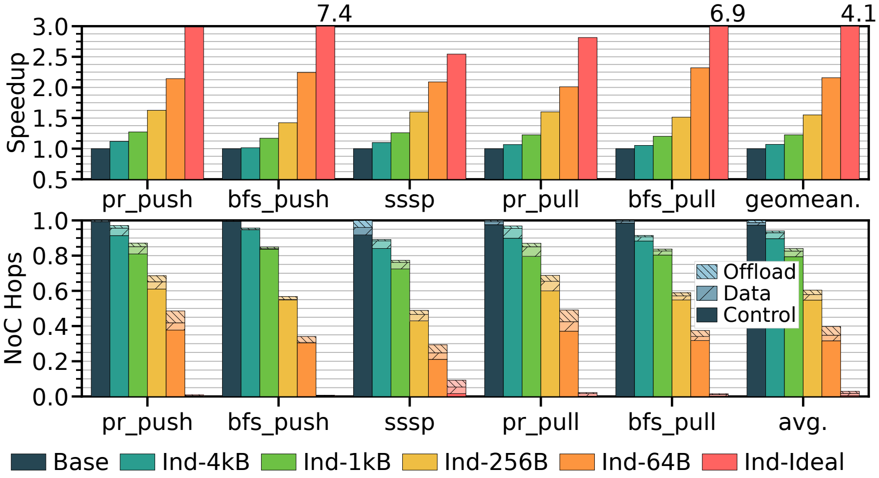


Figure 6.5: Impact of Irregular Data Layout

- **Application**: We choose to enhance the allocator with affinity information (either an affine pattern for affine layouts or a list of affinity addresses for irregular layouts). This significantly reduces the programming complexity as affinity information can be straightforwardly extracted from the data structure, e.g. parent node in the binary search tree. Also, since affinity information is purely determined by the algorithm and data structure but orthogonal to the underlying microarchitecture, portability is maintained by linking a platform-optimized runtime.

- **Runtime**: Similarly in Fig 6.6, The runtime is unaware of the data structure, but simply takes the affinity information and underlying network topology to determine the interleaving and the bank to allocate from. It also tracks the load balance to avoid creating a hot spot in the system. For example, the node `n2` is colocated with its parent `n5` for affinity, while `n7` is spilled to bank 1 for load balancing (see bottom of Fig 6.6). To allocate, the runtime maintains a free list that is aware of the L3 banks and may require more space from the OS.

- **OS**: The OS simply manages a pool for different interleaving sizes. Interleave pools are reserved in virtual address space when starting a program, and backed by contiguous

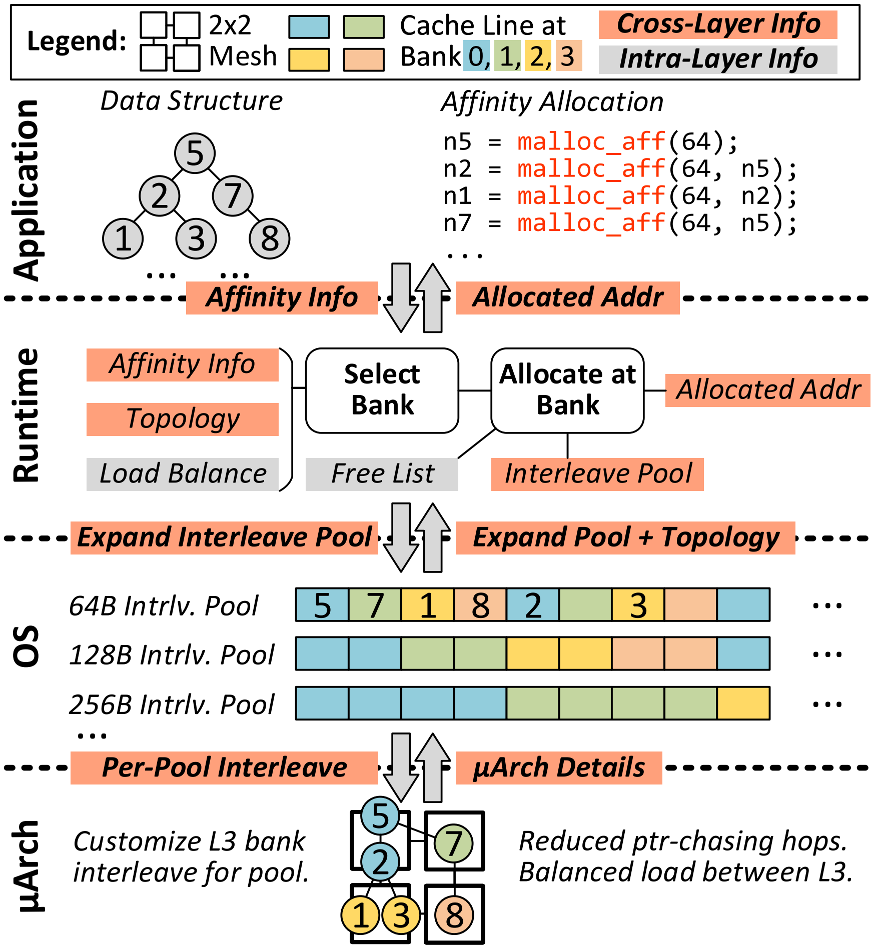


Figure 6.6: Affinity Alloc Approach Overview

physical addresses similar to a segment when accessed. It also passes the topology information to the runtime but is oblivious to the data structure or the load balance.

- **Microarchitecture**: It supports customizable interleaving for physical addresses within interleave pools but is unaware of any program-specific details.

Data Structure Co-Optimization Affinity alloc also enables novel data structure co-optimizations to harness the new opportunities from managing the data affinity. One example in the context of iterative graph processing is a spatially distributed work queue, leveraging the affine layout. Compared to a global queue, it reduces the overhead of managing the frontier in BFS and SSSP, as vertices can be pushed to the aligned local sub-queue with no remote accesses. This is possible in accelerators [179, 174, 169, 148, 147], but difficult for general-purpose processors without control over affinity.

Also, by supporting fine-grained irregular data layout, we can use a linked list to replace the array holding all edges in the compressed sparse row format (CSR). This provides the flexibility to colocate edges with the outgoing vertices, reducing the indirect traffic. To our knowledge, this optimization has not been explored even for accelerators, because of the lack of fine-grain affinity control.

More generally, data structures for near-data computing face significantly different tradeoffs. While contiguous arrays often have the benefit of simple prefetching on general architectures, affinity-based allocation and near-data computing offer significant advantages to pointer-based structures. Thus, affinity alloc opens new opportunities for codesign in the near-data computing era, which would otherwise be impossible or impractical to program.

Affinity Alloc Overview Overall, affinity alloc adopts a clean layered design: the application specifies the affinity information, the runtime performs the affinity-aware allocation with load balancing, the OS manages the pools with different interleaving sizes, and the microarchitecture simply customizes the interleaving for each pool. With these lightweight extensions and data structure co-optimization (see §6.3), affinity alloc provides a general and systematic solution to make NDC computations *truly near data*.

6.2 Affine Data Layout

In this section, we take a bottom-up view: how to efficiently support customizable mapping from virtual address space to L3 bank locations in the microarchitecture and OS, then how the application and runtime leverages it to optimize for data affinity.

Mapping Virtual Addresses to L3 Banks

One obstacle to NDC data affinity optimization is that the mapping from virtual addresses to shared L3 banks is hidden from the user space or even the OS. First, address translation is managed by the OS. Also, modern CPUs usually employ complex hash functions to map a physical address to an L3 bank [180] to exploit bank-level parallelism and avoid hot spots. Therefore, we need to expose the mapping from virtual addresses to L3 banks to the software.

Interleave Pool As shown in Fig 6.6, we introduce *interleave pools*. Each interleave pool is a reserved segment in the virtual address space, and addresses within an interleave pool are guaranteed to be mapped to L3 banks with the specified interleaving. For example, 64B cache lines within the 64B interleave pool are linearly mapped to L3 banks one by one. Given a pool with interleaving *intrlv* and starting virtual address *start*, we can compute the L3 bank for a given virtual address *vaddr* within the pool:

$$\text{bank}(vaddr) = \lfloor \frac{vaddr - start}{intrlv} \rfloor \pmod{N_{bank}} \tag{6.1}$$

Similar to the heap, interleave pools are managed by the OS, and the runtime can request an expansion (similar to how `mmap` or `brk` is used to expand the heap). We provide a pool for power-of-two interleavings from 64B (one cache line) to 4kB (one page, see below for larger interleavings), i.e. 7 interleave pools per process[2].

Physical Address Each interleave pool is mapped to contiguous physical pages. To ensure this, when the OS handles a page fault on an unmapped interleave pool virtual address *vaddr*, it will allocate physical pages from the start of that interleave pool until *vaddr*, and may copy data and remap pages to make sufficient space (similar to how direct segment [181] or RMM [182] supports continuous virtual to physical mapping). This simplifies the data layout control in the hardware. To complete the picture, the microarchitecture is extended

[2] We reserve 1TB per interleave pool, which in total is 2.7% of the 48-bit virtual address space.

Field	Bits	Description	Field	Bits	Description
`start`,`end`	48	[start, end) phys. addr.	`intrlv`	16	Interleaving.

Table 6.1: Interleave Override Table (IOT)

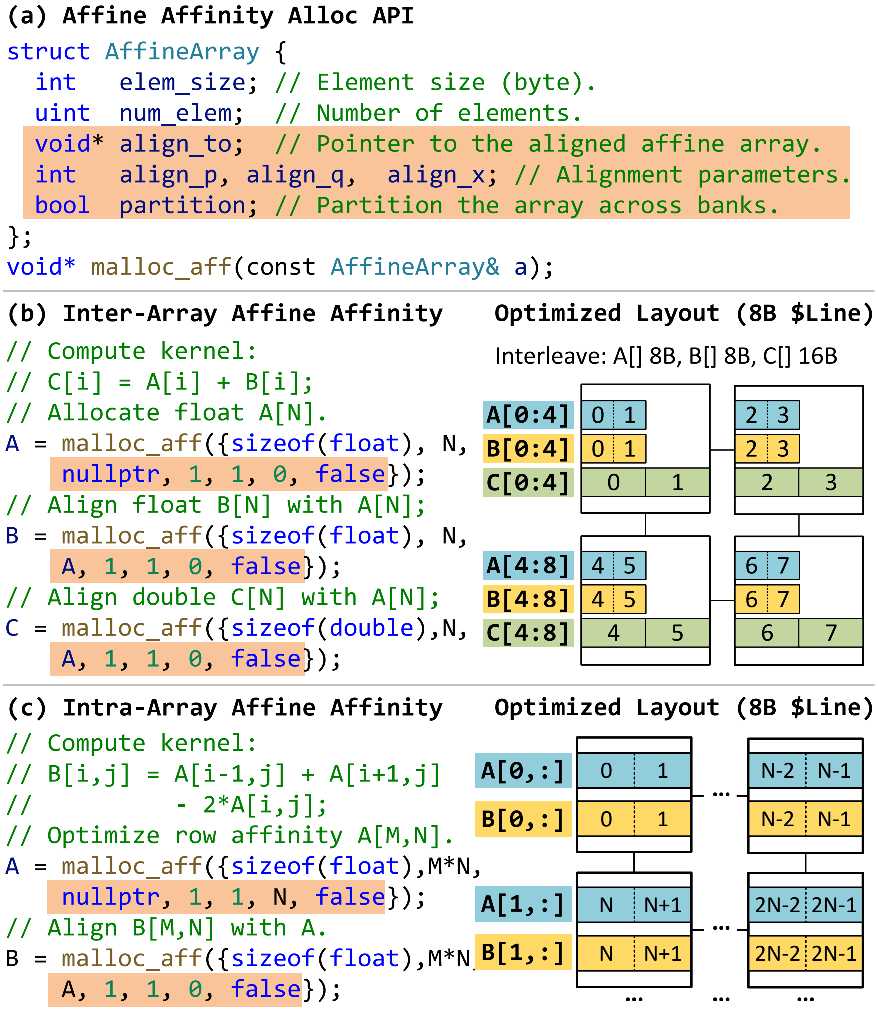
with an interleave override table (IOT, Table 6.1) at each L2 and L3 cache controller.

Each entry overrides the interleave for physical addresses within $[start, end)$. The L2/L3 cache controller as well as the SE_{CORE}/SE_{L3} query this table to determine which bank a cache line is mapped to, so that it can forward the request or offload/migrate the stream. Since this table is accessed frequently (every L2 miss and L3 access), mapping each interleave pool to contiguous physical addresses ensures that only one IOT entry is required per interleave pool, reducing the pressure on the size of IOT.

Other Interleavings Interleavings below a cache line size (64B) are not supported, as they spread a single cache line to multiple L3 banks. This requires extra metadata to track sub-line coherence states and is beyond this work. Large interleavings beyond a page size (4kB) but aligning to page boundaries (e.g. 8kB, 12kB) are supported by mapping virtual pages to 4kB interleaved physical pages at the desired L3 bank[3]. Finally, interleavings that are not power-of-two help reduce the padding overhead, and can be supported at the cost of a more complicated division instead of a right shift in Eq. 6.1 when querying the IOT. This is left as future work.

Other Interleave Patterns The mapping from virtual addresses to L3 banks (i.e. Eq 6.1) is a simple 1D linear pattern. More complicated interleaving patterns can also be supported, e.g. a 2D pattern that fills L3 banks in the order of quadrant, or a two-level wrapping around that first wraps a few times within each row before moving to the next row. These more sophisticated interleave patterns can be supported by either changing how L3 banks are

[3] Physical pages for these interleavings are not continuous and are tracked as 4kB interleaving in the IOT.



(a) Affine Affinity Alloc API

```
struct AffineArray {
  int   elem_size; // Element size (byte).
  uint  num_elem;  // Number of elements.
  void* align_to;  // Pointer to the aligned affine array.
  int   align_p, align_q,  align_x; // Alignment parameters.
  bool  partition; // Partition the array across banks.
};
void* malloc_aff(const AffineArray& a);
```

(b) Inter-Array Affine Affinity

```
// Compute kernel:
// C[i] = A[i] + B[i];
// Allocate float A[N].
A = malloc_aff({sizeof(float), N,
    nullptr, 1, 1, 0, false});
// Align float B[N] with A[N];
B = malloc_aff({sizeof(float), N,
    A, 1, 1, 0, false});
// Align double C[N] with A[N];
C = malloc_aff({sizeof(double),N,
    A, 1, 1, 0, false});
```

(c) Intra-Array Affine Affinity

```
// Compute kernel:
// B[i,j] = A[i-1,j] + A[i+1,j]
//          - 2*A[i,j];
// Optimize row affinity A[M,N].
A = malloc_aff({sizeof(float),M*N,
    nullptr, 1, 1, N, false});
// Align B[M,N] with A.
B = malloc_aff({sizeof(float),M*N
    A, 1, 1, 0, false});
```

Figure 6.7: Affine Data Layout Optimizations

numbered or enhancing Eq 6.1, and can provide more flexibility for the runtime to optimize the data layout. However, we find that a simple 1D linear pattern is expressive enough to achieve optimal spatial affinity for the affine workloads we studied.

Affine Layout Optimizations

With the OS and microarchitectural extensions to expose the mapping from virtual addresses to L3 banks, it is already possible for the application to customize the data layout. However, instead of leaving this burden to the programmer, we provide a runtime that automatically optimizes for the data layout and requires only abstracted affinity information from the application.

Affine Affinity Alloc API Fig 6.7(a) shows the API to allocate an array with affinity information wrapped in the `AffineArray` struct. Besides the size of the element (`elem_size`) and the number of elements (`num_elem`), it also contains parameters to define the affinity relationship between arrays (orange box in Fig 6.7(a)).

Inter-Array Affine Affinity Fig 6.7(b) shows how the API is used to optimize inter-array affine affinity. First, array `A[N]` is allocated with all default parameters, and the runtime simply picks the default interleaving, which is the cache line size (8B in Fig 6.7(b)). When allocating array `B[N]`, we specify that `B[i]` aligns with `A[i]` by setting `align_to` to `A`. More generally, the affinity relationship between the allocating array `B[N]` and the aligned-to array `A[N]` is defined as:

$$B[i] \xrightarrow{aligns\ to} A[\frac{align\_p}{align\_q} \times i + align\_x] \tag{6.2}$$

Here `align_p` and `align_q` control the ratio between the aligned element indexes, and `align_x` adds the offset. Essentially, this is equivalent to defining an affine transformation $y = Ax + b$ between the index space. These parameters can be straightforwardly determined from the access pattern, e.g. to align `B[i]` to `A[4i+2]`, simply set `align_p=4`, `align_q=1` and `align_x=2`.

The runtime records the metadata and selected layout of allocated arrays. When allocating a new array with inter-array affine affinity, it computes the interleaving of the new array by considering the ratio of element sizes and the interleaving of the aligned-to array. Specifically, the new array's interleaving is computed by:

$$intrlv_B = \frac{elem\_size_B}{elem\_size_A} \times \frac{align\_q}{align\_p} \times intrlv_A \tag{6.3}$$

By factoring in the ratio of element sizes, the runtime chooses a 16B interleaving for the array `double C[N]` in Fig 6.7(b). From the perspective of L3 bank locality, this effectively converts the struct-of-array into an array-of-struct, with each element aligned within the same L3 bank to eliminate data forward traffic.

Once the interleaving is determined, the runtime allocates from the corresponding interleave pool and ensures that the start bank is offset by $align\_x \times elem\_size_A / intrlv_A$. Notice that in certain cases the alignment is not perfect, i.e. when $align\_x \times elem\_size_A$ is not a multiple of $intrlv_A$, or when we have to round the computed $intrlv_B$ to a valid interleaving supported by the system. However, such cases can be mitigated by padding the array and supporting interleavings that are not power-of-two in future work (see below). Currently, in these cases, the runtime can simply fall back to the baseline allocator without hurting the performance.

Freeing Data Data allocated by `malloc_aff()` is freed with `free_aff(void*)` (omitted in Fig 6.7(a)). Since the runtime records the metadata for allocated arrays, it can put the space back to the free list similar to a normal allocator.

Intra-Array Affine Affinity We also support affinity within a single array. In Fig 6.7(c) we access the column of the 2D array `A[M,N]` and hence want to optimize for affinity between rows. This can be done by setting `align_to` to `nullptr` and `align_x` to `N`[4]. The runtime picks a valid interleaving that minimizes the Manhattan distance between `A[i]` and `A[i+N]`. For example, in Fig 6.7(c) one row of array `A[M,N]` is mapped to one row of the mesh topology, and the Manhattan distance is one hop to the bank below it. When `N` is small, the runtime could also pick an interleaving that fits one or multiple rows into a single bank to further reduce the distance. Array `B[M,N]` is handled with inter-array affine affinity.

[4] For intra-array affinity `align_p|q=1`, as otherwise the alignment is no longer affine.

Partition Vertexes with Spatial Queue

```
// Distribute vertex partition.
V = malloc_aff({sizeof(T), N,
    nullptr, 1, 1, 0, true});
// Align spatial queue to V[N].
Q = malloc_aff({sizeof(int), N,
    V, 1, 1, 0, false});
// Align queue tails to V[N].
T = malloc_aff({sizeof(int64), P,
    V, N/P, 1, 0, false});
// Push v into Q (atomic ++).
Q[T[v*P/N]++] = v;
```

Optimized Layout

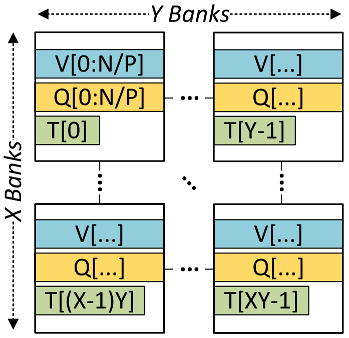


Figure 6.8: Distribute Partitions (Assume $P = X \times Y$)

Distribute Partitions We deliberately design the interface to only specify the general affinity relationship, and delegate the runtime to select a proper interleaving across platforms. However, the programmer may want to have a very coarse-grained interleaving, especially when distributing a partitioned array across banks. Since `align_p/q/x` can only specify the affinity information but not interleaving, we add a `partition` flag to force an interleaving that evenly distributes the array across all banks. Fig 6.8 shows a common use case in graph processing when the vertex array `V[N]` is partitioned among banks by setting `partition` to `true`.

Use Case: Spatially Distributed Queue Another more sophisticated use case of affinity alloc is to implement a spatially distributed queue. In a push-based BFS, the updated vertex `v` is pushed into a global queue for future processing. However, the tail of the global queue and the writing position is not colocated with the vertex, requiring indirect traffic to push into the global queue.

Instead, in Fig 6.8 we allocate a spatially distributed queue, with one sub-queue per partition. The tail pointer and data storage of each sub-queue is aligned with the vertex partition, and when pushing a vertex `v`, it is pushed to the local sub-queue with no indirect traffic. Affinity alloc supports mismatch between the number of partitions `P` and L3 banks

```
void* malloc_aff(uint size, // Alloc size.
    // Specify affinity addrs.
    int num_aff_addrs, void** aff_addrs);
void linked_list_append(Node *prev, T v)
    // Allocate new node near to prev.
    Node *n = malloc_aff(sizeof(Node), 1, &prev);
    n->v = v; n->nxt = prev->nxt; prev->nxt = n;
```

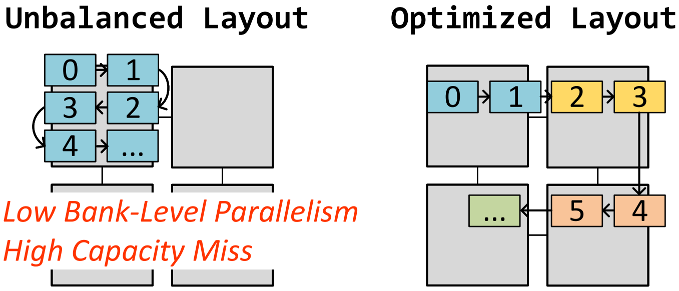


Figure 6.9: Irregular Data Layout API

B (i.e. P≠B), but having them equal yields better load balancing and higher performance. Priority queues, e.g. MultiQueues [183], can also be implemented as one queue per bank. Heap rearrangement involves pointer-chasing, which is supported by NSC. This software optimization is not possible without affinity alloc to control the data alignment.

6.3 Irregular Data Layout

Support Irregular Layout

While affine access patterns are relatively simple to optimize, irregular access patterns such as indirect and pointer-chasing accesses are data-dependent and are notorious for low spatial locality. However, with a small extension to the API, we show that affinity alloc can optimize the data layout for irregular data structures *without extra modification* to the OS or microarchitecture.

Irregular Layout API Fig 6.9 shows the irregular affinity allocation API and function to allocate a new node to a linked list using affinity alloc. In addition to the allocating size, the API can also provide a list of *affinity addresses* that the newly allocated data should be close to. In the linked list example, it is the previous node `prev`. Affinity addresses should be within some interleave pool so that the runtime can infer the mapped L3 bank. This simple yet powerful API conveys sufficient information to the runtime to optimize for irregular affinity while remaining oblivious to the actual allocated data structure. We limit the maximal number of affinity addresses per allocation to 32, and the application can sample a subset if there are more affinity addresses.

Irregular Allocation To allocate, the runtime rounds up the allocating size to a valid interleaving size. This usually incurs no overhead, as irregular data structures often use allocation sizes that are power-of-two and aligned to cache line granularity to avoid false sharing. The runtime also maintains a free list for every valid interleaving size and every bank. After selecting the bank to allocate based on the affinity addresses and load balance (see §6.3), the runtime allocates from the free list of that bank, and may require the OS to expand the specific pool if running out of space.

Free Data To free an object allocated with irregular layout API, we reuse the same interface `free_aff(void*)`. The runtime distinguishes irregular layout objects from affine arrays by checking if the address matches an allocated affine array. The interleaving of the object can be directly inferred from the interleave pool it belongs to. Since irregular layout objects are allocated at interleave granularity, the runtime knows the size of the object and can free the space by adding it back to the free list. Unlike conventional allocators, the runtime maintains *no meta-data* for irregular layout objects, avoiding space overheads for fine-grained allocations.

All modifications to support irregular data layout are limited to application and runtime. The OS and μArch only needs to handle coarse-grained interleave pools.

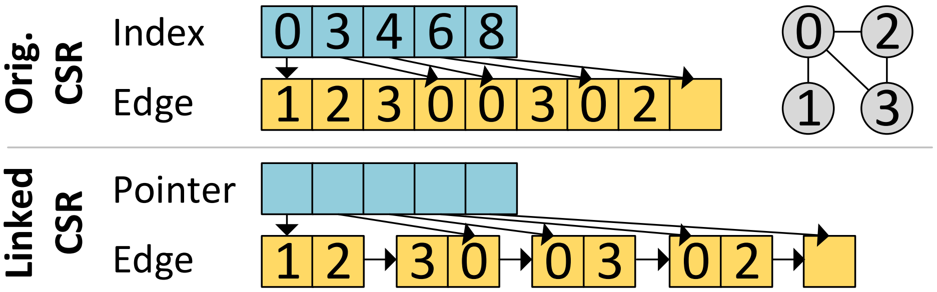


Figure 6.10: Linked CSR Format

Bank Select Policy

Simply optimizing for data affinity may result in pathological unbalanced layout. For example, in the bottom left of Fig 6.9, the whole linked list is allocated to a single bank, leading to low bank-level parallelism and high capacity miss rate. Therefore, we design the bank select policy to consider both data affinity and load balance. Specifically, the runtime computes a score for each bank:

$$score = avg\_hops + H \times (\frac{load}{avg\_load} - 1) \tag{6.4}$$

Here *avg_hops* is the average hops to the provided affinity addresses, and *load* is the number of irregular allocations to that bank. H is a weight coefficient to control how much the runtime should emphasize load balancing. The bank with the minimal score is selected. This score function is inspired by the one used by ABNDP [13] to optimize task scheduling, while here we extend it for data allocation. We evaluate the sensitivity to H in §6.5.

Data Structure Co-Optimization

Supporting irregular affinity allows the runtime to optimize the data layout for a variety of data structures, provided that they offer sufficient flexibility for data placement. This covers many important pointer-based data structures, e.g. linked lists and trees. Such data structures can benefit from affinity alloc without changing the data organization itself, simply by adopting the new allocator API.

Similarly, our approach opens up many new codesign opportunities for coarse-grained data structures that are not flexible enough to directly benefit from affinity alloc, e.g. the index array `B[]` in `A[B[i]]` can only be remapped at page granularity with marginal performance gain (Fig 6.5 in page 128). In this work, we focus on codesigning graph representations to optimize data affinity.

Fig 6.10 shows a toy undirected graph and the original compressed sparse row (CSR) format. In CSR format, each vertex has an `index` pointing to its first `edge`. However, since the edges are stored in a single array, we can only optimize for data affinity at very coarse granularity, i.e. partitioning the graph among banks with the affine layout API. However, power-law graphs are hard to partition with many inter-partition edges. We need more flexibility in the data structure to optimize data affinity at finer granularity.

This motivates for a *Linked CSR* format (Fig 6.10), in which the edges are stored in a linked list, and we can place each edge list node closer to the pointed vertices by specifying the affinity addresses. This is how we achieve the optimizations discussed in Fig 6.4 (page 127). This comes with the cost of extra pointer-chasing between nodes, which is usually much more expensive than the linear accesses in the original CSR format. However, we argue that the tradeoffs in near-data computing are very different: 1. Pointer-chasing overheads are amortized by indirect traffic reduction since each node can hold multiple edges. For example, a 64B cache line can hold 14 edges of 4B after the 8B pointer. 2. Unlike conventional CPUs where the run ahead distance is limited by the size of the reorder buffer (ROB), in NDC the pointer-chasing task can be decoupled and run ahead of the edge processing task, further hiding the latency.

Most importantly, co-optimizing the data structure with affinity alloc unlocks the benefit of the fine-grained irregular layout at a low cost ($O(|E|)$ to scan the edges once). This is the key to unlocking the full potential of NDC and can be applied to other domains.

System	2.0GHz, 8x8 Cores
OOO8 CPU	64 IQ, 72 LQ, 56 SQ+SB
(8-issue)	348 Int/FP RF, 224 ROB
Func. Units	8 Int ALU/SIMD (1-cycle)
	4 Int Mult/Div (3/12-cycle)
	4 FP ALU/SIMD (4-cycle)
	4 FP Div (12-cycle)
L1 D/I TLB	64-entry, 8-way
L2/SE$_{\text{L3}}$ TLB	2k/1k-entry, 16-way, 8-cycle
L1 I/D Cache	32KB, 8-way, 2-cycle
Priv. L2 Cache	256KB, 16-way, 16-cycle
Replacement	Bimodal RRIP, $p = 0.03$
L1 Bingo Pf.	8kB PHT, 2kB region
L2 Stride Pf.	16 streams, 16 pf./stream

NoC	8x8 mesh topology
	32B 1-cycle bidirection link
	5-stage router, multicast
	X-Y routing, 4 mem. ctrls
Shared	20 cycles, MESI
L3 Cache	Static NUCA, 1kB interleave
	16-way, 64 banks, 1MB/bank
	total 64MB
DRAM	3200MHz DDR4 25.6 GB/s
	4 channels at corners
SE$_{\text{CORE}}$	2kB FIFO, 12 streams
SE$_{\text{L3}}$	768 streams, 64kB buf.
	4-cycle compute init. lat.
IOT	16 regions

Table 6.2: System and μarch Parameters

6.4 Methodology

Compiler and Runtime We reuse the open-source LLVM-based near-stream computing compiler [31]. Programs are compiled to x86 and extended with near-stream computing instructions. We implement the affinity alloc runtime in C++ and manually replace the original `malloc` and `free` calls with affinity alloc API.

Simulator We use gem5 v20.0+ [107] for execution-driven, cycle-level simulation, extended with partial AVX-512 support. The caches are extended with NSC support and the interleave override table (IOT) to customize the interleaving between L3 banks. We emulate the syscall

Benchmark	Layout	Parameters
pathfinder [109]	Affine	1.5M entries, 8 iters
srad [109]	Affine	1k×2k, 8 iters
hotspot [109]	Affine	2k×1k, 8 iters
hotspot3D [109]	Affine	256×1k×8, 8 iters
bfs [158]	Linked CSR	Kronecker generated
pr_push [158]	Linked CSR	128k nodes 4M edges
sssp [158]	Linked CSR	A/B/C: 0.57/0.19/0.19
pr_pull [158]	Linked CSR	weight [1,255]
link_list	Ptr-Chasing	8B key, 512 nodes/list, 1 query/list, 1k lists
hash_join	Ptr-Chasing	8B key, 256k ⋈ 512k, Hit Rate 1/8
bin_tree	Ptr-Chasing	128k nodes, 8B key, 512k uniform lookups

Table 6.3: Workloads Parameters

to expand interleave pools in gem5. We leverage McPAT [50] to estimate the energy and area with the 22nm process.

Parameters and Configurations Table 6.2 lists system parameters. The only extension to the baseline near-stream computing system is the IOT to support customized L3 interleavings for interleave pools. The baseline OOO cores use advanced L1 and L2 prefetchers [110], but no computation is offloaded (labelled as **In-Core** in §6.5). For near-memory computing, **Near-L3** offloads streams and the associated computation to $\mathrm{SE}_{\mathrm{L3}}$, but is oblivious to data affinity. For affinity alloc, we simulate the modified binary with affinity information conveyed to the runtime.

Benchmarks We evaluate 10 OpenMP workloads compiled with `-O3` and AVX-512, covering various affine and irregular data layouts. For graph workloads, **In-Core** and **Near-L3**

use the original CSR format, while affinity alloc adopts the new linked CSR representation. For the pointer-chasing workloads, we randomly generate and insert the nodes into the binary tree without balancing it. For `link_list` and `hash_join`, they both search through link lists, but `link_list` has much longer lists (512) while the buckets in `hash_join` are much smaller (¡= 8). Table 6.2 summarizes the input data size and the major data layout pattern for each benchmark.

Some benchmarks have alternate implementations, i.e. push and pull-based for `page_rank` and `bfs`. For `page_rank`, we added the push version besides the original pull-based implementation in GAP suite [158], and select the best implementation for each configuration (pull version for **In-Core** and push version for **Near-L3** and affinity alloc). For `bfs`, the state-of-the-art implementation dynamically switches between pushing and pulling based on some runtime heuristics [184]. We discuss the tradeoffs between pushing/pulling and the heuristics we used in §6.5.

6.5 Evaluation

We first evaluate affinity alloc on a variety of workloads, bank selection policies and input sizes to demonstrate the performance and energy efficiency benefits due to improved data affinity. We then perform a detailed study on how key graph processing workloads benefit from codesigning the data structure with affinity alloc.

General Evaluation

Overall Performance Fig 6.11 shows the overall performance for all benchmarks. The speedup and energy efficiency are normalized to **Near-L3**, while the NoC traffic is normalized to **In-Core** where no computation is offloaded to the L3 cache. Overall, affinity alloc achieves 7.53× speedup and 4.69× energy efficiency over **In-Core**, and 2.26×/1.76× over **Near-L3**. The benefit comes from the reduced NoC traffic for various messages: the data traffic to

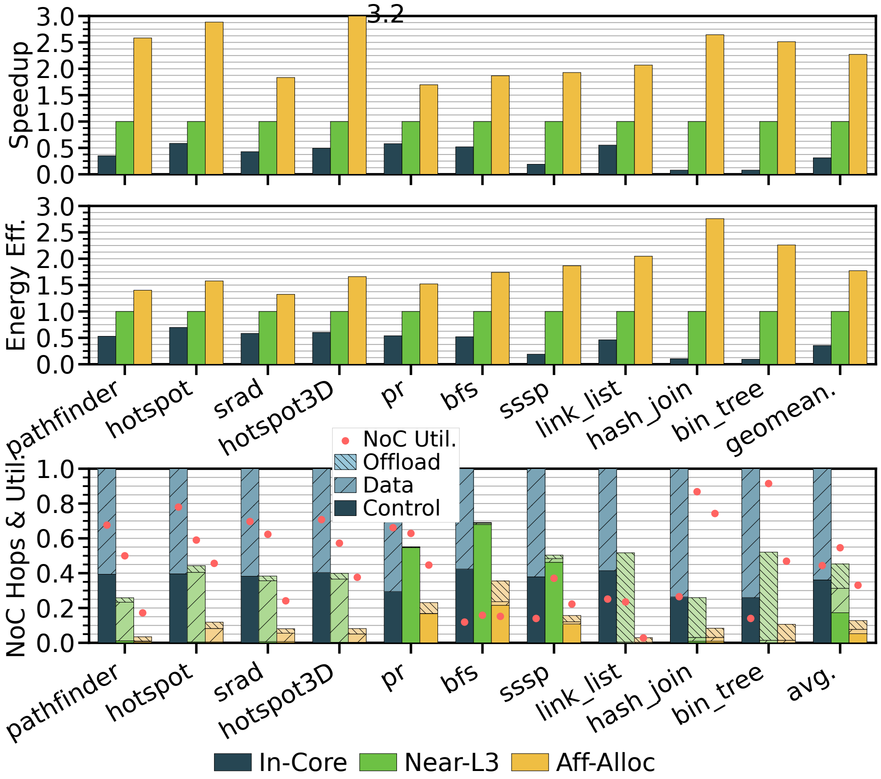


Figure 6.11: Overall Performance and Traffic Reduction

forward the operand in affine workloads (e.g. `stencil1d`), the control traffic to perform indirect remote accesses in graph workloads, as well as the stream migration traffic to chase the pointer in pointer-based data structures. Overall, affinity alloc reduces the network traffic by 72% and 87% over **Near-L3** and **In-Core** respectively, with 34% NoC utilization.

For the microarchitecture, affinity alloc only introduces a small interleave override table (IOT). Estimated with CACTI 7 [185], the IOT takes 32kB (512B per bank), and accounts for 0.07mm^2, less than 0.1‰ of the whole chip.

Bank Selection Policy Fig 6.12 shows the speedup and NoC traffic when affinity alloc employs different bank selection policies for irregular data layout, normalized to **Rnd** which randomly selects the bank to allocate. **Lnr** selects the bank in a round-robin fashion, while **Min-Hop** always picks the bank with the least distance to affinity addresses (same as setting

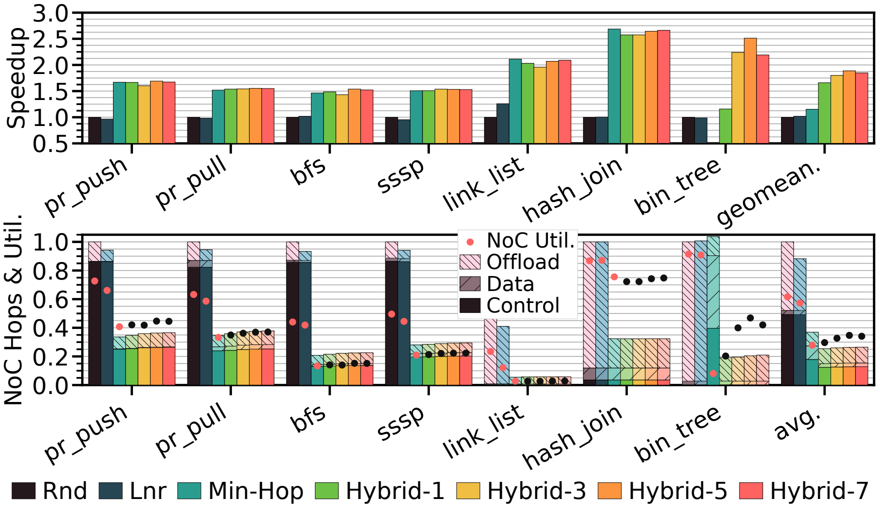


Figure 6.12: Sensitivity on Irregular Layout Policies

$H = 0$ in Eq 6.4). We also evaluate the hybrid policy that considers both affinity information and load balance with various H, labeled as **Hybrid-H**. Higher H forces the policy to favor the less occupied bank to balance the load.

As expected, **Rnd** and **Lnr** are oblivious to the affinity information and achieve similar performance. **Lnr** only outperforms **Rnd** by 25% on `link_list`, as we allocate the nodes one by one and **Lnr** allocates the node to the next bank, reducing the pointer-chasing distance (about 60% traffic reduction). However, this is not optimal compared to colocating neighboring nodes in the same bank, which eliminates the need to migrate. Also, linear allocation is less likely the case in real production scenarios, and when list nodes are inserted randomly, **Lnr** would behave the same as **Rnd**.

On the other hand, **Min-Hop** optimizes the data affinity and achieves significant speedup and traffic reduction on most benchmarks. However, since it does not consider the load balance, it may produce pathological data layout. For example, in `bin_tree` it allocates the entire tree to a single bank. Although it successfully eliminates the migration traffic (much less offload traffic in Fig 6.12), it dramatically increases the miss rate to that L3 bank and results in a huge slowdown.

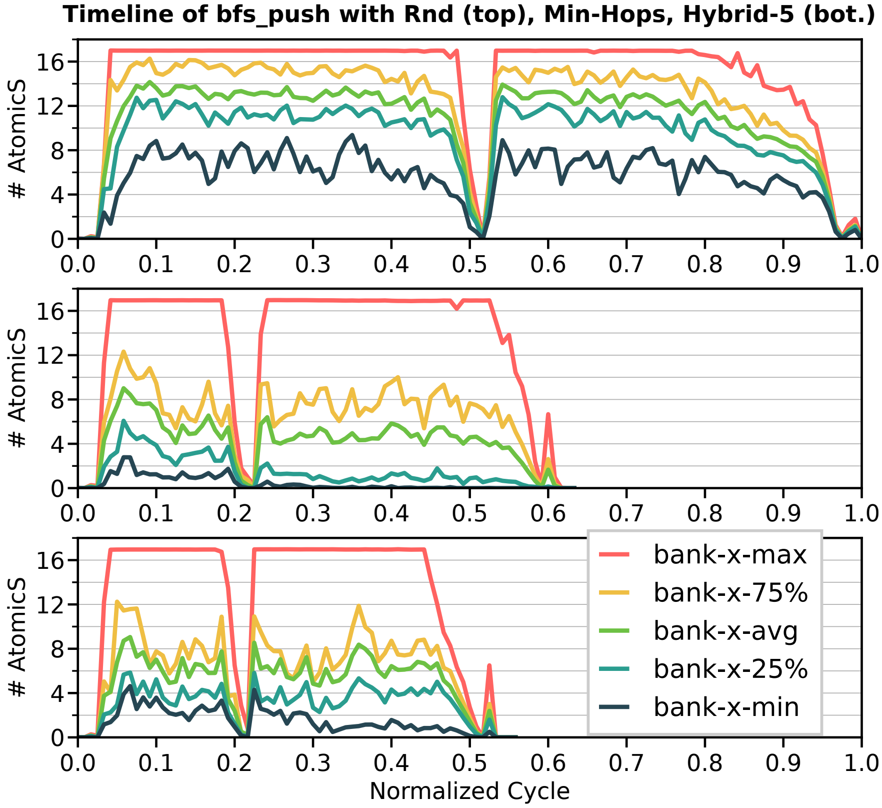


Figure 6.13: Distribution of Atomic Stream in BFS-Push

The hybrid policy **Hybrid-H** avoids such pathological cases by allocating to less occupied banks to balance the load. It also achieves better bank-level parallelism and improves the performance over **Min-Hop**. To see this, Fig 6.13 shows the timeline of number of atomic streams per L3 bank in `bfs_push` for **Rnd**, **Min-Hop** and **Hybrid-5**. We show the distribution by plotting the number of atomic streams from least to most occupied bank. For example, the 25% line indicates that 75% banks have higher occupancy. **Rnd** has higher stream occupancy, as it takes much longer for each stream to finish the indirect atomic access. **Hybrid-5** achieves better load balancing than **Min-Hop** with a higher 25% line. Overall, **Hybrid-5** achieves the highest performance with slightly more traffic, and is chosen as the default policy.

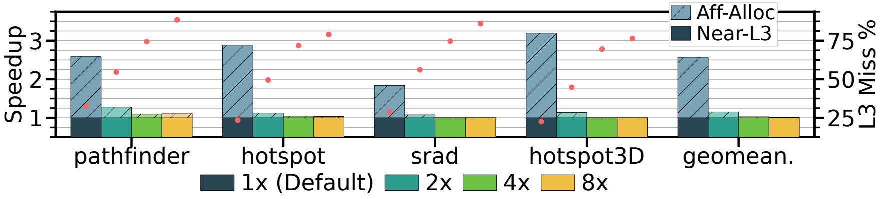


Figure 6.14: Speedup of Affine Layout on Large Inputs

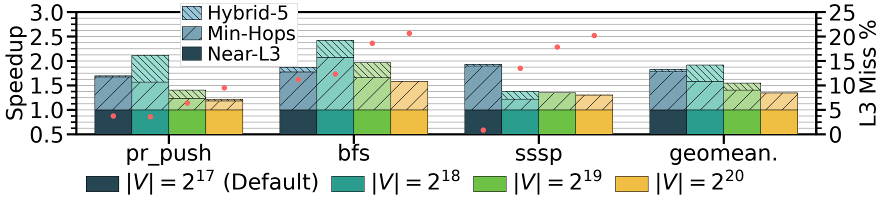


Figure 6.15: Speedup of Linked CSR on Large Graphs

Large Input Size Fig 6.14 shows the speedup and L3 miss rate of affine workloads when scaling up the input size. Since this work focuses on near-cache computing, the benefits of affinity alloc significantly drop when the working set cannot fit in the cache (¿75% L3 miss rate for 8× input size). Fig 6.15 shows the same evaluation on graph workloads. We scale up the graph by increasing the number of vertices, while keeping the average vertex degree the same. Due to the irregular access pattern, we can get some reuse on the vertex properties, leading to ¡20% L3 miss rate. Therefore, affinity alloc still yields some performance improvement for the 8× graph. When $|V| = 2^{18}$, the graph can still fit in the L3 cache for `pr_push` and `bfs`, but not for `sssp` due to extra edge weights.

The implication is that the already common optimization of tiling and partitioning for the on-chip cache becomes even more important. Also, as the on-chip cache continues to scale up (768MB on AMD EPYC 7773X [178]), the number of tiles required can be reduced (hence less overheads). This is orthogonal to this work. When there is no reuse at all on the chip, future work could also apply affinity alloc to align data in DRAM to benefit NDC techniques near the memory controller or inside DRAM.

Graph Processing

Graph processing contains heavy indirect accesses and benefits from improved data affinity provided by affinity alloc. Here we evaluate codesigning the algorithm in NDC scenarios, as well as sensitivity on graph structures.

Pushing vs. Pulling Graph processing algorithms `page_rank` and `bfs` have both push-based and pull-based implementations. These approaches have different trade-offs: Pushing (i.e. top-down) approach propagates updates to outgoing neighbors and is implemented with atomic access, while pulling (i.e. bottom-up) queries incoming neighbors and involves reduction. Near-data computing naturally supports remote atomic accesses, but suffers from indirect reduction which requires collecting operands distributed among LLC banks. On the other hand, general-purpose processors can perform efficient reduction using registers, but suffer from many coherence misses when contention on atomic accesses is high. Overall, we observe that near-data computing usually favors the push-based implementation, while in-core computing works better with the pull-based one. In our evaluation, this is the default choice for `page_rank`, in which all edges are active and processed in each iteration.

However, in `bfs`, each iteration has different characteristics and may benefit from per-iteration choices between pushing and pulling [184]. Fig 6.16 shows three key characteristics for iteration i: Visited Nodes: Total visited nodes after iteration i; Active Nodes: Visited nodes during iteration i; Scout Edges: Outgoing edges from active nodes in iteration i. All three are normalized to the total number of nodes or outgoing edges in the graph. Fig 6.17 shows the timeline of `bfs` using only pushing/pulling and a switching policy.

As expected for **In-Core**, pushing works well for the first and last few iterations, as there are few active nodes and therefore fewer coherence misses compared to the middle iterations. Iterations in the middle (Iter2, Iter3 and Iter4 of **In-Core** in Fig 6.17) favor pulling, as it avoids the overheads of coherence misses on contended vertices. More generally, the number of scout edges represents the number of pushing operations in the next iteration, and the

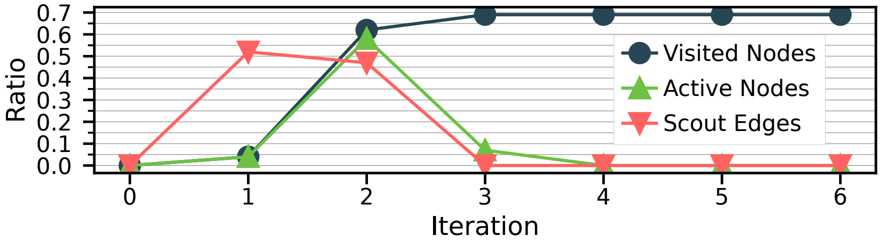


Figure 6.16: BFS Iteration Characteristic

default `bfs` implementation in GAP suite [158] switches to pulling if the ratio of scout edges exceeds a threshold.

This trade-off is different in near-data computing, as it is much cheaper to perform in-place atomic operations in L3 without the overheads of coherence misses. Affinity alloc improves the spatial locality and further reduces the overheads of remote atomic accesses. Therefore, near-data computing chooses pushing for more iterations. For example, in **Aff-Alloc** only Iter3 uses pulling in Fig 6.17, which suffers from excessive failed compare and exchange operations on visited vertices and has a much lower active node ratio compared to the scout edge ratio in the previous iteration in Fig 6.16. We adopt this insight and extend the default switching policy to estimate the chance of failed atomic operations by taking into account the ratio of visited vertices for **Aff-Alloc**:

- *Push* → *Pull*: Visited Node $>$ 40% and Scout Edge $>$ 6%.
- *Pull* → *Push*: Awake Nodes $<$ 25%.

We find this policy robust across all evaluated graphs. This study and the linked CSR format shows that NDC poses many different trade-offs that require software and data structure codesign.

Sensitivity to Node Degree One fundamental difference between affinity alloc and a conventional graph partitioning scheme is the optimization granularity. Conventional graph

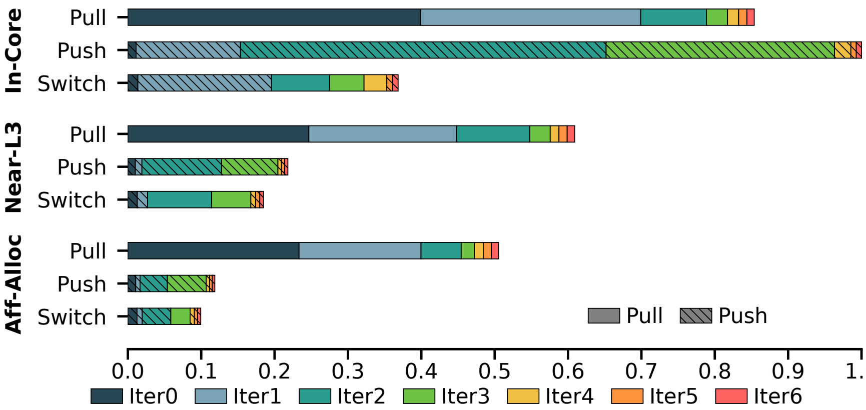


Figure 6.17: BFS Push vs. Pull Timeline

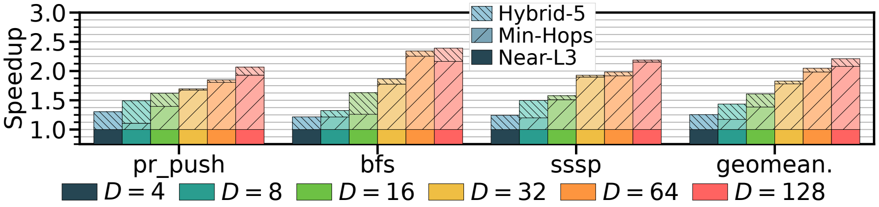


Figure 6.18: Speedup vs. Avg. Node Degree

partitioning divides the graph into a few coarse-grained subgraphs, and usually struggles for high-degree graphs. On the other hand, by co-optimizing the data structure, affinity alloc can optimize data affinity at cache line granularity and scales well with the connectivity.

To quantify this, Fig 6.18 shows the speedup of affinity alloc on various synthesized power law graphs, normalized to **Rnd**. We fix the total number of edges but change the average node degree. Affinity alloc actually achieves higher speedup on high-degree graphs (1.5× when $D = 4$ and 2.4× when $D = 128$). This is because the edge list is sorted by outgoing vertex id (as is common practice), and the longer the edge list, the more likely that outgoing vertices of edges within one cache are mapped to the same or neighboring banks. We believe affinity alloc provides a new angle to co-optimize NDC and data structures.

Real World Graphs We also evaluate affinity alloc on real-world social network graphs.

Input Graph	Type	|Vertex|	|Edge|	Avg. Degree
`twitch-gamers` [186]	Power Law	168,114	13,595,114	81
`gplus` [187]	Power Law	107,614	13,673,453	127

Table 6.4: Real World Graphs

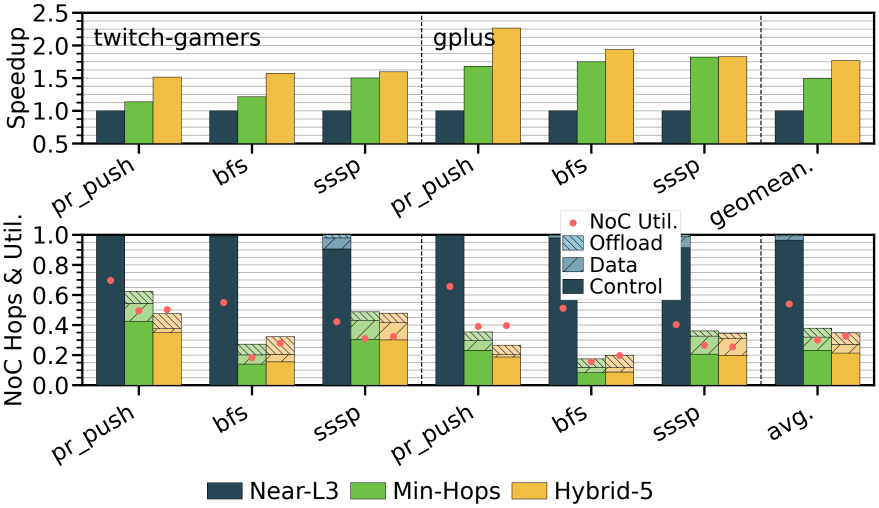


Figure 6.19: Performance on Real World Graphs

Table 6.4 lists the detailed information. These power-law graphs have a high average degree and are hard to partition. Fig 6.19 shows the speedup and traffic reduction of affinity alloc on these graphs, normalized to **Near-L3**. Overall, affinity alloc successfully optimizes the fine-grained irregular data layout, and **Hybrid-5** achieves 2.0× speedup over **Near-L3**. This clearly demonstrates the benefit of co-optimizing the data structure and affinity data layout for near-data computing.

6.6 Discussion

Dynamic Data Structures Although this work focuses on static data structures (i.e. unchanged after creation), it is an interesting direction to apply affinity alloc to dynamic data structures, especially for those that are pointer-based (e.g. trees, linked CSR). A particular example is dynamic graph processing [188, 189, 190, 191, 192] which queries evolving graphs. In this work we extend the static CSR format with pointers to provide the flexibility to support irregular layout optimization, which needs some preprocessing. However, some prior works already leverage pointer-based data structures similar to linked CSR to flexibly insert and delete from the graph [193, 194], which can naturally benefit from the improved spatial locality from affinity alloc without extra preprocessing.

Generally, if the affinity requirement changes, e.g. reinserting the tree node to a different location, the previous layout choice becomes suboptimal. If the runtime is aware of the data structure modification, e.g. via '`realloc()`', the layout could also be dynamically adjusted, or fall back to the default random layout if dynamic remapping overhead is intolerable. This is left as future work.

Fragmentation One major challenge to support dynamic allocation is to handle fragmentation. In principle, the major source of fragmentation is limiting freed space in the interleave pool to allocations with the same interleaving requirement (OS can still reclaim pages at both ends by shrinking the interleave pool). For example, considering three consecutively allocated arrays `A[]`, `B[]` and `C[]` in the same interleave pool. The free space from releasing `B[]` can only be reused for data structures with the same interleaving, as interleave pools are backed by contiguous physical addresses. However, this fragmentation was not seen in our static application set. A software solution is to compact the pool. Another possibility is to dynamically break and merge interleave pools of the same interleaving. In the above example, the single interleave pool can be split into two: one for `A[]` and the other one for `C[]`, and the free space in between can be claimed for other interleaving or normal allocations without the overhead of copying and compacting. This requires a larger interleave override table (IOT) in microarchitecture similar to prior works (e.g. RMM [182] has 32 range entries vs. 7 interleave pools in this work).

6.7 Related Work

Multicore Caching and Dynamic Data Layout Multicore caches are physically distributed, giving rise to non-uniform cache access (NUCA) [195]. Many dynamic NUCA (D-NUCA) designs have been proposed that change the data layout to reduce data movement[196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212]. Unlike affinity alloc, these designs do not offload computation near data. Rather, they move frequently accessed data closer to the cores accessing it.

Several limitations make D-NUCA schemes hard to apply to near-data computing. Early D-NUCAs treated the on-chip cache banks as a hierarchy, gradually migrating data closer to cores that access it[196, 197, 198, 202, 199, 200]. These designs require another layer of directories to locate data dynamically. As a result, most accesses still require an expensive global lookup, eliminating most of the benefit of adapting the data layout. Later D-NUCAs control data layout via the virtual memory system (i.e., page table and TLBs) so that no additional directory lookup is required[198, 201, 202, 203, 204, 205, 212, 213, 214]. These single-lookup D-NUCAs significantly reduce data movement, but can only control data layout at page granularity, which we have shown is insufficient (Fig 6.5). Hotpad [215] designs a scratchpad hierarchy for managed languages (e.g. Java), but does not optimize for data affinity among banks.

Whirlpool [177] is a D-NUCA that controls data layout via the memory allocator, similar to affinity alloc. Whirlpool uses the memory allocator to separate data into different "pools" and uses a different data layout for each pool, letting the cache separate data with different access patterns. By contrast, affinity alloc lets programmers express the *affinity* between related data and control the layout so that related data is placed at the same location.

None of these works support single-lookup for fine-grained irregular affinity, nor do they explore the benefit of co-optimizing the software to enable flexible data placement.

Near-Data Computing Various *near-data* approaches push computation into different memory hierarchy levels to avoid unnecessary data movement: partial thread migration among cores [216], near LLC [2, 31, 3, 32], on-chip network router [4], memory [13, 9, 11, 10, 8, 15, 17, 14, 5, 6, 175, 16, 137, 145, 160, 142] and storage [166, 19, 18], and even across multiple levels or substrates [30, 29, 27, 24, 25, 26]. We can think of these as *vertical* near-data computing.

The scope of near-data computing can be broadened beyond memory-hierarchy offloading to those that only have a *horizontal* dimension: i.e. those that can map tasks to different locations depending on locality. This includes works from the Swarm family of ordered-algorithm accelerators [169, 217, 148, 170, 171] that use task hints to map tasks near-data [172, 147]. Several prior multicore accelerators [179, 218, 10] and reconfigurable architectures [174, 219, 220, 221] have this capability. Most vertical near-data architectures have a horizontal aspect. We focus on improving the effectiveness of horizontal near-data, but future work could also optimize vertically across levels.

Many of these works are oblivious to the data layout and take a best effort approach to fall back to conventional execution when near-data computing is not profitable, e.g. [27, 16, 3, 5, 8, 31, 176, 26]. Other techniques require manual data placement using imperative APIs, e.g. [6, 175, 4, 137, 10, 9]. Hong et al. [145] organize the linked list nodes into the same HMC vault, and Gearbox [17] performs hybrid partition on SpMV and SpMSpV. These techniques are limited to a specific domain or affine workloads. Another line of work [176, 26] leverages the compiler to reschedule computation to optimize the arrival window in NDC. However, it left the mapping between address space and cache banks as future work. Kandemir [222] proposes loop transformation to reduce reuse distance in space for affine loops. Although it does not handle irregular accesses, it could be combined with affinity alloc to handle some tricky cases with less user intervention, e.g. transforming the loop to simplify the affinity requirements.

Affinity alloc is orthogonal to these techniques – it tackles the fundamental data layout problem in a systematic and programmable fashion. These near-data techniques could all benefit from an affinity alloc-like approach to improve data affinity. It is future work to extend affinity allocation to consider multiple memory hierarchy levels simultaneously.

Graph Processing Near-data scheduling is a prevailing optimization in graph processing accelerators [223, 141, 163, 162, 10, 224, 179, 218, 174, 148]. One use case is for vertex-centric graph processing, in which vertex-updates are scheduled near vertex properties storage [10, 224, 179, 218, 174, 148]. Our results suggest that our codesigned linked CSR format plus affinity alloc would be effective for them.

6.8 Summary

This chapter systematically addresses the data layout problem in NDC by constructing a clean layered design across the system. The application only needs to specify the essential affinity information with the extended allocator interface, and the runtime can automatically optimize data affinity and load balance. More importantly, affinity alloc opens up new design space to co-optimize data structures with data affinity. This is a first but critical step to revisiting many tradeoffs and realizing the full potential of the near-data computing paradigm, where computation is *truly* near the data.

CHAPTER 7

In-/Near-Memory Computing Fusion

While our high-level stream abstraction exposes rich program semantics, it remains neutral to the underlying microarchitecture details. This makes it possible to leverage streams as a unified abstraction to fuse multiple near-data computing paradigms. While previous chapters leverage the L3 stream engine and the remote core as the computing substrate, the emerging in-SRAM computing using bitlines is an interesting alternative. However, key challenges from its unique execution model remain unsolved: automated parallelization, transparently orchestrating data transposition/alignment/broadcast for bit-serial logic, and mixing in-/near-memory computing. Most importantly, the solution should be programmer-friendly and portable across platforms.

In this chapter, we propose an execution model and intermediate representation (IR) that enables hybrid CPU-core, in-memory, and near-memory processing. Our IR is the tensor dataflow graph (tDFG), which is a unified representation of in-memory and near-memory computation. The tDFG exposes tensor-data structure information so that the hardware and runtime can automatically orchestrate data management for bit-serial execution, including runtime data layout transformations. To enable microarchitecture portability, we use a two-phase, JIT-based compilation approach to dynamically lower the tDFG to in-memory commands.

Our name our approach infinity stream, as it enables in-/near-memory fusion. Evaluated on a cycle-accurate simulator, across data-processing workloads with fp32, it achieves 2.6× speedup and 75% traffic reduction over a state-of-the-art near-memory computing technique,

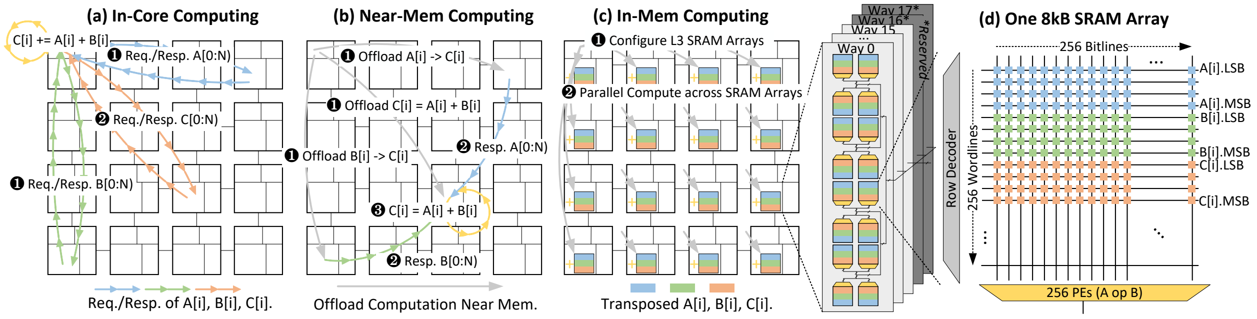


Figure 7.1: Overview of In-Core/Near-Mem/In-Mem Computing Paradigms

with 2.4× energy efficiency.

Organization §7.1 gives background on in-memory computing and overviews our approach, followed by the execution model and tDFG IR in §7.2. §7.3 details the runtime and dynamic compilation, with the μarch in §7.4 and limitations in §7.5. Methodology and evaluation are in §7.6 and §7.7, and related work is in §7.8.

7.1 Background and Overview

Here we overview the three computing paradigms with a simple vector addition example. This characterizes in-memory computing and its challenges, which motivate this work.

Near-Memory Computing

Conventional systems adopt a core-centric view: all computation is centralized in the core, with data fetched from the memory subsystem. Fig 7.1(a) shows a tiled multi-core system. Each tile contains a core with a private L1/L2 and a shared L3 cache bank, and is connected by a mesh network-on-chip (NoC). To perform `C[i]=A[i]+B[i]`, the core issues multiple requests to fetch `A[i]` and `B[i]`, as well as writing back `C[i]`. Vectorization and multi-threading can be used to exploit the massive data parallelism in this example. One major overhead here is the unnecessary data movement, as all three arrays `A[]`, `B[]` and `C[]` have no reuse at all. Techniques like prefetching and cache bypassing can only partially help, as the data movement is inevitable and incurs a high energy cost. Such overheads are only going to be more severe as the system scales up and the data grows.

Near-Memory Computing To fundamentally eliminate unnecessary data movement, near-memory computing moves computation closer to the data, and has been applied in many contexts: e.g. near on-chip SRAM [31, 3], within the NoC [4, 16], near memory controller [27, 14, 137]. They also offload computation at different granularities from coarse-grained kernel-level [5, 159, 175, 160, 161, 163] to fine-grained short instruction sequences [14, 4, 16].

Near-Stream Computing For the near-memory computing baseline, we use near-stream computing [31], which offloads long-term memory accesses (i.e. streams) with computations near the L3 cache. In Fig 7.1(b), the memory accesses are decoupled into three streams `A[i]`, `B[i]`, `C[i]`, and offloaded to the shared L3 banks where the data resides. Stream `A[i]` and `B[i]` directly forward their data to stream `C[i]`. Stream `C[i]` coordinates with the remote CPU core to perform SIMD ops on a spare thread, and then writes directly to L3. This significantly reduces the data traffic and control overheads.

Bit-Serial In-Cache Computing

Near-L3 approaches still read the data out from the L3 SRAM arrays, hence are still bound by the L3 cache's bandwidth. To fully unlock the massive potential data parallelism, in-memory computing moves the computation *inside* SRAM arrays. For this work, we assume the same compute SRAM technology as Neural Cache [20].

In Fig 7.1(c), SRAM arrays are configured to add `A[i]` and `B[i]` in *parallel* and directly write back to `C[i]`, with no sequential reads and writes at all. Fig 7.1(d) demonstrates how in-memory computing works in one 8kB SRAM array with 256 wordlines (row) and 256 bitlines (column). Specifically, it requires the data to be transposed and bit-serial logic.

Transposed Data Layout In Fig 7.1(d), array elements (4 bits each) are transposed from a horizontal layout across columns to a vertical layout on the same column. E.g. the least significant bit (LSB) of `A[0]` is stored in the cell indexed by wordline 0 and bitline 0, and the most significant bit (MSB) of `A[0]` by wordline 3 and bitline 0.

Bit-Serial Compute In-memory computing leverages bit-serial logic to compute the result. This requires operands to be aligned in the same column. In the example in Fig 7.1(d), `A[i]`, `B[i]`, and `C[i]` are all placed in the same bitline. To start the computation, we activate the wordlines of `A[i].LSB` and `B[i].LSB` at the same time, and the 256 PEs perform the bit operation on the sensed bit (e.g. `AND` for carry, `XOR` for addition). The PEs have cells holding intermediate results (e.g. carry of addition). The result bit is then written back to `C[i].LSB` by activating wordline 8 with the write signal. This process repeats to compute the result one bit at a time (hence "bit-serial"). It takes $O(n)$ cycles to perform integer addition and $O(n^2)$ for integer multiplication, where n is the data type width. However, this is amortized by the massive parallelism it provides.

Max System Speedup Assuming a 64-core system with 16-way 2MB L3 banks (total 128MB) and 16 256×256 SRAM arrays/way, the peak throughput of `int32` addition is:

$$\begin{aligned} T &= N_{bank} \times N_{way} \times N_{array/way} \times N_{bitline} / Latency^{1} \\ &= 64 \times 16 \times 16 \times 256/32 = 131072 \text{ ops/cycle} \end{aligned} \tag{7.1}$$

Assuming each baseline core can issue one 512-bit vector op per cycle ($64 \times 16 = 1024$ ops/cycle), in-memory provides 128× peak speedup. Fig 7.2 shows the speedup of two microbenchmarks with various input sizes on the baseline (AVX-512 and 1 or 64 OpenMP threads), near-L3, and in-L3 computing using bit-serial logic. We assume data is cached in L3 and already transposed for in-memory computing. in-L3 computing usually favors larger

[1] We adopt the integer addition from [21]. System params in §7.6. See In-/Near-Memory Computing [225] for more details, and §7.8 for related works.

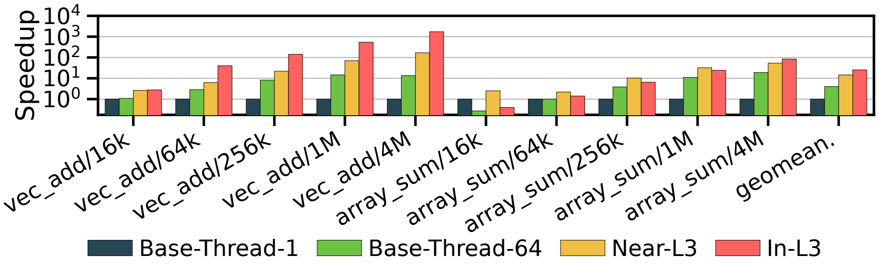


Figure 7.2: Speedup of Different Paradigms (Fp32)

input sizes as they amortize the overhead of bit-serial operation. Despite this, in-L3 achieves the best performance for `vec_add` across all input sizes. With 4M elements, it achieves 21× over Near-L3, making it a promising approach to exploit the available data parallelism.

Infinity Stream Approach Overview

There are still barriers to the broad adoption of in-memory computing. An ideal in-memory system would be programmer-transparent, compatible with existing core-centric and near-data execution without adding much overhead, and also preserve program compatibility with future microarchitectures. No existing in-memory system has achieved all three due to the challenges of the unique paradigm. Here, we overview the challenges and our approach.

Automated Orchestration Bit-serial logic requires transposing large arrays, managing on-chip space, and enforcing bitline alignment. A suitable data layout, tiling, and explicit reuse are critical to reducing data traffic. Also, distributing computation to bitlines requires massive vector parallelism. Ideally, this orchestration would be done without any programmer involvement. Thus, the system must automate this management and ease integration with conventional code. The key challenge is expressing sufficient information to the hardware and software runtime.

Our approach: We develop a program representation called the tensor dataflow graph (tDFG). The tDFG operates over tensors with explicit data-parallel semantics, and represents inter-data structure alignment with the concept of a global lattice space. Reuse can be determined precisely, and the tDFG can be annotated with hints about optimal tiling patterns. The tDFG is embedded as an extension to a traditional ISA, and gives the runtime sufficient information to make good decisions.

Fused In-/Near-Memory Computing Sometimes it is better to split the work between in-/near-memory computing. E.g. an in-memory reduction to produce partial results, which are reduced to the final value by near-memory computing; or a phase with both irregular and regular data structures, where only the latter is suitable for in-memory. This requires a unified execution model and low-overhead hardware implementation. As suggested by Fig 7.2, in-memory struggles with small input sizes. Also, many code patterns like irregular control and memory (e.g. `A[B[i]]`) are only potentially suitable for *near*-memory. This motivates both a runtime selection between in-/near-memory computing, and a fused in-/near-memory paradigm.

Our approach: The tDFG can express both in-memory and near-memory opportunities in a unified representation. This generalizes the near-data approach from near-stream computing [31]. At runtime, the system decides the offload target (in-/near-memory) based on data size and access behavior. One key hardware feature is to integrate the transposed data layout with the coherence protocol to allow data communication between the two paradigms.

Portability High-performance in-memory code requires exploiting both low-level hardware details (e.g. # of bitlines/array, SRAM-level instructions) and runtime values, e.g. array dimensions, compute constants. Thus, it is difficult for a single low-level binary to be compatible with all software parameters and future microarchitectures without sacrificing performance.

Our approach: We take a just-in-time (JIT) approach, with the tDFG playing a similar role to PTX virtual assembly for CUDA GPUs. A JIT runtime is in charge of quickly lowering the tDFG "virtual" ISA into in-memory computing commands and managing the

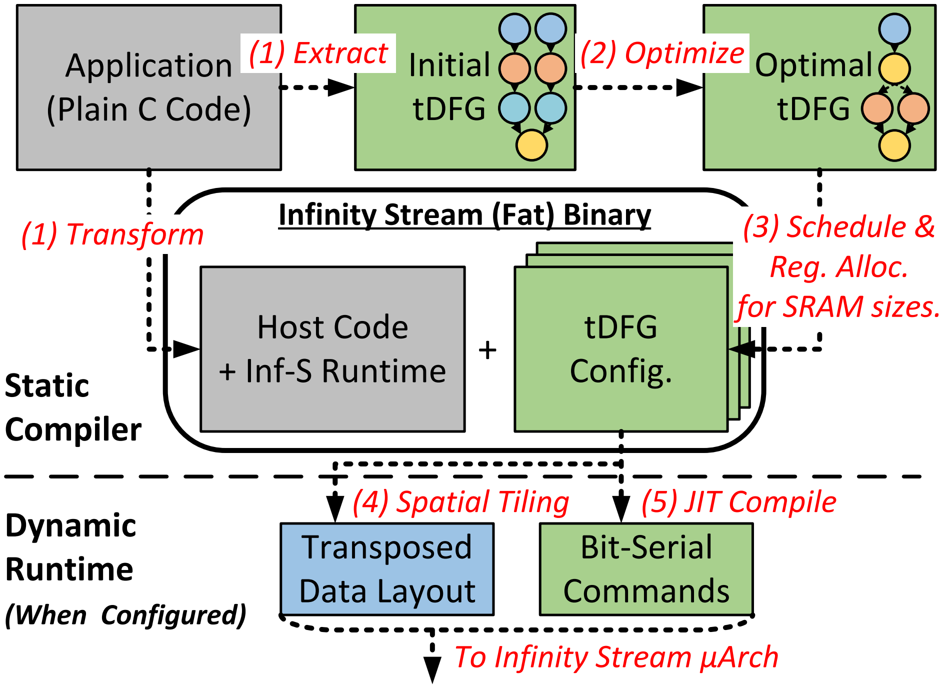


Figure 7.3: Infinity Stream Workflow Overview

transposed data layout. This requires carefully splitting the job between the compiler and the runtime to maintain compatibility while keeping JIT overheads reasonable.

Programmability Ideally, the system should be easy to program, without programmers writing multiple code versions, worrying about data orchestration, and switching between paradigms. This requires a unified compiler and ISA abstraction, as well as a flexible runtime library and microarchitecture support.

Our approach: The tDFG is constructed purely by the compiler using plain C code. The algorithm and program transformations (e.g. inner vs. outer product) can of course affect the performance, so we discuss programming implications in §7.2. Overall, infinity stream requires only minimal programmer intervention.

Infinity Stream Workflow Overview Fig 7.3 summarizes the overall workflow: our static compiler first extracts an initial tDFG from plain C code and optimizes it for compute reuse and less data traffic. The optimal tDFG is scheduled for common SRAM sizes (we use 256×256 and 512×512). This generates a fat binary with multiple tDFG configurations, which reduces the complexity of JIT compilation. At runtime, when an infinity stream region is configured, the runtime dynamically decides the transposed data layout with tiling based on the data size and hardware parameters. The matched version of tDFG is JIT lowered into bit-serial commands. The infinity stream μarch transposes the data and executes the commands to perform in-memory computing.

7.2 Infinity Stream Abstraction

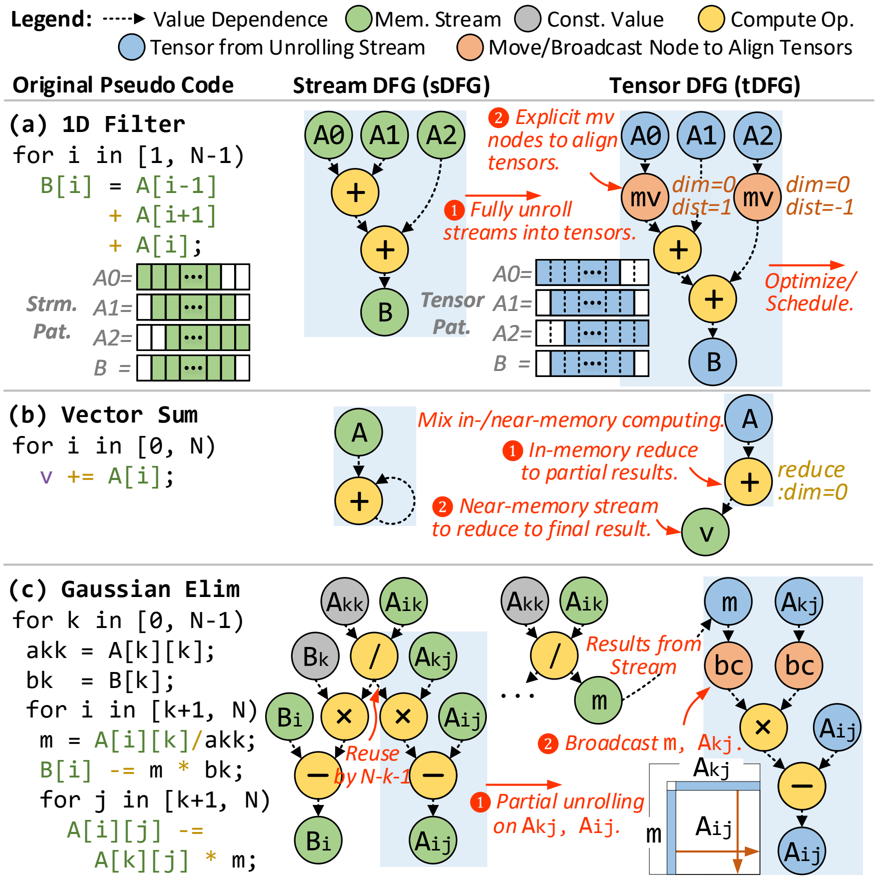
This section shows how the proposed abstraction captures the unique properties of in-memory computing to enable helpful optimizations while simplifying programming complexity.

Stream Dataflow Graph

We first extract the stream dataflow graph (sDFG) from the program, which embeds memory access patterns as *streams* with associated near-stream computations. We leverage the sDFG as the foundation and later extend it to support in-memory computing.

Stream The compiler decouples access patterns into streams. E.g. Fig 7.4(a) contains three load streams `A[i-1]`, `A[i]`, `A[i+1]`, and one store stream `B[i]`, with linear access patterns. Streams may be extracted from outer loops if the access pattern is supported. Irregular access patterns (e.g. `A[B[i]]` and `p=p->next`) are also streams but are inefficient for pure in-memory computing.

Near-Stream Computation Computation can also be associated with streams. E.g. in Fig 7.4(b) the reduction is associated with stream `A[i]`. Although the operation is applied to all elements, streams still implicitly define the access order and preserve sequential semantics. In hardware, each stream (and associated computation) can be independently moved near



```
(a) 1D Filter
for i in [1, N-1)
  B[i] = A[i-1]
       + A[i+1]
       + A[i];
```

```
(b) Vector Sum
for i in [0, N)
  v += A[i];
```

```
(c) Gaussian Elim
for k in [0, N-1)
 akk = A[k][k];
 bk  = B[k];
 for i in [k+1, N)
  m = A[i][k]/akk;
  B[i] -= m * bk;
  for j in [k+1, N)
    A[i][j] -=
      A[k][j] * m;
```

Figure 7.4: Examples of Infinity Stream Abstractions

the L3 if more locality there.

Stream Dataflow Graph Streams and near-stream computations form the stream DFG. Streams can have dependencies: data from the outer loop can be reused by the inner loop, e.g. in Fig 7.4(c) where the value m is reused (N-k-1) times.

Tensor Dataflow Graph

Intuition In-memory computing requires unrolling computation across all bitlines. Inspired by this observation, if the domain of the stream is a hyperrectangle (i.e. N-dimensional rectangle) of the data structure, we can fully unroll the stream into a *tensor*. We can then

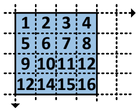
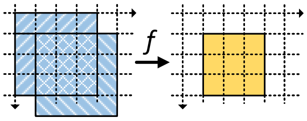
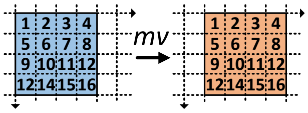
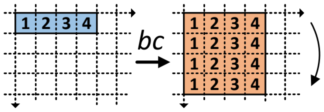
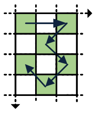
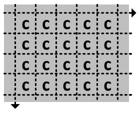
tDFG Node	Lattice Space Representation	Semantic
A = **tensor**(ptr$_{data}$, p$_0$, q$_0$, … p$_{N-1}$, q$_{N-1}$)	 $[p_0,q_0)\text{x}\ldots\text{x}[p_{N-1},q_{N-1})$ E.g., A=[0,4)x[0,4)	A N dimension ***hyperrectangle*** set of **data elements** in lattice space.
A$_o$ = **cmp**(f, A$_0$, … A$_M$)	 A$_o$=∩A$_i$	Apply an element-wise ***f(A$_0$,…)*** to the ***intersection*** of input tensors. ***Assume no inter-elem. order.*** @ret: a tensor ***A$_o$***
A$_o$ = **mv**(A, dim, dist)	 1 2 3 4 5 6 7 8 9 10 11 12 12 14 15 16 mv 1 2 3 4 5 6 7 8 9 10 11 12 12 14 15 16 *dim=0* *dist=1*	Move the input tensor ***A*** by ***dist*** in dimension ***dim***. @ret: a tensor ***A$_o$***
A$_o$ = **bc**(A, count, dim, dist)	 *count=4* *dim=1* *dist=0*	Broadcast tensor ***A count*** times in dimension ***dim*** with offset ***dist***. @ret: a tensor ***A$_o$***
v|A = **strm**(acc_pat)	 *Access Pattern:* $\underbrace{i_0^{N_i} j_0^{N_j} k_0^{N_k}}_{\text{any order}} w_0^{\text{size}} B[A[i][j][k]+w]$	***Sequentially accesses*** the array using the access pattern. @ret: normal values ***v*** | a tensor ***A***
C = **const**(c)	 *Value c broadcasted* *to all lattice cells.*	An infinite tensor ***C*** with compile-/run-time constant ***c*** at ***all lattice cells***.

Figure 7.5: tDFG Node Semantics

reformulate the computation as a dataflow graph where the operands are tensors; we call this the tensor DFG (tDFG). Fig 7.4 shows three example tDFGs, and Fig 7.5 summarizes all types of tDFG nodes. We now define the key concepts and semantics of the tDFG.

Global Lattice Space A key feature of the tDFG is the ability to reason about the relative location of different tensors in memory, so that data can be aligned at the bitline level. To enable abstract reasoning about relative locality, we introduce a global lattice space to the tDFG. All tDFG tensors are positioned on an N-dimension global lattice space (its dimensionality is that of the data structure with the highest dimension), shown as the dashed grid in Fig 7.5. Each lattice cell can hold an arbitrary number of data elements. At runtime, cells are mapped to physical locations, e.g. SRAM bitlines. More importantly, the lattice space serves as a homogeneous coordinate system to abstract away the complex underlying hardware hierarchy, including bitlines, SRAM arrays, banks, NoC, etc. This helps keep the tDFG abstraction portable across platforms.

Tensor As in Fig 7.5, a tDFG tensor is a hyperrectangle set of data in the lattice space, denoted by $[p_0, q_0) \times ... \times [p_{N-1}, q_{N-1})$ where p_i and q_i are the start and end coordinates in dimension i. Each data element of a tensor resides in its own lattice cell. An N dimensional array is by itself a tensor with $p_i = 0, q_i = S_i$ where S_i is the array size on dimension i. Unlike streams, tensors do not imply a temporal sequential order but are fully expanded in the lattice space.

Compute with Tensors A compute node takes one or more input tensors, applies the computation to a domain which is the intersecting hyperrectangle (see Fig 7.5), and produces an output tensor. The tDFG uses a static single-assignment form (SSA), i.e. nodes always produce a new tensor without overwriting existing ones. There are two key characteristics of tensor computation:

- **Data Parallelism:** Since tensors are fully unrolled, the tDFG does not assume an elementwise order within one tensor computation, exposing massive data parallelism.
- **Data Alignment:** Tensor computation requires operand elements from different tensors to be exactly aligned within the same lattice cell. This captures the data alignment requirement for in-memory computing.

Explicit Tensor Alignment We introduce two types of node in the tDFG to facilitate explicit tensor alignment, which is crucial to optimize and compile data movement for in-memory computing:

- **Move:** A move node (`mv`) in Fig 7.5 shifts a tensor along a dimension by a certain distance. E.g. in Fig 7.4(a), tensor `A[0,N-2)` is moved to the right by 1 to align with `A[1,N-1)`.

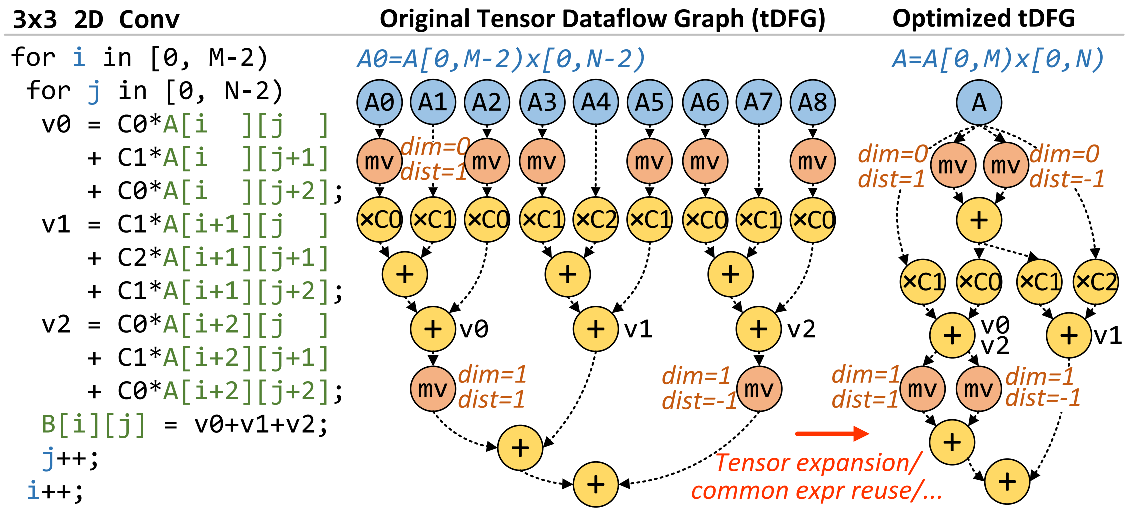
- **Broadcast:** To capture reuse spatially, a broadcast node (`bc`) in Fig 7.5 broadcasts a small reused tensor along the reuse dimension to align with the larger tensor. In Fig 7.4(c) `A[k,k+1)x[k+1,N)` is broadcast downwards to align with `A[k+1,N)x[k+1,N)`.

Global Bounding Hyperrectangle Due to the finite hardware resources, not every lattice cell has a valid physical location. we define the global bounding hyperrectangle as the minimal one that contains all involved data structures. semantically, data elements outside the bounding hyperrectangle have undefined values, so data moved or broadcasted outside is discarded. For now, we implicitly assume all data structures are aligned to the origin, but this can be relaxed to placing the array anywhere in the lattice.

Optimizing tDFG We leverage equality graphs (e-graphs) [226, 227] to search for an optimized tDFG. E-graphs are a representation of all possible rewrites to a graph in a compact form, which leverages equality relationships between different rewrites. To construct an e-graph for our case, we start from the initial tDFG, then repeatedly grow the e-graph by applying rewrites and maintaining equivalence points between them. The final tDFG selection is based on architecture-informed cost metrics (e.g. estimated latency of move vs. compute node), and can be exhaustive or terminated early to reduce compile time. Fig 7.6 shows the initial and optimized tDFG for Fig 7.4(c). Besides the basic associative, commutative, and distributive rules, two transformations are widely applicable (see the Appendix for a full list of transformation rules):

- **Tensor Expansion**: We can merge two `mv`s with the same distance and dimension but on slightly different patterns. In Fig 7.6, $\texttt{A}_0$`:[0,M-2)x[0,N-2)` and $\texttt{A}_3$`:[1,M-1)x[0,N-2)` are both shifted to the right by 1 and can be merged into one `mv` on the expanded tensor `[0,M-1)x[0,N-2)`.

- **Reuse Common Comp.**: We can also reuse common computations. In Fig 7.6, instead of multiplying by $C0$ four times, we can reuse the result by shifting it to where it is needed in the lattice.



```
for i in [0, M-2)
 for j in [0, N-2)
  v0 = C0*A[i  ][j  ]
     + C1*A[i  ][j+1]
     + C0*A[i  ][j+2];
  v1 = C1*A[i+1][j  ]
     + C2*A[i+1][j+1]
     + C1*A[i+1][j+2];
  v2 = C0*A[i+2][j  ]
     + C1*A[i+2][j+1]
     + C0*A[i+2][j+2];
  B[i][j] = v0+v1+v2;
  j++;
 i++;
```

Figure 7.6: Example of Optimized tDFG

Hybrid In-/Near- Memory

tDFG is also general and flexible to support hybrid in/near memory execution by embedding streams.

Embedding Streams in tDFG Some streams/ops in the tDFG are not unrolled into tensors, e.g. alias, non-hyperrectangle accesses, etc. Keeping streams in the tDFG enables data to be read or written in a strided affine pattern or an indirect pattern, providing a better setup for tensor computation (e.g. a stream performs an indirect access and lays out the data in a tensor format). We allow up to three dimensions for affine access and dependent one-level indirect access (see the access pattern in Fig 7.5). A stream node can produce:

- **Normal Values:** Load and reduce streams generate normal values (non-tensor) consumed by the core or other streams. E.g. the reduction in Fig 7.4(b) is split into two nodes: a tensor compute node to perform partial in-memory reduction, and a stream node to perform the final reduction, as in-memory computing is inefficient for the final rounds.

- **Tensor Values:** Store streams produce a new tensor with the bounding hyperrectangle of all touched lattice cells. Semantically, this can be as large as the entire accessed array, e.g. an indirect stream updates a subset of the elements. However, in implementation, this is just updating an existing tensor and does not allocate a new one. In Fig 7.4(c), stream `B_i` is not unrolled due to low parallelism, and stream `m` writes the division result into a tensor `m`, which is later consumed by in-memory computing.

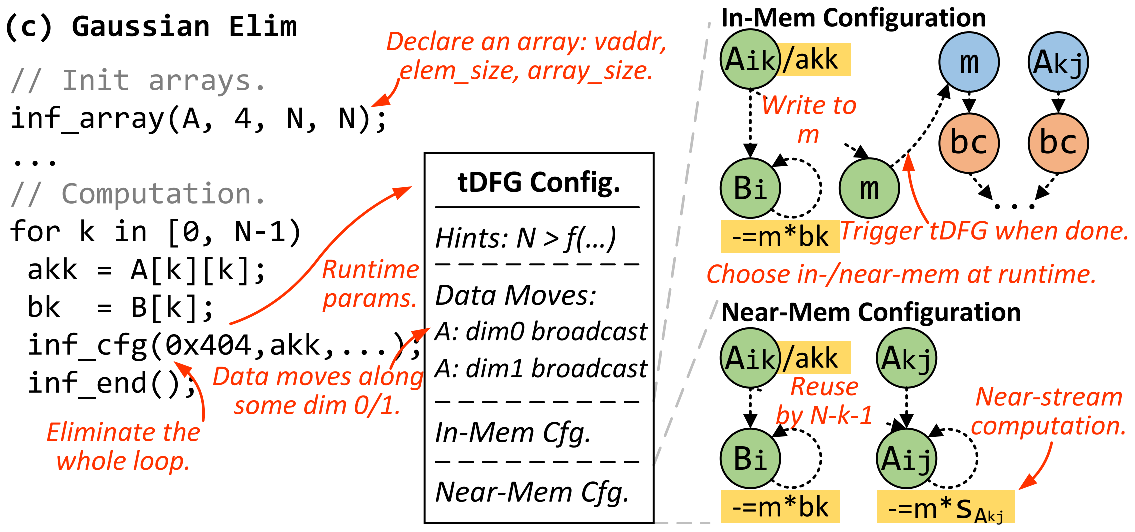
Supporting Irregularity Hybrid in-/near-memory execution enables infinity stream to handle some forms of irregularity, i.e. streams in tDFG can have irregular access patterns (e.g. A[B[i]]). For example, in `kmeans`, in-memory computes the closest centroid for each point using tensor operations, while near-memory performs the indirect update to recalculate centroids' coordinates. For future work, the tDFG can also be extended with control flow and predication to handle control irregularity.

ISA Interface

Both the sDFG and tDFG for each relevant program region are encoded in the binary, to enable a dynamic choice between near-memory and in-memory respectively. Fig 7.7 shows the compiled Fig 7.4(c) with both DFGs and data layout hints.

Infinity Stream Configuration The `inf_cfg` instruction marks the beginning of infinity stream regions, and passes in the runtime parameters (e.g. constant values). This triggers the runtime library to read in the configuration and configure the microarchitecture (details in §7.3 and §7.4). As in prior work [31], near-stream computations are compiled into conventional functions in the native ISA. A pointer to this function is stored in the sDFG.

Layout Hints for Tiling We add layout hints into the configuration to help the runtime quickly make good decisions about tiling: e.g. which dimensions the array would be shifted along (favoring tiling along those dimensions), as well as which arrays are used for the same

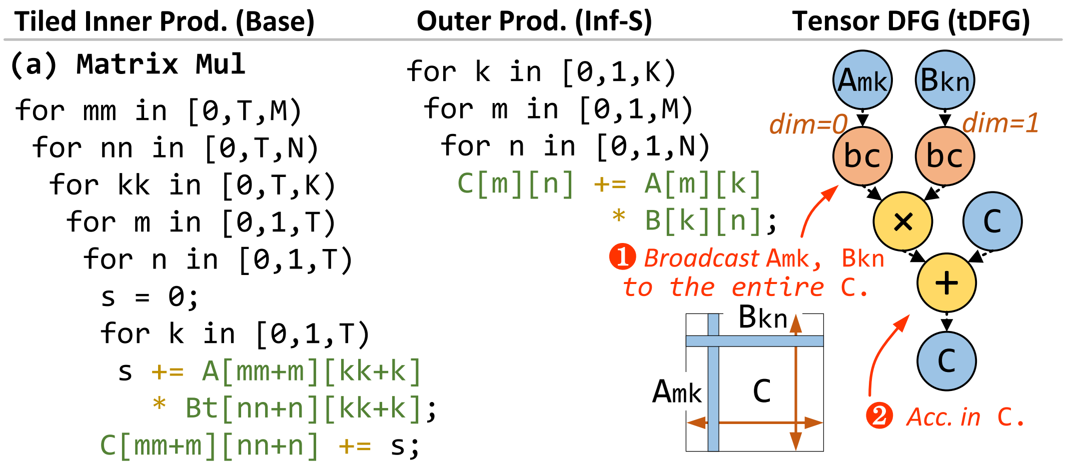


```
(c) Gaussian Elim
// Init arrays.
inf_array(A, 4, N, N);
...
// Computation.
for k in [0, N-1)
 akk = A[k][k];
 bk  = B[k];
 inf_cfg(0x404,akk,...);
 inf_end();
```

Figure 7.7: Example of Compiled Infinity Stream Program

computation (and should be bitline-aligned). The compiler generates the layout hints by analyzing the tDFG's data movement patterns. The runtime also requires the array sizes, which are passed in using the `inf_array` API. Fig 7.7 demonstrates using `inf_array` to declare a 2D array `A[N][N]`, where the infinity stream configuration defines that array `A` is broadcast in both dimensions. The runtime combines this information and picks a suitable data layout to reduce the traffic (see §7.3). Currently, we manually insert `inf_array` calls in the initialization phase.

tDFG Backend Compilation To generate a tDFG configuration, the backend compiler serializes the tDFG and allocates values to wordlines (once for each SRAM array size in the fat binary). In this work, we use a straightforward approach of scheduling instructions in topological order, and using a local register allocation scheme [228]. Though there are few effective registers (e.g. 8 32-bit registers in a 256-wordline SRAM array), no register spilling was observed in the studied workloads. Fusing multiple physical SRAM arrays into a larger virtual array with more registers is possible but left for future work.



```
(a) Matrix Mul
for mm in [0,T,M)
 for nn in [0,T,N)
  for kk in [0,T,K)
   for m in [0,1,T)
    for n in [0,1,T)
     s = 0;
     for k in [0,1,T)
      s += A[mm+m][kk+k]
        * Bt[nn+n][kk+k];
     C[mm+m][nn+n] += s;
```

```
for k in [0,1,K)
 for m in [0,1,M)
  for n in [0,1,N)
   C[m][n] += A[m][k]
            * B[k][n];
```

Figure 7.8: Programming GEMM for Infinity Stream

Programming Infinity Stream

Due to its special execution model, programmers face different trade-offs when programming an in-memory system, with tiling and dataflow being the two major design choices.

Tiling Since in-L3 computing flattens the memory hierarchy, it becomes unnecessary to tile for L1/L2 caches at the programming interface. The runtime will handle the tiling across SRAM arrays using microarchitecture support. E.g., Fig 7.8 shows the baseline 2-level tiled code for matrix multiplication `mm`, while infinity stream's implementation has no tiling with only 3 loop levels.

Inner vs. Outer Product Another critical design choice is the dataflow. In-core computing usually favors inner product as it accumulates the result in the register (see Fig 7.8). However, as in Fig 7.2, in-memory computing does not handle reduction well as the data parallelism is halved after each round of reduction, and prefers outer product to convert the reduction to element-wise operations. In Fig 7.8, during each round of `k`, one column of `A[]` and one row of `B[]` is broadcast to the entire `C[]`, followed by multiplication and accumulation. We evaluate both dataflow choices in §7.7.

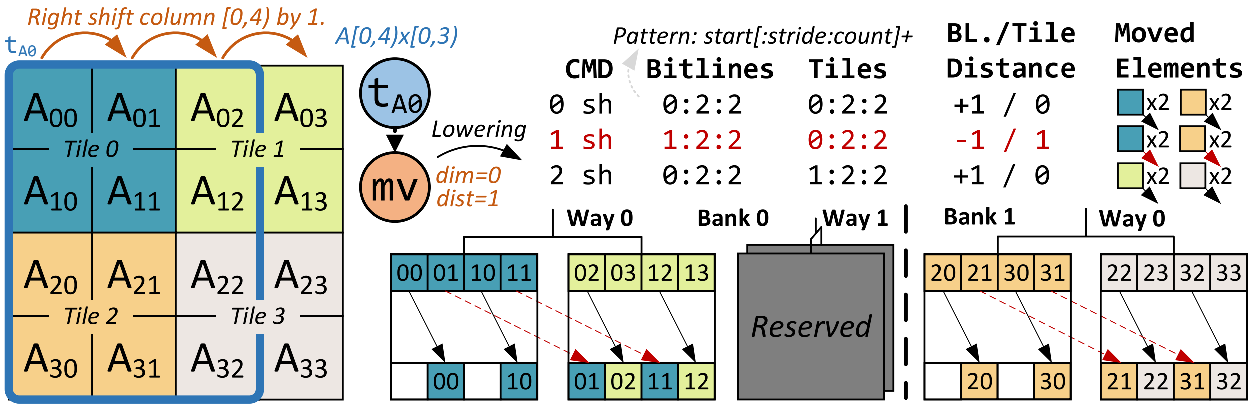

Right shift column [0,4) by 1.
t_{A0}
A[0,4)x[0,3)
A_{00} A_{01} A_{02} A_{03}
Tile 0
Tile 1
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
Tile 2
Tile 3
A_{30} A_{31} A_{32} A_{33}
t_{A0}
mv
Lowering
dim=0
dist=1
Pattern: start[:stride:count]+
CMD Bitlines Tiles BL./Tile Distance Moved Elements
0 sh 0:2:2 0:2:2 +1 / 0 x2 x2
1 sh 1:2:2 0:2:2 -1 / 1 x2 x2
2 sh 0:2:2 1:2:2 +1 / 0 x2 x2
Way 0
Bank 0
Way 1
Bank 1
Way 0
00 01 10 11
00 10
02 03 12 13
01 02 11 12
Reserved
20 21 30 31
20 30
22 23 32 33
21 22 31 32

Figure 7.9: Moving a Tensor in Tiled Layout (View in Color)

Best Practice Programmers should choose outer product or a similar dataflow that exposes more parallelism for inner loops and move reduction to outer loops. Also, there is no need to tile for private caches as in-memory computing is performed at L3. As in standard practice, programmers should still tile for L3 to provide a suitable working set for in-memory computing.

7.3 Runtime Support

The tDFG is neutral to hardware details and input sizes to maintain compatibility. Instead, a runtime library manages the transposed data layout, lowers the tDFG into in-memory commands, and decides between in-/near-memory computing, described as follows.

Transposed Data Layout

The transposed data layout is left to runtime as it requires information that is usually unavailable at compile time, e.g. input sizes, SRAM array sizes, NoC bandwidth, etc.

A trivial data layout would treat the data structure as a 1D array and map elements to contiguous bitlines. However, tensors are often shifted/broadcast along higher dimensions.

Therefore, to reduce data traffic across SRAM arrays, the data layout within an SRAM is modified through tiling. Here, a *tile* is defined by the data dimensions mapped to one SRAM array. In Fig 7.9 we consider a 4-bit-wide SRAM array, where a 4x4 2D software array is split into 4 2x2 tiles, and mapped to SRAM arrays (some SRAMs belong to ways reserved for conventional cache). We only transform the data layout through tiling at the SRAM array level, as it captures most of the traffic reduction benefits, and keeps the mapping between physical address and bitlines simple. Applying further data-layout tiling at a coarser level could further reduce data traffic.

Tiling Constraints Assume an N-dimensional $S_0 \times ... \times S_{N-1}$ array with L elements per cache line, B bitlines per SRAM array and W SRAM arrays per L3 bank used for in-memory computing. The tile size $T_0 \times ... \times T_{N-1}$ must ensure that:

1. $\prod_{i=0}^{N-1} T_i = B$: Each tile occupies all bitlines in one SRAM array. This simplifies the logic for intra-tile data movement.

2. $T_0 \times W \bmod L = 0$: For dimension 0 (continuous in address space), tiled elements at each L3 bank ($T_0 \times W$) aligns with elements per cache line (L). This ensures that each line is mapped to only one L3 bank.

The runtime gets the array's element size and shape from the `inf_array` API, and searches for a valid tile size meeting the constraints. If none is found, the array is not transposed and in-memory computing is disabled. Notice that the array size is not required to align to tile size; boundary tiles with unused bitlines require special handling (see §7.3 and §7.4). In addition, it checks that the array's innermost dimension aligns to the cache line ($S_0 \bmod L = 0$). Along with constraint 2, this guarantees a transposed cache line is not split across L3 banks, and is still accessible by normal requests (with longer latency to transpose back, see §7.4). This rarely fails for large arrays, as they are often padded for cache line alignment.

When multiple arrays are used by the same computation, e.g. the input and output array of 2D convolution, the runtime picks one primary array (the output or the reduced array) and uses its tile size for others. Using the same tile sizes eases the complexity to align tensors at runtime.

Tiling Heuristics The runtime picks one valid tile size using hints in the configuration. Shifts favor a close-to-square tile size, as it keeps most traffic within the same tile. For reduction, a larger tile size on the reduced dimension allows more rounds of in-memory reduction. Broadcast reads favors a smaller innermost tile size if it can spread one row to more L3 banks to avoid the hotspot. When tensors are used for multiple kinds of data movement, we prioritize by the order of reduction, shift, and broadcast, as reduction is usually more expensive due to low compute intensity, while broadcast is inexpensive, as it can reuse the read data. The runtime can pick the best data layout for each program phase. Our heuristic is within 2% of an oracle configuration (see §7.7).

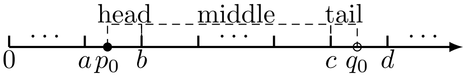
JIT Lowering tDFG

The runtime also lowers the tDFG into in-memory commands. In Fig 7.9, an example `mv` node (right shift columns [0, 3) by 1) is lowered through the following steps.

1. Tensor Decomposition As tensors may not align to the tile boundary (e.g. moving a subregion of the array), they are decomposed into smaller ones to separately handle those tiles at the boundary. Alg 2 recursively decomposes an N-D tensor along the tile boundary at each dimension. For the start and end position p_0, q_0 of dimension 0, it identifies their respective tile boundaries $[a, b)$, $[c, d)$ such that $p_0 \in [a, b), q_0 \in [c, d), \{a, b, c, d\} \mod t_0 \equiv 0$ (line 3-4). Depending on the relative positions of p_0 and q_0, it decomposes the 1D tensor $[p_0, q_0)$ into one to three new ones: additional subtensors for the head and/or tail if p_0 and/or q_0 do not align with the tile boundary. For multiple dimensions, we take the cross product of all decomposed tensors (line 8-18). When the tensor aligns with the tile boundary in every

Algorithm 2: Decompose Tensor

```
   Input: A $N$-dim tensor $A = [p_0, q_0) \times ... \times [p_{N-1}, q_{N-1})$ where $p_i < q_i$
   Input: A list of tile size of each dim $ts = [t_0, ..., t_{N-1}]$
   Result: A list of decomposed tensors $ret$ initialized as []
1  if N ¿ 0 then // Decompose dimension 0
2      //
```



```
3      $a \leftarrow \lfloor \frac{p_0}{t_0} \rfloor \times t_0$, $b \leftarrow \lfloor \frac{p_0+t_0-1}{t_0} \rfloor \times t_0$ // Align $p_0$ to tile boundary
4      $c \leftarrow \lfloor \frac{q_0}{t_0} \rfloor \times t_0$, $d \leftarrow \lfloor \frac{q_0+t_0-1}{t_0} \rfloor \times t_0$ // Align $q_0$ to tile boundary
5      // Recursively decompose remaining dimensions
6      $rs \leftarrow$ Decompose($[p_1, q_1) \times ... \times [p_{N-1}, q_{N-1}), [t_1, ..., t_{N-1}]$)
7      forall $A' \leftarrow rs$ do // Construct final decomposed tensors
8          if b ¡= c then // $a \leq p_0 < b \leq c \leq q_0 < d$
9              if $a < p_0$ then
10                 $ret$ += $[p_0, b) \times A'$ // Head interval
11                 if $b < c$ then
12                     $ret$ += $[b, c) \times A'$ // Possible middle interval
13             else
14                 $ret$ += $[a, c) \times A'$ // $p_0$ aligns with $a$
15             if $c < q_0$ then
16                 $ret$ += $[c, q_0) \times A'$ // Add possible tail interval
17         else // $a = c \leq p_0 < q_0 < b = d$
18             $ret$ += $[p_0, q_0) \times A'$ // Same tile, no decomposition
19 else // No more dimension to decompose
20     $ret$ += $A$
```

dimension, no decomposition is needed.

For example in Fig 7.9, `A[0,4)x[0,3)` is decomposed into two subtensors `A`$_\mathtt{L}$`[0,4)x[0,2)` made of full tile 0 and 2, and `A`$_\mathtt{R}$`[0,4)x[2,3)` made of partial tile 1 and 3. Since dimension 0 is perfectly aligned, the original range $[p_0 = 0, q_0 = 4)$ is kept (line 13). For dimension 1, the range $[p_1 = 0, q_1 = 3)$ means the tail is not aligned ($t_1 = 2 \implies q_1 \mod t_1 \not\equiv 0$). Therefore dimension 1 is decomposed into $[p_1 = 0, 2)$ and $[2, q_1 = 3)$. The cross product between decomposed dimensions 0 and 1 yields two subtensors $[0, 4) \times [0, 2)$ and $[0, 4) \times [2, 3)$.

2. Intra-/Inter-Tile Shifts Alg 3 lowers a decomposed `mv` node into intra-/inter-tile shift commands. Each shift command takes five arguments: 1) a tensor A, 2) a shift dimension k, 3) a shift mask that selects the bitlines to shift, and 4,5) the inter-/intra-tile shift distances that indicate the direction and number of tiles/bitlines to shift (intra-tile shifts always have 0 inter-tile shift distance). Depending on whether the shift distance aligns with the tile boundary ($d_{intra} == 0$), we may generate an inter-array shift command and optionally an extra intra-array shift command (line 5-12). Notice that not all shift commands will necessarily generate traffic, as the intersection of the shift mask and the tensor may be the empty set. Such shift commands are filtered out later (ommitted in Alg 3).

As an example, in Fig 7.9, shifting `A`$_{\texttt{L}}$`[0,4)x[0,2)` to the right by one requires one intra-tile shift to move the column 0 (`CMD 0`, Alg 3 line 6), and one inter-tile shift to move the column 1 across the tile boundary (`CMD 1`, Alg 3 line 8). Each command has the bitline/tile pattern generated by intersecting the tensor with the shift mask. These patterns are applied to bitlines/tiles, specified using the `start[:stride:count]+` format. E.g. `CMD 1` has bitline pattern `1:2:2` and tile pattern `0:2:2`, therefore shifts bitline 1, 3 of tile 0, 2 (red arrow). These patterns are expanded into masks by the hardware when executed (see §7.4). Activated wordlines are also encoded, but are omitted in Fig 7.9 for simplicity. Shift commands also have the bitline/tile distance to determine the destination bitline/tile. Similarly, `A`$_{\texttt{R}}$`[0,4)x[2,3)` is shifted to the right by one intra-tile shift (`CMD 2`, Alg 3 line 6), but requires no inter-tile shift (skipped Alg 3 line 8). The runtime ensures data is not shifted beyond the array boundary by checking the tensor size and the shift distance.

3. Map to L3 Banks Some commands, e.g. those for boundary tiles, may be skipped by some banks. The runtime intersects the commands' tile pattern and the tiles mapped to each L3 bank. If the intersection is empty, the command can be skipped at that L3 bank.

Algorithm 3: Compile `mv` to Shift Commands

```
Input: A $N$-dim tensor $A = [p_0, q_0) \times ... \times [p_{N-1}, q_{N-1})$ where $p_i < q_i$
Input: Tile size $t_k$ of move dimension $k$ and move distance $d$
Result: A list of shift commands $ret$ initialized as []
$d_{inter} \leftarrow \lfloor \frac{\text{abs}(d)}{t_k} \rfloor$ // Inter-tile shift distance
$d_{intra} \leftarrow \text{abs}(d) \bmod t_k$ // Intra-tile shift distance
$\overline{d}_{intra} \leftarrow t_k - d_{intra}$ // Complement of $d_{intra}$
// Shift(tensor, dim, mask, inter_tile_dist, intra_tile_dist)
if $d > 0$ then // Shift forward
    $ret$ += Shift$(A, k, [0, \overline{d}_{intra}), d_{inter}, d_{intra})$
    if $d_{intra} > 0$ then
        $ret$ += Shift$(A, k, [\overline{d}_{intra}, t_K), d_{inter} + 1, -\overline{d}_{intra})$
else if $d < 0$ then // Shift backward
    if $d_{intra} > 0$ then
        $ret$ += Shift$(A, k, [0, d_{intra}), -(d_{inter} + 1), \overline{d}_{intra})$
    $ret$ += Shift$(A, k, [d_{intra}, t_K), -d_{inter}, -d_{intra})$
```

In Fig 7.9, since `CMD 0` operates on tile 0 (mapped to L3 bank 0) and tile 1 (mapped to L3 bank 1), it is mapped to both L3 banks.

Other tDFG Nodes Element-wise compute nodes do not move the data and can skip step 2, but still needs step 1 and 3 to handle the boundary tiles and to be mapped to L3 banks. The compute commands also encode the opcode and the wordlines of the operands and result. Reduction nodes are lowered into a sequence of interleaving compute and intra-tile shift commands to fully reduce each tile on the reduced dimension. Broadcast nodes are handled similarly to move nodes, with the broadcast destination encoded.

Synchronization All commands are synchronous at L3 banks (i.e. do not issue until the previous one finished) except inter-tile shifts, which are considered finished when all data movement within the L3 bank *and* the inter-bank packets are injected into the NoC (but may before they arrive at the destination L3 bank). Therefore, the runtime inserts a sync command between an inter-tile shift command and the consuming command, which serves as a global memory barrier, ensuring that data movements before the sync command are visible to commands after the sync command. (i.e. arrived at the destination bitline). A sequence of pure intra-tile shift and compute commands require no synchronization.

Reducing JIT Overheads Being on the critical path of offloading, JIT lowering can incur significant overheads. Thus, we co-design the software and hardware for JIT performance:

- **Division of labor:** The static compiler handles register allocation and scheduling (see §7.2), so the JIT compiler only needs to map the scheduled tDFG according to the tiled data layout and lower into bit-serial commands. This is possible by scheduling for common SRAM array sizes (256x256 and 512x512), forming a fat binary similar to CUDA. Note that our fat binary does not expose any microarchitecture beyond the SRAM array sizes, and we believe there will only be a small handful that are useful over many generations of hardware.
- **Memoization:** We reuse JIT results when the same tDFG is re-executed with the same parameters by adding a small hardware cache (see §7.4) for intermediate reuses and software memorization for longer-term reuses. This is particularly useful for iterative algorithms (e.g. stencils).
- **Array dimension specialization:** While our JIT compiler can handle higher dimensional arrays, we specialize for common 1-3D arrays by leveraging C++ templates. This enables the compiler to unroll the loop and eliminate expensive recursion (e.g. Alg 2 recursively decomposes the tensor according to the tile boundary).

With these optimizations, we reduced JIT lowering time by more than 1000×, and it takes 12% of overall runtime (see §7.7). We believe additional optimizations could further reduce the overhead, e.g.:

- **Phase overlapping:** We can overlap JIT compiling with the data preparing phase (to fetch and transpose data, see §7.4), or lowering for future regions as the core is waiting for the current region to finish.

- **Hardware implementation:** We can broadcast commands after step 2 to all L3 banks and let the hardware skip those not applied to its local tiles, eliminating step 3 (the most time-consuming one as it is $O(N_{bank} \times N_{cmd})$) in software.

In-/Near-Memory Decision

The runtime also decides between in-/near-memory computing by evaluating the following condition:

$$\frac{N_{elem} \times N_{op}}{TP_{core}} > \Sigma_i Lat_{op_i} + N_{node} \times Lat_{JIT} \tag{7.2}$$

The LHS models the latency of a core at peak throughput, and the RHS captures the in-memory computing delay (first term, no N_{elem}, as computation is fully parallelized) and the JIT time (second term). The compiler generates aggregate information as hints in the configuration, e.g. # of each op, so that the runtime can make a quick decision without analyzing the tDFG. Other platform-specific parameters can be obtained by querying the hardware or profiling offline. This is just a basic and conservative heuristic (assuming peak core performance), but is sufficient for the studied workloads.

7.4 Microarchitecture Extensions

Fig 7.10 overviews infinity stream's microarchitecture, with stream engines ($\mathrm{SE_{CORE}}/\mathrm{SE_{L3}}$) handling offloaded near-memory streams, layout override tables (LOT) recording transposed data layout, and tensor controllers ($\mathrm{TC_{core}}/\mathrm{TC_{L3}}$) executing in-memory commands and synchronizing with the core.

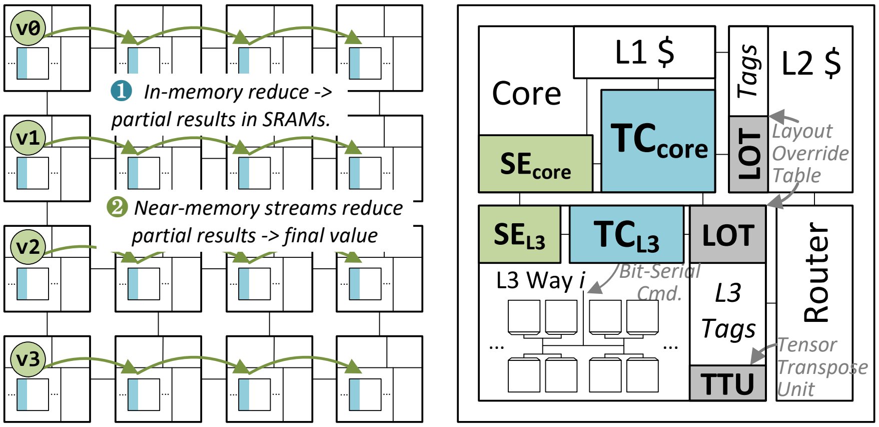


Figure 7.10: Infinity Stream Microarchitecture

Near-Memory Computing

We adopt near-memory computing μarch support from NSC [31] to execute streams at the L3 stream engine (SE_{L3}). Streams read/write data directly from L3 banks and forward operands to consuming streams without going back to the core for computing. Streams automatically migrate to the L3 bank where the next data is mapped, with coarse-grained flow control messages (i.e. sync every N cache lines between SE_{CORE} and SE_{L3}) to reduce coordination.

In-Memory Computing

During in-memory computing mode, the microarchitecture needs to manage the transposed data layout (LOT and TC_{core}), execute the in-memory commands (TC_{L3}), and synchronize with the core (TC_{core} and TC_{L3}). We assume the SRAM arrays are enhanced to support bit-serial logic and shifts, as well as a buffered H tree to enable efficient broadcast, similar to [20, 21].

Transposed Data Layout The layout override table (LOT, Table 7.1) tracks the transposed arrays initialized by the runtime (up to 3D, so higher-dim arrays should have some

Field	Bits	Description	Field	Bits	Description
`base`	48	Base phys. addr.	`end`	48	End phys. addr.
`size`	8	Element size.	`dim`	2	Array dim (max 3).
$\mathtt{S_i}$	32	Array size (3×).	$\mathtt{T_i}$	32	Tile size (3×).
`wl`	10	Start wordline.	`trans`	2	Transpose state.

Table 7.1: Layout Override Table (LOT)

dimensions fused). It tracks the physical address, as the L2 and L3 caches are indexed by physical addresses. This requires the array to be contiguous in physical address space (with huge pages or special malloc functions). Directly mapping virtual addresses to bitlines is possible by extending the page table and TLB for transposed pages, but is beyond this work.

Map Physical Address ⇔ Bitlines The LOT essentially overrides how physical addresses are mapped to SRAM arrays. For transposed data structures, the physical address is subtracted by `base` and divided by `size` to get the element index, which is used to find the containing tile and coordinates within that tile. Since tiles are mapped contiguously to SRAM arrays, it is straightforward to locate the actual bitline and wordlines. Reverse mapping from bitlines to physical addresses is similar.

Prepare Transposed Data Before in-memory computing, TC$_{\text{core}}$ prepares the data in transposed format by first issuing flush requests to the L3 cache controller to reserve the cache ways used for in-memory computing (we use 16 ways).

The `trans` field in LOT (initialized to 0) indicates whether the data is currently cached in transposed layout. If `trans=0`, TC$_{\text{core}}$ offloads a load stream to fetch the data into transposed format, and sets `trans=2` when finished. During this process, TC$_{\text{core}}$ sets `trans=1`, and any core requests to that physical range is blocked. These load streams are executed in SE$_{\text{L3}}$ to avoid the traffic overheads between L2 and L3. Our design uses a tensor transpose unit (TTU) to convert between transposed and normal format, similar to prior works [20, 21].

Execute Commands After the data is prepared, TC_{core} sends out commands in a small command cache (2kB) to TC_{L3} at mapped L3 banks. Commands are generated by the runtime (see §7.3) or reused if the same region is executed multiple times. TC_{L3} is a micro-controller to convert the command's bitline and tile pattern to masks for its local tiles and broadcast commands to SRAM arrays. For inter-tile shifts, it generates the control signals to configure the H tree to shift or broadcast the data, and packs the bits into NoC packets if the destination tile is mapped to another L3 bank. For compute commands, it first broadcasts constant operands (if any) to bitlines, and configures the SRAM arrays to perform the bit-serial computation (using algorithms from prior work [21]). Since commands are long latency ($n^2 + 5n$ for n-bit integer multiply), TC_{L3} can preprocess the next command to hide the processing latency.

Synchronization For sync commands, TC_{L3} reports to the other TC_{L3} the # of packets sent there since the last sync, and the total sent packets to TC_{core}. Therefore, the receiving TC_{L3} knows how many packets to expect and can report back to TC_{core} when all packets arrived. After hearing back from all TC_{L3}s, TC_{core} checks that # of sent/received packets matches before broadcasting a message to clear the barrier.

Delayed Release of Transposed Data To release the transposed data, TC_{core} offloads a special store stream to evict data to the memory, which releases the reserved cache ways. To capture the reuse across program regions, e.g. iterative algorithms, TC_{core} delays releasing the data until any of the following conditions:

- The number of normal requests to the transposed data exceeds a threshold (we use 100k), suggesting that it is now used for in-core/near-mem computing.
- The L3 miss rate exceeds a threshold, suggesting releasing the reserved ways to reduce the pressure on the L3.
- A timer expires (we use 100k cycles).

Fused In-/Near-Memory Computing

One key advantage of infinity stream is to enable normal core/stream accesses to the transposed data, which allows cores/streams to be *unaware* of the data layout, providing flexibility across paradigms.

Coherence Tiling constraints in §7.3 guarantees that transposed cache lines are still mapped to a single (but maybe different) L3 bank. Therefore, the coherence state can be tracked in the newly mapped L3 bank, enabling accesses to transposed data structures using normal requests when in-memory computing is not used. Before in-memory computing starts, $\mathrm{TC}_{\mathrm{core}}$ evicts any dirty copies in private caches to ensure the data in L3 is up-to-date. During in-memory computing, cores are disabled from accessing the data structure by blocking the requests from private caches (setting `trans` in LOT to 1). However, streams at $\mathrm{SE}_{\mathrm{L3}}$ can still read and write transposed data, as the dependence between stream and tensor operations is guaranteed through the dataflow graph and synchronization. E.g. the final reduce stream is not offloaded until the partial in-memory reduce is synchronized at $\mathrm{TC}_{\mathrm{core}}$. Similarly, if a tensor is generated by a store stream, the dependent in-memory computation will not start until that stream completes.

Context Switch As in [31], context switches in near-memory computing are delayed until all streams reach a synchronization point (every few cache lines). Similarly, during in-memory computing, context switches are delayed until $\mathrm{TC}_{\mathrm{core}}$ completes a sync command so that all computation and data movement is committed. The progress of streams (including iteration number) and in-memory computing progress (commands), as well as the LOT, are saved as part of architectural state. The OS may flush transposed data so that LLC space can be reclaimed.

7.5 Implementation Limitations

Our implementation of infinity stream has some limitations that can be relaxed in future works: 1. While it is possible to share the L3 to enable in-memory computing in a multi-program scenario, we allow only one thread to reserve the L3 for in-memory computing at a time by locking the LOT. 2. We assume the input data is already tiled to fit in the L3. Otherwise, in-memory computing is disabled. Future work could support automatically tiling at runtime. 3. We currently do not support register spilling because all studied kernels can fit in the available registers. Register spilling can be implemented by a stream writing back and loading from the DRAM.

7.6 Methodology

Compiler and Runtime We extend the open-sourced LLVM-based near-stream computing compiler [31] to unroll sDFGs into tDFGs as described in §7.2. For tDFG optimization, we define the tDFG rewrite rules in the *egg* library [229] to explore the e-graph (see Appendix for details). Optimized tDFGs are serialized back to the x86 backend in LLVM (extended with infinity stream instructions). The compiler inserts calls to a C++ runtime library to JIT compile tDFGs and manage the data layout.

Simulator We use gem5-20 [107] for execution-driven, cycle-level simulation, extended with partial AVX-512 support. The L3 cache is extended to model the transposed data layout and in-memory bit-serial computation.

Parameters and Configurations Table 7.2 lists system parameters. In total, it has 4M bitlines and provides massive parallelism for in-memory computing. The **Base** OOO cores use advanced L1 and L2 prefetchers [110]. For near-memory computing, **Near-L3** offloads streams and the associated computation to $\mathrm{SE}_{\mathrm{L3}}$. For infinity stream, we evaluate three configurations:

System	2.0GHz, 8x8 Cores
OOO8 CPU	64 IQ, 72 LQ, 56 SQ+SB
(8-issue)	348 Int/FP RF, 224 ROB
Func. Units	8 Int ALU/SIMD (1-cycle)
	4 Int Mult/Div (3/12-cycle)
	4 FP ALU/SIMD (4-cycle)
	4 FP Div (12-cycle)
L1 D/I TLB	64-entry, 8-way
L2/SE$_{\text{L3}}$ TLB	2k/1k-entry, 16-way, 8-cycle
L1 I/D Cache	32KB, 8-way, 2-cycle
Priv. L2 Cache	256KB, 16-way, 16-cycle
Replacement	Bimodal RRIP, $p = 0.03$
L1 Bingo Prefetcher	8kB PHT, 2kB region
L2 Stride Prefetcher	16 streams, 16 pf./stream

NoC	32B 1-cycle link, 8x8 Mesh
	5-stage router, multicast
	X-Y routing, 16 mem. ctrls
Shared	20 cycles, MESI
L3 $	Static NUCA, 1kB interleave
	256x256 SRAM array (8kB)
	5-level H tree, 64B total BW.
	16 arrays per way, 18 ways
	64 banks, total 144MB
DRAM	3200MHz DDR4 25.6 GB/s
SE$_{\text{CORE}}$	2kB FIFO, 12 streams
SE$_{\text{L3}}$	768 streams, 64kB buf.
	4-cycle compute init. lat.
LOT	16 regions

Table 7.2: System and μarch Parameters

- **In-L3** invokes a runtime JIT library to manage the data layout and lower tDFG into bit-serial commands to compute with L3 SRAMs, but no near-memory computing support.
- **Inf-S** adds near-memory computing to **In-L3** by offloading sDFG to the SE$_{\text{L3}}$.
- **Inf-S$_{\textbf{no JIT}}$** assumes that input and hardware parameters are known, so tDFG is pre-compiled (no runtime lowering).

Benchmarks We evaluate 13 dense fp32 OpenMP workloads, compiled with `-O3` and vectorized by AVX-512 for **Base** and **Near-L3**. For infinity stream, a single-thread scalar version is sufficient, as streams are spatially unrolled to all bitlines. Table 7.2 summarizes the input data sizes and the major data movement (tensor shift vs. tensor broadcast) and

Benchmark	Move	Cmp.	Parameters
stencil1d	Shift	Elem	4M-entry, 10-iter
stencil2d	Shift	Elem	2k×2k, 10-iter
stencil3d	Shift	Elem	512×512×16\ 10-iter
dwt2d	Shift	Elem	2k×2k
gauss_elim	BC	Elem	2k×2k
conv2d	Shift	Elem	2k×2k
conv3d	BC	Elem	H/W=256,K=\ 3×3, I/O=64
mm/in	BC	Rdc	M/N/K=2k
mm/out	BC	Elem	Same
kmeans/in	BC	Rdc	32k-point,dim=128,128-center
kmeans/out	BC	Elem	Same
gather_mlp/in	BC	Rdc	M=32k,\
gather_mlp/out	BC	Elem	N/K=128

Table 7.3: Workloads (BC: Broadcast)

Krnl.	K, N, r, [$dims$]
SA1	512, 32, 0.2, [64, 64, 128]
SA2	128, 64, 0.4, [128, 128, 256]
SA3	1, 128, Inf, [256, 512, 1024]
SA4	512, 16, 0.1, [32, 32, 64]
SA5	512, 32, 0.2, [64, 64, 128]
SA6	512, 128, 0.4, [64, 96, 128]
SA7	128, 16, 0.2, [64, 64, 128]
SA8	128, 32, 0.4, [128, 128, 256]
SA9	128, 128, 0.8, [128, 128, 256]
FCx3	1, 1, /, [512, 256, 10]
SSG	SA1 → SA2 → SA3 → FCx3
MSG	[SA4, SA5, SA6] → [SA7, SA8, SA9] → SA3 → FCx3

Table 7.4: PointNet++

computation patterns (element-wise vs. reduction) for each benchmark.

Some benchmarks have different implementations, e.g. inner product vs. outer product for `mm`. We pick the best implementation for each configuration when comparing the performance and energy efficiency, and provide a detailed sensitivity study of the preferences of different paradigms in §7.7.

We also perform an end-to-end study on PointNet++ [230], a popular hierarchical neural network for point cloud classification and segmentation, in §7.7.

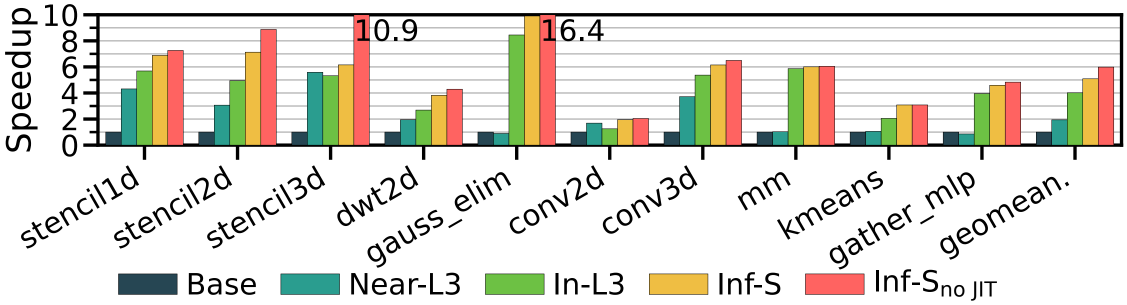


Figure 7.11: Overall Speedup

7.7 Evaluation

Overall Performance Fig 7.11 shows the overall speedup over **Base**, and Fig 7.12 shows the NoC utilization and traffic breakdown. The NoC traffic is categorized as the traffic of the coherence control messages (control), the traffic of moving data around (data), and the traffic of all the control messages to manage the offloaded computation, e.g. flow control for streams and synchronization for in-memory computing. For benchmarks with multiple dataflow designs (`mm`, `kmeans`, `gather_mlp`), we pick the best implementation for each configuration (see below for a detailed comparison between dataflow choices). Overall, **Near-L3** achieves 2.0× speedup and 29% traffic reduction by offloading streams near L3 banks, but may hurt the performance as it is unable to capture the reuse; e.g. for `kmeans` **Near-L3** introduces 2.6× extra NoC traffic.

By leveraging massive parallelism in bitlines, **In-L3** achieves 2.1× speedup over **Near-L3**. However, without near-memory computing support, **In-L3** failed to realize the full potential of near-data computing, e.g. in `kmeans`, both aggregation and centroid recomputation are executed by the core and not offloaded. On the other hand, by enabling hybrid in-/near-memory computing, **Inf-S** yields another 24% speedup over **In-L3** (2.6× over **Near-L3**), and 90% NoC Traffic reduction over **Base**. To understand the benefit of traffic reduction, Fig 7.13 shows the detailed traffic breakdown for **Inf-S**, adding the intra-/inter-tile shift

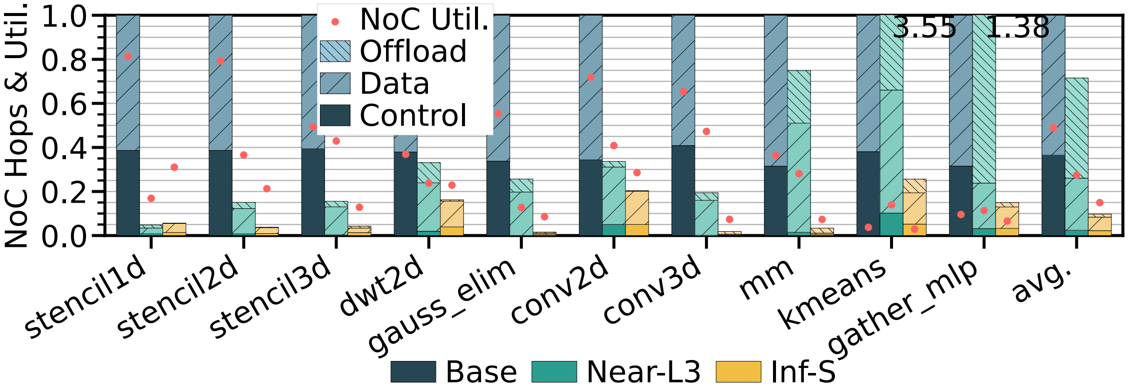


Figure 7.12: NoC Traffic Breakdown (Bar) and Util. (Dot)

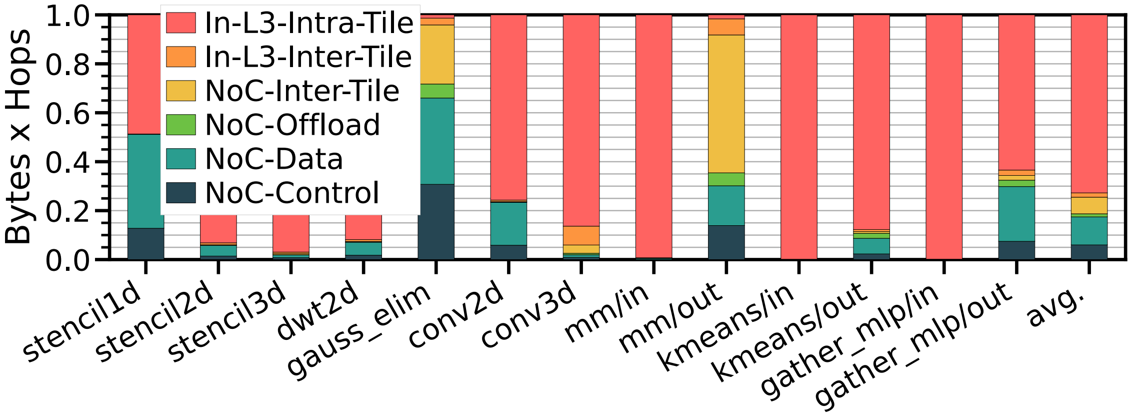


Figure 7.13: **Inf-S** Traffic Breakdown

traffic. Notice that some inter-tile shift traffic goes through the NoC if the destination tile is not mapped to the same L3 bank, and is shown separately from NoC-Data as NoC-Inter-Tile. By choosing a reasonable tile size, **Inf-S** converts most of the data movement into intra-tile shifts, leveraging the massive parallelism to shift bitlines within each SRAM array.

Cycle Breakdown Fig 7.14 breaks down the cycles of **Inf-S** into transferring and transposing data from/to DRAM (DRAM), lowering tDFG to commands (JIT Lower), moving tensors (Move), bit-serial in-memory computing (Compute), final reduction of the in-memory partial results (Final Reduce), hybrid in-/near-memory computing (Mix), as well as pure

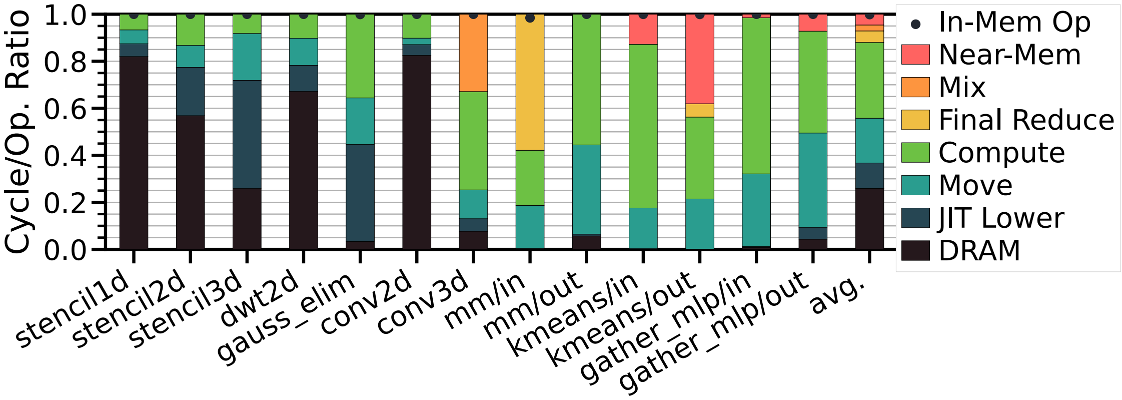


Figure 7.14: **Inf-S** Cycle Breakdown

near-memory computing (Near-Mem). Overall, in-memory computing takes 88% of total cycles, with 26%, 32%, and 19% spent on DRAM transfer, computing, and tensor moving respectively. 4.9% of cycles are spent waiting for the final reduction from near-memory streams, e.g. `mm_inner`. Transposing is cheap when there is high reuse, e.g. `gauss_elim` and `mm`. Dots in Fig 7.14 indicates the percentage of ops offloaded to bitlines – nearly all computation (99%) are performed in-memory.

JIT Overheads As shown in Fig 7.14, JIT lowering contributes 11% of the total runtime, and can be more than 50% when we cannot reuse the lowered commands (51% for `gauss_elim`), or when a high-dimensional tensor is not aligned to the tile size and requires more commands to handle boundary tiles (50% for `stencil3d`). If all input sizes and hardware parameters are known at compile time, the compiler could precompile the tDFG into commands without invoking the JIT runtime. **Inf-S$_{\textbf{no JIT}}$** in Fig 7.11 represents such a configuration and yields another 19% speedup over **Inf-S**. The average JIT time is 220us (σ 449us), with `gauss_elim` as the outlier (1616us) as the tensor is shrinking every time. We believe by overlapping JIT lowering with DRAM fetching and command execution, as well as applying more advanced software optimizations, the overheads would be further reduced.

Dataflow Choices Fig 7.15 shows the speedup of inner and outer product versions of

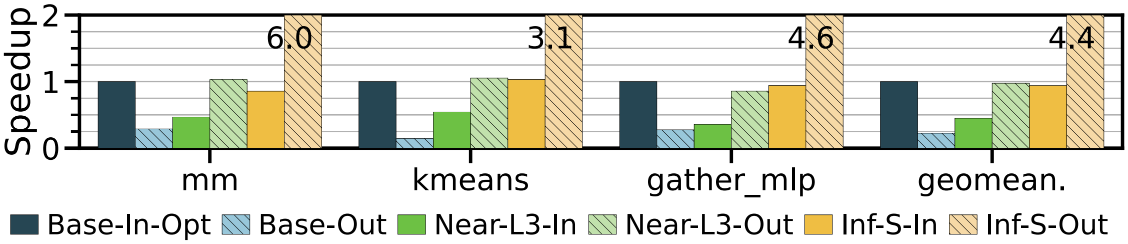


Figure 7.15: Inner vs. Outer Product Dataflow

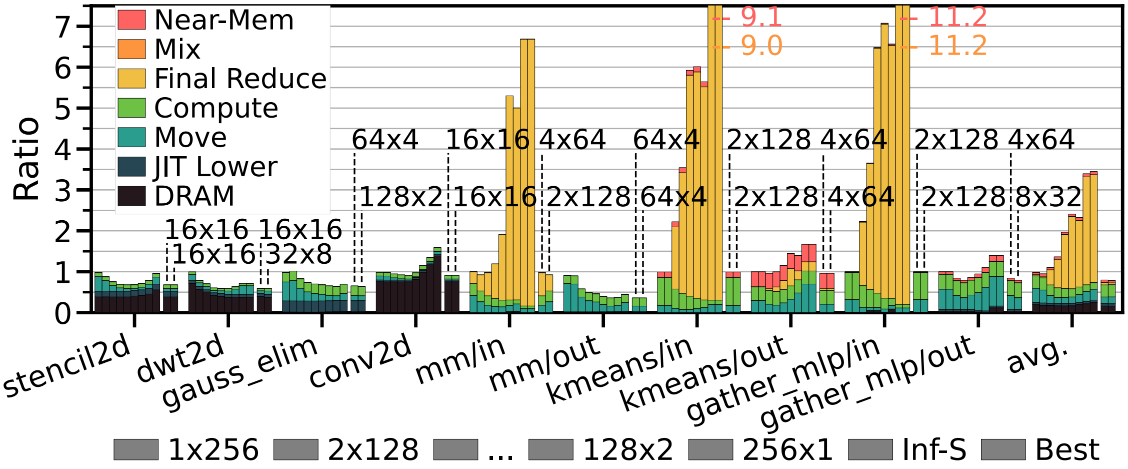


Figure 7.16: Cycle Breakdown vs. 2D Tile Size

`mm`, `kmeans`, and `gather_mlp` on different paradigms, normalized to a tiled inner product version for **Base**. As expected, **Base** favors the inner product implementation, as it could accumulate the result in the register file. **Near-L3** generally suffers as it cannot explore the data reuse when offloaded to L3, and favors the outer product version, as the dataflow allows the stream engine to partially recognize the broadcast pattern and save some data traffic (similar to [32]). For **Inf-S**, the outer product is a clear win, as it exposes the maximal data parallelism in the inner loops, and avoids the inefficient in-memory reduction. Overall, it achieves 4.4× speedup over **Base**. Therefore we implement tiled inner product for **Base** and outer product for **Near-L3** and **Inf-S**.

Data Layout Fig 7.16 shows the cycle breakdown of all 2D benchmarks with various

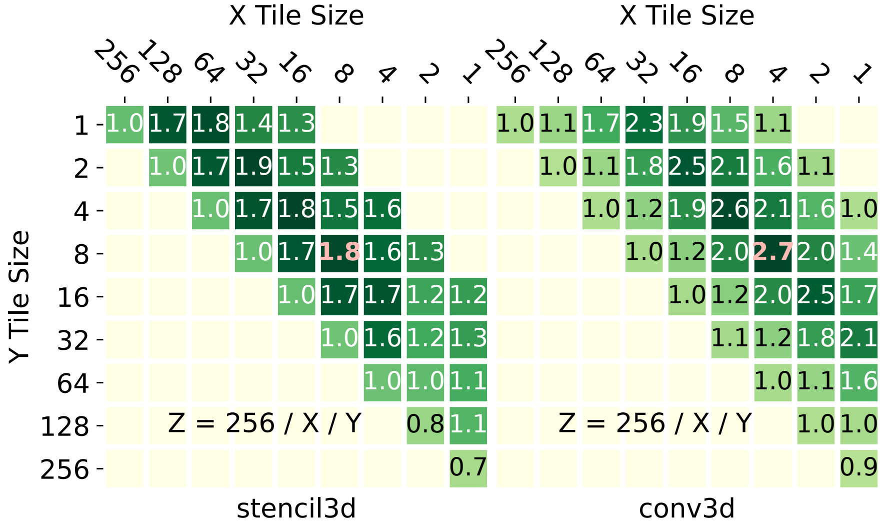


Figure 7.17: Inf-S Speedup vs. 3D Tile Size (Default as Bold)

tiling sizes, annotated with the best and default tile size chosen by the runtime. Similarly, Fig 7.17 shows the speedup vs. 3D tiling sizes. For benchmarks with shift data movement, e.g. `stencils` and `dwt2d`, picking a balanced tile size (16 × 16 for 2D arrays) usually yields close to optimal performance. When tensors are broadcast, e.g. `gauss_elim` and `mm`, having a smaller innermost tile size helps avoid the hotspot of reading the source row from a single L3 bank. When reduction is needed, a larger tile size at the reduced dimension increases the computation density for in-memory computing and improves the performance. For example, for `kmeans/in` and `gather_mlp/in`, since the size of the reduced dimension is 128, tiling by 128 allows pure in-memory reduction to produce the final results in each SRAM array (hence no Final Reduce bar). Overall, our heuristic achieves within 2% of an oracle, and yields 34% speedup over no tiling (laying the innermost dimension continuously) across all 2D/3D benchmarks.

Energy and Area The energy breakdown for the SRAM arrays and H tree were obtained from CACTI [185] (22nm) where compute only involves the SRAM arrays while tDFG `mv` node uses both. Fig 7.18 shows the energy efficiency over **Base**. **Inf-S** yields better energy

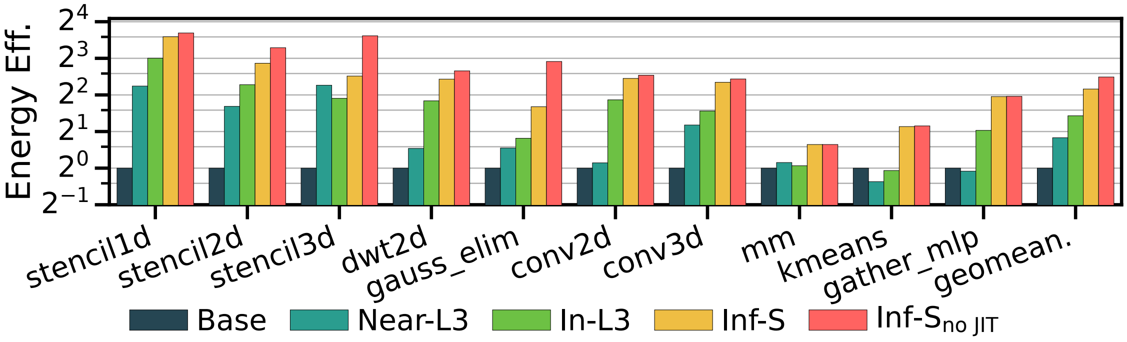


Figure 7.18: Overall Energy Efficiency

efficiency for workloads with less reuse by converting NoC traffic into intra-tile shifts. Overall, **In-L3** and **Inf-S** achieve 1.5× and 2.4× energy efficiency over **Near-L3** respectively.

Most of the area overhead comes enhancing existing SRAM caches for compute: additional sense amps and write drivers so every bitline can compute, an extra decoder to read two wordlines simultaneously, and the compute logic. Our area model consists of the overall CPU area reported by McPAT [50] (22nm), the in-memory compute overhead from Neural Cache's [20] die analysis [2], and near-memory support logic [31]. After adding additional logic for in-memory compute (66.75mm^2) and near-memory support (28.16mm^2), the whole chip area overhead is 6.52%.

Case Study of PointNet++ To better understand the benefit of infinity stream on real applications, we perform an end-to-end study on PointNet++ [230], a widely applied hierarchical neural network for point cloud applications. The basic component of PointNet++ is set abstraction (SA), which consists of the following stages:

- **Furthest Sample:** Iteratively picks K centroids (points) from the input point cloud. For iteration $k + 1$, the new centroid is the furthest point from the k prior centroids, with the first one randomly selected.

[2] We determine the subcircuit area with COFFE [231].

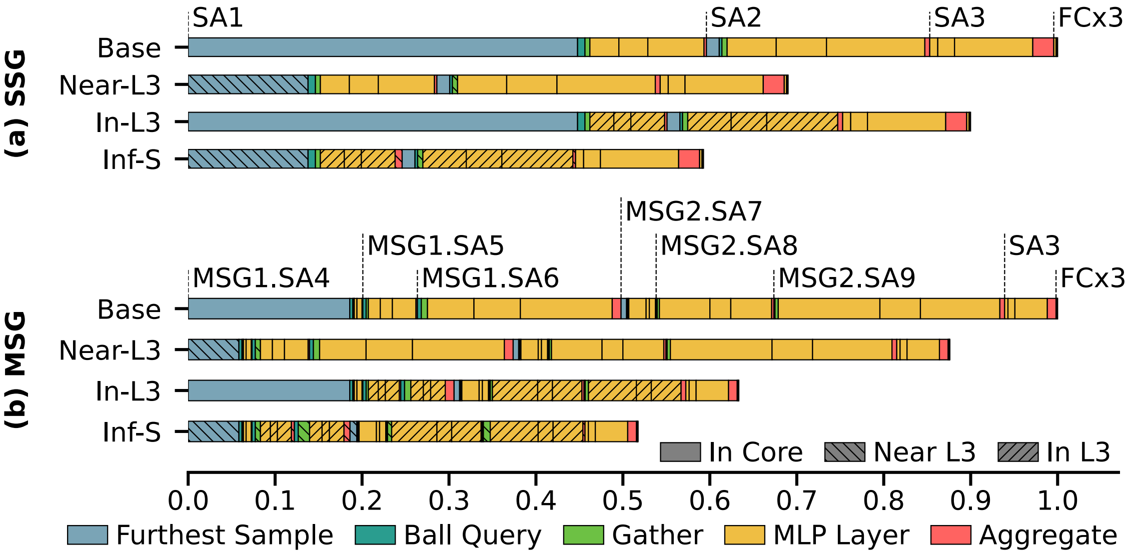


Figure 7.19: Timeline of PointNet++ SSG/MSG Classifier

- **Ball Query:** Searches for N neighbor points within radius r of each centroid. If less than N neighbors are found, the first neighbor is duplicated to fill the remaining spots.

- **Gather:** Performs an indirect gather to collect neighbors' feature vectors. Generates a matrix of $(K \times N) \times D_{in}$ where D_{in} is the dimension of the input feature vector.

- **MLP:** Feeds the gathered matrix into a 3-layer MLP. All layers use ReLU as the activation function. The output matrix is $(K \times N) \times D_{out}$ where D_{out} is the dimension of the last MLP layer.

- **Aggregate:** Reduces the neighbors' feature vectors by taking the max value of each dimension. Outputs a matrix of $K \times D_{out}$.

To perform point cloud classification or segmentation, the authors proposed two network architectures:

- **Single Scale Grouping (SSG):** Multiple SAs are chained with previous output centroids being sampled and grouped by the next SA. This is usually followed by a few fully-connected (FC) layers to produce the final scores for classification.

- **Multiple Scale Grouping (MSG):** To better adapt to various sampling densities, multiple SAs with different radii are applied simultaneously to the input, with their output feature vectors concatenated as the final output. Similar to SSG, this can be chained and followed by more SA/FC layers.

We evaluate both SSG and MSG for classification inference. Table 7.4 lists the detailed parameters of all SAs and the network structure of SSG/MSG, taken from [230]. The input point cloud contains 4k randomly generated points, normalized to $[0, 1)$.

Fig 7.19(a) shows the normalized timeline of PointNet++ SSG, broken into different stages with the texture indicating where the computation is executed (in-core, near-L3 cache, or in-L3 SRAM). For SSG, the MLP layers are relatively small with high reuse in the private cache, and with AVX-512 and OpenMP, it only takes 48% of the total runtime in **Base**. This also limits the potential for in-memory computing, e.g. for the first MLP layer in SA1, the amount of data parallelism can only fill 1/4 of the available bitlines, falling short to amortize the long compute latency of bit-serial operation. Therefore, **In-L3** only yields a 10% speedup over **Base**.

On the other hand, furthest sampling takes 46% of the total runtime. This is because it is an iterative algorithm without sufficient work in each iteration to amortize the synchronization overhead of OpenMP. Also, the working set cannot fit in the private L1 cache, yielding a high miss rate and hurting the performance. These characteristics make it a good candidate for near-memory computing, which achieves 3.1× speedup for sampling, and 31% performance improvement for **Near-L3** over **Base**.

Fig 7.19(b) shows a similar normalized timeline for PointNet++ MSG. In MSG, sampling is less of a bottleneck as the sampled centroids are shared between SAs within the same MSG. Also, MSG uses larger MLP layers, increasing the data parallelism. This makes in-memory computing more favorable, and **In-L3** achieve 37% speedup over **Base** (12% for **Near-L3**).

Finally, by leveraging the fused compiler/ISA/runtime abstraction, **Inf-S** can flexibly execute the kernel in the core, near the L3 cache, or in the L3 SRAM. The runtime can avoid offloading small MLP layers to in-memory computing as it hurts the performance, e.g. SA3 and FC layers. Overall, it achieves the highest performance (1.69× and 1.93× over **Base** for SSG and MSG respectively).

*Key Takeaway: **Inf-S** fuses the benefits of in-/near-memory computing, unlocking the full potential of near-data processing.*

7.8 Related Work

In-memory Computing for CPU Caches Prior works also augment CPUs for computing in on-chip SRAM caches. Compute cache [149] enables in-memory computation for CPU cache SRAMs, but only supports the less general bit-parallel layout, single-dimension bit-level vector ops (as opposed to multi-dim tensor level). GenPIM adds NVM-based in-memory computing to a general purpose core [232]. Inhale and Sealer enable in-memory encryption at L1 [233, 234]. Neither of the above implements a high-level compiler. Duality cache proposes a bit-serial in-memory approach for CPUs codesigned for CUDA programming [21]. None of these enables portable/transparent support for in-cache computing.

Improving Near-Data Programmability Various *near-data* approaches have developed techniques to improve programmability. PEI enables programming through instruction intrinsics [14]. SnackNoC [4], Active Routing [16] and Dist-DA [2] specify computation offloads with dataflow graphs. Tesseract uses remote function calls [10]. Livia uses single-cache-line accessing functions [27]. Our work relies on stream abstractions, i.e. long-term memory access patterns, which have been applied both in general purpose processors [63, 28, 32, 31, 29] and accelerators architectures [112, 174, 113, 114, 235, 236].

Other near-data programming models are nearly transparent to the programmer. Several are limited to thread-level near-data decisions, programmed with CUDA or OpenMP [11, 12, 143, 142]. Other works enable transparent near-data at a finer grain, but have other limitations, like OmniCompute [3] (only for short RMW instruction chains), EMC [115] (only for address gen.), and Near-stream computing [31]. These cannot be naively applied to enable programmability for PIM, because they do not manage data transposition or guarantee bitline-level alignment.

In-Memory Foundations Prior works have explored bit-parallel in-memory computing, primarily for bulk bitwise ops [149, 23, 237, 238]. We adopt the bit-serial approach for this work, which enables broader support for more operations, including floating point.

DRAM devices have been the target of both in-memory [23, 168, 239, 240, 241, 242, 243] and near-memory processing [244, 245, 246]. In-DRAM computing provides more parallelism, while in-SRAM computing limits modifications to the CPU. We choose SRAM as the first step due to the trend towards large LLCs and the fact that many algorithms are already tiled for the LLC. However, infinity stream can be applied to both cases, as the abstraction (tDFG) is neutral to the hardware, and the JIT runtime can be extended for in-DRAM computing (e.g. triple-row activation). The memory controller also needs to be extended to support streams. Similar to DMA, coherence could be maintained by evicting cache lines from SRAM.

This work relies heavily on prior efforts to develop the paradigm and circuits of in-SRAM computing devices, including for bit-serial integer [247] and floating point ops [21, 248, 249]. Our contribution is about architecture support for these existing technologies.

Recent works have also proposed offloading to multiple hierarchy levels, leveraging properties like data density (SISA [24]), cache presence (Livia [27]), or offline analysis (MLILP [25]). None of them enable portable in-memory computing from a general-purpose language.

Domain-Specialization A variety of prior in-memory accelerators are domain-specialized. Many focus on ML [20, 168, 250, 251, 252, 253], while others target graph processing, mining, and physics simulation [10, 24, 254, 255]. Many broader workloads are prime candidates for in-memory computation with infinity stream. For example, several key data center workloads have been adapted to bitvector parallelism. BitWeaving's [256] database column scan produces a comparison bitmask by organizing data to facilitate bit-serial digital comparison. BitFunnel [257] filters documents with a bloom filter, independently computed by determining the hash indices in memory and constructing the bitvector near memory.

7.9 Summary

Infinity stream is a new approach that makes in-memory computing programmer-friendly: We proposed an execution model that fuses in-/near-memory, using an IR called the tensor dataflow graph (tDFG) to capture parallelism, reuse, and layout optimizations; we built an optimizing compiler and JIT-approach to enable long-term portability without sacrificing performance, with a microarchitecture that transparently orchestrates data management and performs data-layout transforms at runtime. Our optimizations provide integer-factor improvement for data processing for only a modest area overhead. More broadly, we believe that rethinking how to compute throughout the memory hierarchy will be critical for enabling extreme system scaling.

CHAPTER 8

Conclusion

This dissertation explores a promising direction to continue the performance and energy efficiency scaling of general-purpose processors: to adopt general, flexible and unified near-data computing with high-level stream abstractions. We have demonstrated that with memory accesses and computations abstracted as streams and near-stream computations, we can orchestrate data and computation seamlessly and dramatically reduce the communication overheads. Such near-stream computing paradigm can provide more than 2× performance and 76% on-chip network traffic reduction.

In addition, this work also uncovers serval key findings. First, a broad range of programs' memory accessing and computing behaviors can be represented by our highly expressive streams and near-stream computations with only well-known compiler analysis and minimal programmer hints, making them suitable candidates to enable general near-data computing. Second, co-optimizing the data layout and data structure is the key to fully realizing the potential benefits of near-data computing. Third, with more and more non-conventional near-data computing substrates, streams provide a unified abstraction to enable hybrid execution and harness the power of a more heterogeneous near-data computing system.

Table 8.1: Characterization of This Work

NDC Work	Yr.	ABST.	Near Where	Substrate	Domain	Program	Data Layout
Goal		**Unified**	**All**	**All**	**General**	**Trans.**	**Automatic**
SSP [28]	'19	Stream	Core	FU	Prefetching	Trans.	Oblivious
Stream Floating [32]	'21	Stream	LLC	FU	Prefetching	Trans.	Oblivious

NSC [31]	’22	Stream	LLC	Local Core	General	Trans.	Oblivious
Infinity Stream [30]	’23	Stream	Multi.[1]	Multi.[2]	General	Pragma	Scratchpad
Affinity Alloc [33]	’23	Stream	LLC	FU	General	API	Automatic

Finally, this dissertation presents five different near-data computing proposals, each with unique tradeoffs and summarized in Table 8.1. Starting from address generation and perfect prefetching in the core (SSP [28]) and the LLC (stream floating [32]), we extend streams with near-stream computations and offload them to LLC (near-stream computing [31]) and enable hybrid near-LLC and in-LLC computing (infinity stream [30]). The latest work provides an automatic framework to optimize the data layout (affinity alloc [33]). Most importantly, they all leverage a *unified* and *general* abstraction: stream, with near-stream computation. This fundamentally eliminates the overhead of translating across different abstractions while retaining the potential to customize according to different scenarios.

We believe this dissertation is just the beginning of enabling general, flexible and unified near-data computing, and we conclude by discussing the implications of this work and future research directions.

Implications

Beyond the promising performance and energy efficiency potential, the broad impact of this work lies in three dimensions: 1. Industry product applicability; 2. Paradigm shift from purely in-core computing to general near-data computing for processor design. 3. Informed and flexible data and computation orchestration through the entire system.

Industry Impact In this work, we take a step-by-step approach to leverage high-level stream semantics to improve the system: started with a stream-specialized processor that can

[1] In-site LLC and Near LLC controller.

[2] In-situ SRAM bitline and near LLC FU (stream engine).

perfectly prefetch stream data, followed by a stream-aware cache system that can remotely generate the address and proactively stream back the data, and finally add the in-cache near-stream computing support. This path is already proven to be a feasible route for industry adoption. For example, the latest H100 GPU from Nvidia added a Tensor Memory Accelerator (TMA) [258], which can be configured with up to a 5D stream pattern and prefetches the tensor into the private cache. Another example is the new Data Streaming Accelerator (DSA) introduced by Intel in their fourth generation Xeon processors [259], which can free the core from data movement and transformation tasks.

As major chip manufacturers continue to scale up their products, it is foreseeable that future designs will embrace more and more near-data computing. For example, both Intel and AMD have shifted from a monolithic chip to a chiplet design to continue the scaling of the number of cores. However, communication bandwidth between chiplets is much more scarce, and near-data computing becomes even more appealing as it fundamentally reduces data movements.

General-Purpose Core Design Our design fundamentally breaks the conventional in-core computing paradigm and shifts to general and flexible near-data computing. Unlike previous near-data computing techniques, the key to our approach is the novel stream abstraction that explicitly expresses high-level long-term program behaviors. This could inspire future general-purpose core design to rethink the ISA abstraction and microarchitecture beyond the conventional von Neumann architecture.

We also note that our evaluation unveils some breaking new tradeoffs for general-purpose core design. With near-data computing, a smaller in-order core could outperform a wider out-of-order core on certain workloads. Also, with the stream pattern exposed, the core pipeline can be tailored to exploit more memory parallelism with fewer hardware resources, e.g. LSQ entries, MSHRs, etc. Stream-aware cache policies could also effectively avoid thrashing and improve cache utilization. Overall, the new design space is huge and full of unexplored opportunities.

Data and Computation Orchestration This work also demonstrates that achieving an informed and efficient data and computation orchestration requires deep coordination throughout the entire system stack. For example, to systematically optimize the data layout for near-data computing requires an extended allocator interface to capture the data affinity requirement, a runtime library to perform data placement and manage allocated data, as well as OS and microarchitecture extensions to support customizable data mapping from the virtual address space to the physical cache banks. More importantly, many optimizations require software codesigns to fully exploit the underlying system. This work provides some initial attempts to break the information gap across various system levels and enable system-level data and computation orchestration.

Open Questions and Future Directions

While this work attains substantial performance improvements, it also opens up many possible future directions to enable heterogeneous, efficient and secure near-data computing.

Embracing Heterogeneity and Disaggregation This work on hybrid near-cache and in-cache computing already demonstrates the power of a heterogeneous near-data computing system, in which computation is flexibly offloaded to suitable computing substrates with a unified abstraction. There is huge potential in this direction as the system continues to scale up with more and more computing and memory resources connected. For example, AMD adopts chiplet architectures with vertically stacked L3 caches, providing massive on-chip storage (768MB for EPYC 7773X) and making near-cache computing even more attractive. When the data can not fit in the on-chip storage, it is natural to further decouple compute logic into the memory hierarchy, i.e. DRAM, persistent memory, or even storage. These memory resources also become more and more disaggregated to provide more flexible provisioning of memory capacity and bandwidth shared between multiple hosts, but come at the price of higher latency and limited bandwidth for remote accesses, which could also benefit from near-data computing. The compute substrates can also be heterogeneous, from general-purpose CPU and GPU architectures to more domain-specific accelerators and low-level in-memory computing using bitline operations.

It is certainly challenging to adopt near-data computing on such a heterogeneous and massive system: how to optimize the data layout to balance parallelism and locality, how to dynamically schedule computation to the efficient substrate, and most importantly, how to design a unified program abstraction that fuses all these paradigms and eases the pain of massive adoption. It certainly requires full-stack codesign from user applications, runtime libraries to architectural interfaces and microarchitectural details to tackle these challenges and realize these opportunities. Based on our previous success on on-chip near-data computing, I believe our approach is promising to efficiently tackle these challenges and enable a more heterogeneous near-data computing system.

Fully Utilized General Purpose Architectures While the compute throughput in general architectures is constantly increasing, they are severely underutilized due to inefficiencies in the memory hierarchy and core pipelines. For example, high-end GPUs from Nvidia in 2023 (H100) provide 12.1× more tensor operation throughput, but only 2.5× memory capacity and 2.2× memory bandwidth compared to prior models in 2018 (V100). To improve the performance and energy efficiency, architects introduce many domain specific accelerators, e.g. TPU from Google, with explicit dataflow to improve utilization of the compute units.

Interestingly, there are some high-level similarities between today's general-purpose processors and accelerators: they both employ a tiled design with cores or processing elements (PEs) connected with an on-chip network. But when we zoom in, they are very different in terms of orchestration of data and computation – general-purpose processors usually manages the data implicitly with caches, while accelerators usually explicitly dictate how data is moved and reused by mapping a dataflow graph.

This work already demonstrates that explicitly encoding the access pattern in the memory abstraction unlocks new opportunities for data orchestration. This could eventually creating a spatial accelerator overlay on general architectures, significantly improving the utilization of existing computing resources and avoiding data movement back and forth between general-purpose processors and accelerators.

Security and Near-Data Computing While my previous research focuses more on improving the generality, performance and energy efficiency of near-data computing, security is an inevitable challenge, and at the same time opportunity, for the wide adoption of near-data computing. On the one hand, with more and more information and logic being offloaded and flown across the entire system, we need to be extremely careful to avoid introducing new side channels. On the other hand, near-data computing may help to improve security. For example, a wide range of attacks exploits the on-chip resources such as caches, prefetchers, interconnects, etc. If sensitive computation is offloaded off-chip and performed near memory, all these attacks are naturally defended. Near-data computing could also help efficiently encrypt and decrypt data on the fly, providing higher security without impeding the performance. There are many open and exciting questions about the intersection between security and near-data computing.

Supporting General-Purpose Programs It is still an open question to efficiently support near-data computing for general-purpose programs. Many prior works simply offload the thread, which is still essentially compute-centric and does not align well with the nature of distributed computing for near-data computing. Near-stream computing can schedule streams independently at different locations and enables efficient coordination, but still has limited support for complex control flows. It is a challenging but rewarding direction to map arbitrary general-purpose program to near-data computing and achieve maximal performance and energy efficiency.

Final Thoughts

This work proves that explicit data access pattern and computation can be directly encoded in the ISA and decoupled from the core pipeline, enabling general and flexible near-data computing. The fundamental insight that data and computation should not be separated but expressed, scheduled, and orchestrated *together* has a profound impact on future general-purpose processor designs, and also sheds light on this promising path to continue the performance and energy efficiency scaling.

APPENDIX A

NDC Related Works

Table A.1 characterizes recent near-data techniques related to this dissertation ordered by where it performs the near-data computation (the `near where` column). Here we relate them to each dimension of the design space of near-data computing in §1.1.

Table A.1: Characterization of Near-Data Approaches

Begin of Table (Trans.: Transparent)

NDC Work	Yr.	ABST.	Near Where	Substrate	Domain	Program	Data Layout
Goal		**Unified**	**All**	**All**	**General**	**Trans.**	**Automatic**
Xulong Tang et al. [176]	’17	Thread	Core	Local Core	General	Trans.	Oblivious
CDCS [1]	’15	Thread	Core	Local Core	General	Trans.	Limited[1]
EM2 [216]	’15	Thread	Core	Local Core	General	Trans.	Oblivious
Dist-DA [2]	’22	DFG	LLC	Core/CGRA	General	Trans.	Oblivious
Omni-Compute [3]	’19	Inst.	GPU LLC	FU	General	Trans.	Oblivious
SCU [260]	’19	Kernel	GPU LLC	ASIC	Graph	API	Oblivious
SnackNoC [4]	’20	Kernel	NoC Router	FU	Regular	API	Scratchpad
Fafnir [5]	’21	Kernel	DRAM[2]	FU	Gather	API	Manual
GenASM [6]	’20	Kernel	DRAM	ASIC	Genomic	API	Scratchpad
CoNDA [165]	’19	Thread	DRAM	Core	Graph/DB	API	Oblivious
GraphPIM [261]	’17	Inst.	DRAM	FU	Graph	API	Manual
EMC [7]	’16	μop Seq.	DRAM	FU	Prefetching	Trans.	Oblivious
UPMEM [261]	’18	Kernel	DRAM	Core	General	API	Manual
SimplePIM [262]	’23	Kernel	DRAM	Core	General	API	Limited[3]
RecNMP [263]	’20	Inst.	DRAM	ASIC	Recommend	OpenCL	Manual

[1] Only at page granularity.

[2] In interconnects of DDR ranks and channels.

[3] Only coarse-grained continuous arrays.

Continuation of Table A.1 (Trans.: Transparent)

NDC Work	Yr.	ABST.	Near Where	Substrate	Domain	Program	Data Layout
Goal		**Unified**	**All**	**All**	**General**	**Trans.**	**Automatic**
MViD [264]	'20	Kernel	DRAM	FU	SpMV	API	Manual
ABC-DIMM [265]	'21	Kernel	DRAM	FU	Graph	API	Manual
DIMM-Link [266]	'23	Kernel	DRAM	FU	General	API	Manual
TRiM [267]	'21	Kernel	DDR Bank	FU	Recommend	API	Scratchpad
To PIM or Not [8]	'22	Thread	DDR Bank	Core	General	Trans.	Oblivious[4]
MeNDA [9]	'22	Kernel	DDR Rank	ASIC	Sparse LA[5]	API	Manual
Newton [268]	'20	Kernel	DDR Rank	FU	ML	API	Manual
Tesseract [10]	'15	Thread	HMC	Core	Graph	API	Manual
Mondrian [137]	'17	Thread	HMC	Core	DB	API	Manual
Active Memory Cube [142]	'15	Thread	HMC	Core	General	Pragma	Manual
Byungchul Hong et al. [145]	'16	Kernel	HMC	FU	LLT[6]	API	Limited[7]
TOM [11]	'16	Thread	HMC	GPU Core	General	Trans.	Limited[8]
GPU-PIM [12]	'16	Thread	HMC	GPU Core	General	Trans.	Oblivious
Near-Data SIMD Unit [269]	'17	Thread	HMC	GPU Core	General	Trans.	Oblivious
ABNDP [13]	'23	Thread	HMC	Core	General	API	Oblivious[9]
PIM-Enabled Inst. [14]	'15	Inst.	HMC	FU	General	Trans.	Oblivious
IMPICA [15]	'16	Kernel	HMC	ASIC	Ptr-Chasing	API	Oblivious
NeuroStream [270]	'17	Kernel	HMC	Core	CNN	API	Manual
Active Routing [16]	'19	Packet	HMC	FU	Aggregation	API	Oblivious
Neurocube [160]	'16	Kernel	HMC	ASIC	NN[10]	API	Manual
Gearbox [17]	'22	Kernel	HMC Bank	ASIC	Sparse LA[11]	API	Manual
FANS [18]	'21	Kernel	SSD	FPGA	Sorting	API	Manual
ASSASIN [19]	'22	Kernel	SSD	ASIC	General[12]	API	Oblivious
Summarizer [166]	'17	Kernel	SSD	Core	General	API	Manual

[4] Customized physical address layout in DRAM.

[5] Sparse matrix transposition.

[6] Linked list traversal.

[7] Specific to linked lists.

[8] Specific to strided patterns on GPU.

[9] With DRAM-based cache to capture locality.

[10] Neural network.

[11] Sparse linear algebra, mainly SpMV and SpMSpV.

[12] Programs need to be transformed into streaming computing.

Continuation of Table A.1 (Trans.: Transparent)

NDC Work	Yr.	ABST.	Near Where	Substrate	Domain	Program	Data Layout
Goal		**Unified**	**All**	**All**	**General**	**Trans.**	**Automatic**
GraFBoost [271]	’18	Kernel	SSD	FPGA	Graph	API	Oblivious
GraphSSD [272]	’19	Inst.	SSD	ASIC	Graph	API	Limited[13]
Neural Cache [20]	’18	Kernel	In-LLC	Bitline	ML	API	Scratchpad
Duality Cache [21]	’19	SIMT	In-LLC	Bitline	General	Trans.	Oblivious
Compute Cache [149]	’17	Inst.	In-Cache[14]	Bitline	Limited[15]	API	Oblivious
PIM-DH [273]	’22	Kernel	ReRAM	Bitline	Deep Hashing	API	Manual
Ben Feinberg et al. [274]	’18	Kernel	ReRAM	Bitline	Sparse LA	API	Manual
FloatPIM [248]	’19	Kernel	ReRAM	Bitline	CNN	API	Manual
SIMDRAM [242]	’21	Kernel	In-DRAM	Bitline	General	API	Scratchpad
DUAL [22]	’20	Kernel	In-DRAM	Bitline	Clustering	API	Manual
GraphiDe [275]	’19	Inst.	In-DRAM	Bitline	Graph	API	Scratchpad
Ambit [23]	’17	Inst.	In-HMC	Bitline	General	API	Scratchpad
SISA [24]	’21	Set	Multi.[16]	Multi.[17]	Graph Mining	API	Manual
MLIMP [25]	’22	DFG	Multi.[18]	Multi.	GEMM/GNN	API	Scratchpad
NDC Compiler [26]	’21	Inst.	Multi.[19]	FU	General	Trans.	Oblivious
Livia [27]	’20	Kernel[20]	Multi.[21]	Core/FPGA	General	API	Oblivious
SSP [28]	’19	Stream	Core	FU	Prefetching	Trans.	Oblivious
Stream Floating [32]	’21	Stream	LLC	FU	Prefetching	Trans.	Oblivious
NSC [31]	’22	Stream	LLC	Local Core	General	Trans.	Oblivious
Infinity Stream [30]	’23	Stream	Multi.[22]	Multi.[23]	General	Pragma	Scratchpad
Affinity Alloc [33]	’23	Stream	LLC	FU	General	API	Automatic

[13] Do not optimize affinity between vertices and edge.

[14] It also adds FUs near cache controllers when there is no operand locality.

[15] Since data is not transposed, it only supports a limited set of operations.

[16] In-situ DRAM (SISA-PUM) and near DRAM controller (SISA-PNM).

[17] In-situ DRAM bitline for SISA-PUM, and small in-order cores for SISA-PNM.

[18] In-situ LLC, In-situ DRAM and ReRAM.

[19] NoC routers, LLC controllers, DRAM controllers, inside DRAM.

[20] Can only process a single cache line.

[21] LLC controllers and DRAM controllers

[22] In-site LLC and Near LLC controller.

[23] In-situ SRAM bitline and near LLC FU (stream engine).

Continuation of Table A.1 (Trans.: Transparent)

NDC Work	Yr.	ABST.	Near Where	Substrate	Domain	Program	Data Layout
Goal		**Unified**	**All**	**All**	**General**	**Trans.**	**Automatic**

End of Table

- **Offloading Location:** Many works can be categorized as "near-memory computing", as they offload computation near the controller of the cache, DRAM, HMC, or SSD. Emerging technologies such as in-situ bitline operation offers massive data parallelism by directly turning the SRAM/DRAM arrays into vector units (labeled as `In-X` in Table A.1). Also, we include those multi-core works that do not offload computation from the core, but aim to schedule and migrate the computation *among* the cores to improve the data locality (labeled as `Core` in `Near Where` column). These techniques do not introduce new computing units, but reuse the existing cores. Some works also support scheduling the computation among multiple locations, which offers more flexibility.

- **Computing Substrate:** Different works also leverage different computing substrates to perform the near-data operation. In-situ techniques simply use the SRAM/DRAM bitlines, while those near-memory works varies from small general-purpose in-order cores to fixed-function ASIC. In between we also have reconfigurable logic such as FPGA/CGRA as well as tailored FUs that can be programmed to perform some predefined compute primitives.

- **Application Domain:** Many prior works are specialized for a specific application domain, such as neural networks, GEMM, graph processing, graph mining, and database, etc. These applications are usually memory intensive and can benefit the most from the improved memory bandwidth of near-data computing. Other works strive to support general near-data computation.

- **Programming Model:** Most prior works require manual programming using specific APIs, or at least some pragma hints from the compiler. Clearly this poses significant challenges for the broad adoption of near-data computing. Only a few works provide programmer-transparency leveraging coarse-grained thread/SIMT abstraction as the offloading unit. However, as discussed in §5.1, these coarse-grained abstractions fall short of capturing the inherent distributed property of near-data computing and leads to suboptimal performance.
- **Data Layout:** Finally, most works are either oblivious to the data layout and simply abort near-data computing when there is not sufficient data locality, or require manual data placement which is error-prone and tedious. Affinity alloc [33] is the first work to systematically and automatically optimize the data layout to improve data affinity.

Goal and Our Work As shown in the top entry of Table A.1, an ideal near-data computing system should provide a unified and general abstraction across all available offloading locations and computing substrates, while remaining transparent to programmers and automatically optimizing the data layout. This is an ambitious goal. The bottom of Table A.1 lists our works, and each of them made some advancement across different dimensions. We believe this dissertation is just the beginning of realizing general, flexible and unified near-data computing.

APPENDIX B

tDFG Optimization

Here we discuss the rewrite rules and equality-graph approach to optimizing the tDFG.

Intuition A unique aspect of optimizing the tDFG is the need to reason about the tensor domains (i.e. the hyperrectangle in lattice space). For example, two same element-wise computations on tensor `A[1, n)` and `A[0, n-1)` can be merged into a single computation on `A[0, n)`, provided that the tensor size information is correctly tracked after we slightly expanded the computed tensor. This cuts the computation by half. More generally, there is a large transformation space with many equivalent tDFGs producing the same result, and the compiler needs to efficiently search for the optimal tDFG with less data traffic and computation. We first introduce the tDFG equivalence rules used to rewrite the tDFG, followed by an optimized example and details in our implementation.

tDFG Equivalence Rules We define two tDFG nodes to be equivalent if they represent the same result and share the same domain in the lattice space. To transform the tDFG, we now formalize the tDFG equivalence rules, with these notations:

- $\mathbb{T}, \mathbb{C}, \mathbb{M}, \mathbb{B}$: Tensor, compute, move, and broadcast node respectively, with their definition and semantics in Fig 7.5 (page 165). Note that all these nodes produce a *tensor*, while $\mathbb{T}$ constructs the input tensor from the input array.
- A, B, C: Arbitrary tensors in the tDFG, e.g. compute, move, broadcast node.
- i, j: Operated dimension, e.g. move, broadcast.

- p_i, q_i: Range of the i^{th} dimension $[p_i, q_i)$.
- f: Computation applied to input tensors.

As a simple example, Eq. B.1a defines the associative rule for compute node, when the operation f is associative by itself, i.e. $f(f(a,b),c) \Leftrightarrow f(a,f(b,c))$. Similarly, Eq. B.1b defines the commutative rule for compute node when the operation f is commutative, e.g. addition, multiplication. We can also define the distributive rule similar to $a \times (x+y) \Leftrightarrow a \times x + b \times y$ (Eq. B.1c).

$$\mathbb{C}(f, \mathbb{C}(f, A, B), C) \Leftrightarrow \mathbb{C}(f, A, \mathbb{C}(f, B, C)) \tag{B.1a}$$

$$\mathbb{C}(f, A, B) \Leftrightarrow \mathbb{C}(f, B, A) \tag{B.1b}$$

$$\mathbb{C}(f, \mathbb{C}(g, A), \mathbb{C}(g, B)) \Leftrightarrow \mathbb{C}(g, \mathbb{C}(f, A, B)) \tag{B.1c}$$

Exchanging Compute and Move/Broadcast Eq. B.2a defines the commutative rule to exchange a unary compute node and a move node. Recall that a move node shifts the tensor along a certain dimension by some distance in the lattice space. Therefore, the move operation can happen before or after the computation, i.e. it is commutative with compute nodes. Similarly, when the compute node takes multiple operands, a move node is applied to every input tensor. Also, Eq. B.2b shows the commutative rule for a compute node and a broadcast node.

$$\mathbb{C}(f, \mathbb{M}(A, i, dist)) \Leftrightarrow \mathbb{M}(\mathbb{C}(f, A), i, dist) \tag{B.2a}$$

$$\mathbb{C}(f, \mathbb{B}(A, i, dist, cnt)) \Leftrightarrow \mathbb{B}(\mathbb{C}(f, A), i, dist, cnt) \tag{B.2b}$$

Expanding and Shrinking Tensor To reuse common computation results, it may be necessary to expand a tensor. For example, $\mathbb{C}(f, \mathbb{T}(1, N))$ and $\mathbb{C}(f, \mathbb{T}(0, N))$ share common results on the domain $[1, N)$. However, they are not equivalent as the first computation is applied to a slightly smaller tensor. If we can expand the first tensor to $\mathbb{T}(0, N)$, we can reduce the operations from $2N - 1$ to N.

To maintain equivalence, an expanded tensor must be later shrunk to the original domain. Therefore, we introduce a shrink node, $\mathbb{S}$, which resizes the tensor along dimension i to have a new domain $[p_i, q_i)$.

Putting these together, Eq. B.3 shows the rule to expand a smaller tensor of size $[p_i, q_i)$ in the i^{th} dimension into a larger tensor of size $[p'_i, q'_i)$, where $p'_i <= p_i$ and $q'_i >= q_i$. The shrink node returns the output tensor to the original domain, hence it is equivalent to the original tensor. Shrink nodes are only for tracking the tensor size information, and are lowered to a nop by the JIT compiler (similar to how the ϕ nodes are not lowered to instructions in SSA IR [37]). We omit shrink nodes in the paper for simplicity, as they are only needed during optimization.

$$\begin{aligned} \mathbb{T}(..., p_i, q_i, ...) \Leftrightarrow\ & \mathbb{S}(i, p_i, q_i, \mathbb{T}(..., p'_i, q'_i, ...)) \\ & \text{where } p'_i <= p_i, q'_i >= q_i \end{aligned} \tag{B.3}$$

Exchanging Shrink and Other Nodes A shrink node by itself is not sufficient to unlock the optimization opportunities in the tDFG. We need to define how it interacts with other tDFG nodes. Eq. B.4a is a straightforward rule that two shrink nodes on different dimensions are commutable. When they operate on the same dimension, we can combine them into a single shrink node by taking the intersection, as in Eq. B.4b.

$$\begin{aligned} \mathbb{S}(i, p_i, q_i, \mathbb{S}(j, p_j, q_j, A)) \Leftrightarrow\ & \mathbb{S}(j, p_j, q_j, \mathbb{S}(i, p_i, q_i, A)) \\ & \text{when } i \neq j \end{aligned} \tag{B.4a}$$

$$\mathbb{S}(i, p_i, q_i, \mathbb{S}(i, p'_i, q'_i, A)) \Leftrightarrow\ \mathbb{S}(i, \max(p_i, p'_i), \min(q_i, q'_i), A) \tag{B.4b}$$

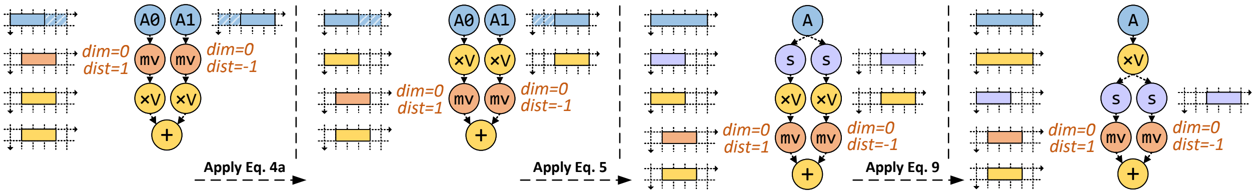


Figure B.1: Example of Applying Rewrites

Similarly, shrink node and move node on different dimensions are commutable (Eq. B.5a). If they are on the same dimension, we can also apply a shrink node on the moved tensor with the shifted domain $[p_i + dist, q_i + dist)$.

$$\mathbb{M}(\mathbb{S}(i, p_i, q_i, A), j, dist) \Leftrightarrow \mathbb{S}(i, p_i, q_i, \mathbb{M}(A, j, dist)) \quad \text{when } i \neq j \tag{B.5a}$$

$$\mathbb{M}(\mathbb{S}(i, p_i, q_i, A), i, dist) \Leftrightarrow \mathbb{S}(i, p_i + dist, q_i + dist, \mathbb{M}(A, i, dist)) \tag{B.5b}$$

This also applies to broadcast node and shrink node: they are commutable if on different dimension (Eq. B.6a). When they are on the same dimension, we can combine them by directly broadcasting to the shrunken region.

$$\mathbb{B}(\mathbb{S}(i, p_i, q_i, A), j, dist, cnt) \Leftrightarrow \mathbb{S}(i, p_i, q_i, \mathbb{B}(A, j, dist, cnt)) \quad \text{when } i \neq j \tag{B.6a}$$

$$\mathbb{S}(i, p_i, q_i, \mathbb{B}(A, i, dist, cnt)) \Leftrightarrow \mathbb{B}(A, i, p_i, q_i - p_i) \tag{B.6b}$$

Finally, a shrink node is also commutable with the compute node (Eq. B.7).

$$\mathbb{S}(i, p_i, q_i, \mathbb{C}(f, A)) \Leftrightarrow \mathbb{C}(f, \mathbb{S}(i, p_i, q_i, A)) \tag{B.7}$$

Optimization Example Fig B.1 shows an example of applying our rewrite rules to discover opportunities for reuse. The original tDFG first moves the input tensor `A` left and right by one before applying a constant element-wise multiply to both tensors. Since in-memory processing applies element-wise functions to all elements, we can save a redundant compute by first performing the computation on the entirety of tensor `A` before realigning the result.

We begin with the original tDFG. By rule B.2a, we can commute the move and compute nodes.

$$\begin{aligned}
&\mathbb{C}(+, \mathbb{C}(\times V, \mathbb{M}(\mathbb{T}(0, n-2), 0, 1)),\\
&\quad \mathbb{C}(\times V, \mathbb{M}(\mathbb{T}(2, n), 0, -1)))\\
\xrightarrow{\text{Eq. } B.2a}\ &\mathbb{C}(+, \mathbb{M}(\mathbb{C}(\times V, \mathbb{T}(0, n-2)), 0, 1),\\
&\quad \mathbb{M}(\mathbb{C}(\times V, \mathbb{T}(2, n)), 0, -1))
\end{aligned}$$

We can expand the two tensor $\mathbb{T}$s to the entire domain of array `A` with rule B.3. By commuting the shrink $\mathbb{S}$ nodes and compute $\mathbb{C}$ nodes with rule B.7, we can discover a common subexpression, indicating there is an opportunity for compute reuse.

$$\begin{aligned}
\xrightarrow{\text{Eq. } B.3}\ &\mathbb{C}(+, \mathbb{M}(\mathbb{C}(\times V, \mathbb{S}(0, 0, n-2, \mathbb{T}(0, n))), 0, 1),\\
&\quad \mathbb{M}(\mathbb{C}(\times V, \mathbb{S}(0, 2, n, \mathbb{T}(0, n))), 0, -1))\\
\xrightarrow{\text{Eq. } B.7}\ &\mathbb{C}(+, \mathbb{M}(\mathbb{S}(0, 0, n-2, \mathbb{C}(\times V, \mathbb{T}(0, n))), 0, 1),\\
&\quad \mathbb{M}(\mathbb{S}(0, 2, n, \mathbb{C}(\times V, \mathbb{T}(0, n))), 0, -1))
\end{aligned}$$

Fig 7.6 (page 168) shows a more complicated example of optimized tDFG. To see how the equivalence rules rewrite the program, first expand all the tensors to the full array, and exchange the shift nodes to the final output of the tDFG (ommitted in Fig 7.6). Use Eq. B.1b and Eq. B.1a to add `v0` and `v2` together. Since we are multiplying by a constant C_0 and C_1, we can use distributive rule to swap the addition and multiplication. The optimized tDFG reuses the computated results and avoids unnecessary data movements.

Equality Graphs We leverage equality graphs (e-graphs) to efficiently search the optimal tDFG in the design space. Equality graphs represent all possible rewrites of an expression tree. Given a rewrite rule $e_1 \rightarrow e_2$ for two expressions e_1, e_2, an e-graph will apply it to all matches in its underlying expression tree. These *nondestructive* updates are performed by marking e_1 and e_2 as equivalent. Given a set of rewrite rules, all possible permutations of the original expression tree are discovered by continuously applying them. The final tDFG selection is based on architecture-informed cost metrics combining the estimated latency of move vs. compute node, the amount of moved/broadcast data, as well as the number of computations.

REFERENCES

[1] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 538–550, 2015.

[2] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1160–1177, 2022.

[3] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. Opportunistic computing in gpu architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 210–223, 2019.

[4] Karthik Sangaiah, Michael Lui, Ragh Kuttappa, Baris Taskin, and Mark Hempstead. Snacknoc: Processing in the communication layer. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 461–473, 2020.

[5] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *HPCA*, 2021.

[6] D. S. Cali, G. S. Kalsi, Z. Bingöl, 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 *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[7] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale 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)*, pages 444–455. IEEE, 2016.

[8] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To pim or not for emerging general purpose processing in ddr memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 231–244, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Menda: A near-memory multi-way merge solution for sparse transposition and dataflows. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 245–258, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[11] Kevin Hsieh, Eiman Ebrahim, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2016.

[12] 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 *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.

[13] Boyu Tian, Qihang Chen, and Mingyu Gao. Abndp: Co-optimizing data access and load balance in near-data processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 3–17, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 336–348, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32, 2016.

[16] Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum, and Eun Jung Kim. Active-routing: Compute on the way for near-data processing. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 674–686. IEEE, 2019.

[17] Marzieh Lenjani, Alif Ahmed, Mircea Stan, and Kevin Skadron. Gearbox: A case for supporting accumulation dispatching and hybrid partitioning in pim-based accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 218–230, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. Fans: Fpga-accelerated near-storage sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 106–114, 2021.

[19] Chen Zou and Andrew A. Chien. Assasin: Architecture support for stream computing to accelerate computational storage. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 354–368, 2022.

[20] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 383–396. IEEE Press, 2018.

[21] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 397–410, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. Dual: Acceleration of clustering algorithms using digital-based processing in-memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[23] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287, 2017.

[24] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 282–297, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Daichi Fujiki, Alireza Khadem, Scott Mahlke, and Reetuparna Das. Multi-layer in-memory processing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 920–936, 2022.

[26] Mahmut Taylan Kandemir, Jihyun Ryoo, Xulong Tang, and Mustafa Karakoy. Compiler support for near data computing. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 90–104, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan 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*, ASPLOS '20, page 417–433, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 736–749, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Zhengrong Wang, Christopher Liu, and Tony Nowatzki. Infinity stream: Enabling transparent and automated in-memory computing. *IEEE Computer Architecture Letters*, 21(2):85–88, 2022.

[30] Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. Infinity stream: Portable and programmer-friendly in-/near-memory fusion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 359–375, New York, NY, USA, 2023. Association for Computing Machinery.

[31] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 331–345, 2022.

[32] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. Stream floating: Enabling proactive and decentralized cache optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 640–653, 2021.

[33] Zhengrong Wang, Christopher Liu, Nathan Beckmann, and Tony Nowatzki. Affinity alloc: Taming not-so near-data computing. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[34] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[35] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, Oct 2009.

[36] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. Cortexsuite: A synthetic brain benchmark suite. In *IISWC*, pages 76–79, 2014.

[37] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88.

[38] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 118–130, New York, NY, USA, 2015. ACM.

[39] Naveen Vedula, Arrvindh Shriraman, Snehasish Kumar, and William N Sumner. Nachos: Software-driven hardware-assisted memory disambiguation for accelerators. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 710–723. IEEE, 2018.

[40] Ruke Huang, Alok Garg, and Michael Huang. Software-hardware cooperative memory disambiguation. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 244–253. IEEE, 2006.

[41] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[42] Neal Clayton Crago and Sanjay Jeram Patel. Outrider: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[43] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[44] Ilhyun Kim and Mikko H Lipasti. Understanding scheduling replay schemes. In *Software, IEE Proceedings-*, pages 198–209. IEEE, 2004.

[45] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[46] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 97–108, Piscataway, NJ, USA, 2014. IEEE Press.

[47] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Y. Wei, and D. Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[48] Tony Nowatzki and Karthikeyan Sankaralingam. Analyzing behavior specialized acceleration. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 697–711. ACM, 2016.

[49] T. Nowatzki, V. Govindaraju, and K. Sankaralingam. A graph-based program representation for analyzing hardware specialization approaches. *IEEE Computer Architecture Letters*, 14(2):94–98, July 2015.

[50] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO '09*.

[51] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 14–25. IEEE, 2001.

[52] Amir Roth and Gurindar S Sohi. Speculative data-driven multithreading. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 37–48. IEEE, 2001.

[53] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. *ACM SIGARCH Computer Architecture News*, 29(2):2–13, 2001.

[54] Sushant Kondguli and Michael Huang. Bootstrapping: Using smt hardware to improve single-thread performance. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 687–700, New York, NY, USA, 2019. ACM.

[55] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 52–61, June 2001.

[56] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1309–1324, 2009.

[57] Weifeng Zhang, Dean M Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 85–95. IEEE, 2007.

[58] Alok Garg and Michael C Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 306–317. IEEE Computer Society, 2008.

[59] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, pages 3–14, September 2001.

[60] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE micro*, 21(2):35–46, 2001.

[61] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (rsvp). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 141–, Washington, DC, USA, 2003. IEEE Computer Society.

[62] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[63] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 416–429, New York, NY, USA, 2017. ACM.

[64] Nathan Clark, Amir Hormati, and Scott Mahlke. Veal: Virtualized execution accelerator for loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[65] Gabriel Weisz and James C Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.

[66] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 207–220. IEEE, 2018.

[67] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus. ISCA, 2018.

[68] L. Kurian, P. T. Hulina, and L. D. Coraor. Memory latency effects in decoupled architectures with a single data memory module. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 236–245, May 1992.

[69] Lizy Kurian John, Vinod Reddy, Paul T Hulina, and Lee D Coraor. Program balance and its impact on high performance risc architectures. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 370–379. IEEE, 1995.

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

[71] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, September 2012.

[72] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO*, 2004.

[73] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman. Chainsaw: Von-neumann accelerators to leverage fused instruction chains. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2016.

[74] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ASPLOS '10*.

[75] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman. Needle: Leveraging program analysis to analyze and extract accelerators from whole programs. pages 565–576, Feb 2017.

[76] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, et al. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, 2011.

[77] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel 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*, ASPLOS '19, pages 137–151, New York, NY, USA, 2019. ACM.

[78] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186. ACM, 1991.

[79] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 178–190. ACM, 2015.

[80] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 252–263. IEEE Computer Society, 2006.

[81] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. *SIGARCH Comput. Archit. News*, 33(2):222–233, May 2005.

[82] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 69–80, New York, NY, USA, 2009. ACM.

[83] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–259. ACM, 2013.

[84] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18(2SI):364–373, 1990.

[85] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 42–53, New York, NY, USA, 2000. ACM.

[86] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 40–52. ACM, 1991.

[87] N. Kohout, Seungryul Choi, Dongkeun Kim, and D. Yeung. Multi-chain prefetching: effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 268–279, 2001.

[88] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. Comput. Syst.*, 22(2):214–280, May 2004.

[89] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, April 2008.

[90] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 81–92, New York, NY, USA, 2011. ACM.

[91] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. Adaptive gpu cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 25–35, New York, NY, USA, 2015. ACM.

[92] 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)*, pages 293–304, Sep. 2012.

[93] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[94] 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*, pages 222–233, Nov 2008.

[95] 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)*, pages 76–88, Feb 2015.

[96] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores. *arXiv preprint arXiv:1911.08356*, 2019.

[97] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. Soda: stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[98] Intel. Intel virtualization technology for directed i/o, architecture specification. June 2019.

[99] AMD. Amd i/o virtualization technology (iommu) specification. December 2016.

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

[101] 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)*, pages 37–48, 2017.

[102] 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*, pages 85–96, 2001.

[103] Valentina Salapura, Matthias Blumrich, and Alan 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*, MEDEA '07, pages 25–32, New York, NY, USA, 2007. ACM.

[104] Jason Zebchuk, Elham Safi, and Andreas 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*, MICRO 40, pages 314–327, Washington, DC, USA, 2007. IEEE Computer Society.

[105] Andreas Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. *SIGARCH Comput. Archit. News*, 33(2):234–245, May 2005.

[106] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[107] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. In *CoRR*, volume abs/2007.03152, 2020.

[108] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj 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*, pages 33–42, 2009.

[109] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, page 44–54, USA, 2009. IEEE Computer Society.

[110] 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)*, pages 399–411, Feb 2019.

[111] Seth Pugsley. 3rd data prefetching championship. June 2019.

[112] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 924–939, New York, NY, USA, 2019. Association for Computing Machinery.

[113] 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)*, pages 703–716, 2020.

[114] 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)*, pages 268–281, 2020.

[115] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale 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)*, pages 444–455. IEEE, 2016.

[116] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 144–154. IEEE, 2001.

[117] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 209–220. IEEE, 2002.

[118] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186. IEEE, 2010.

[119] Andreas Sembrant, Erik Hagersten, and David 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)*, pages 133–144, Piscataway, NJ, USA, 2014. IEEE Press.

[120] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, 2007.

[121] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010.

[122] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr, and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219, 2008.

[123] 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*, pages 328–339, 2007.

[124] Abdullah Kayi and Tarek El-Ghazawi. An adaptive cache coherence protocol for chip multiprocessors. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, New York, NY, USA, 2010. Association for Computing Machinery.

[125] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. *SIGARCH Comput. Archit. News*, 21(2):98–108, May 1993.

[126] Håkan Grahn and Per Stenström. Evaluation of a competitive-update cache coherence protocol with migratory data detection. *Journal of Parallel and Distributed Computing*, 39(2):168 – 180, 1996.

[127] Guru Venkataramani, Christopher J. Hughes, Sanjeev Kumar, and Milos Prvulovic. Deft: Design space exploration for on-the-fly detection of coherence misses. *ACM Trans. Archit. Code Optim.*, 8(2), June 2011.

[128] Guru Venkataramani, Christopher J. Hughes, Sanjeev Kumar, and Milos Prvulovic. Deft: Design space exploration for on-the-fly detection of coherence misses. *ACM Trans. Archit. Code Optim.*, 8(2), June 2011.

[129] 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*, pages 204–215, 1997.

[130] 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*, pages 1–12, 2015.

[131] 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*, pages 63–74, 2007.

[132] Sam Ainsworth and Timothy M Jones. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 305–317. IEEE, 2017.

[133] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong 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)*, pages 96–109. IEEE, 2018.

[134] Sam Ainsworth and Timothy M Jones. An event-triggered programmable prefetcher for irregular workloads. *ACM SIGPLAN Notices*, 53(2):578–592, 2018.

[135] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and A. Arvind. Aquoman: An analytic-query offloading machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[136] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data

movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.

[137] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The mondrian data engine. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[138] Ivan Fernandez, Ricardo Quislant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. Natsa: A near-data processing accelerator for time series analysis. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020.

[139] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015.

[140] Mingyu Gao and Christos Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2016.

[141] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, 2017.

[142] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3), 2015.

[143] Po-An Tsai, Changping Chen, and Daniel Sanchez. Adaptive scheduling for systems with asymmetric memory hierarchies. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[144] 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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, 2015.

[145] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. Accelerating linked-list traversal through near-data processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, New York, NY, USA, 2016. Association for Computing Machinery.

[146] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 251–264. IEEE Press, 2021.

[147] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. Data-Centric Execution of Speculative Parallel Programs. In *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49)*, October 2016.

[148] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient speculative parallelism for accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[149] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492. IEEE, 2017.

[150] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 209–222. IEEE Press, 2021.

[151] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667, 2021.

[152] Mario Nemirovsky and Dean M Tullsen. Multithreading architecture. *Synthesis Lectures on Computer Architecture*, 8(1):1–109, 2013.

[153] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery.

[154] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 278–289, New York, NY, USA, 2007. Association for Computing Machinery.

[155] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence, second edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[156] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Comput. Archit. Lett.*, 19(2):106–109, jul 2020.

[157] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *ISWC '06*, pages 182–188.

[158] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.

[159] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *HPCA*, 2021.

[160] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.

[161] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[162] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[163] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[164] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Syncron: Efficient synchronization support for near-data-processing architectures. In *HPCA*, 2021.

[165] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, 2019.

[166] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 219–231, New York, NY, USA, 2017. Association for Computing Machinery.

[167] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-memory data parallel processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery.

[168] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301. IEEE, 2017.

[169] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[170] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. Fractal: An execution model for fine-grain nested speculative parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[171] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 217–230, 2018.

[172] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. T4: Compiling sequential code for effective speculative parallelization in hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 159–172. IEEE, 2020.

[173]

[174] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 595–608. IEEE Press, 2021.

[175] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. Dual: Acceleration of clustering algorithms using digital-based processing in-memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 356–371, 2020.

[176] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. Data movement aware computation partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 730–744, New York, NY, USA, 2017. Association for Computing Machinery.

[177] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Whirlpool: Improving dynamic cache management with static data classification. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 113–127, New York, NY, USA, 2016. Association for Computing Machinery.

[178] Amd epyc 7773x, 2023.

[179] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi. Dalorex: A data-local program execution and architecture for memory-bound applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 718–730, Los Alamitos, CA, USA, mar 2023. IEEE Computer Society.

[180] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[181] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News*, 41(3):237–248, June 2013.

[182] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 66–78, New York, NY, USA, 2015. Association for Computing Machinery.

[183] Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism*

in Algorithms and Architectures, SPAA '15, page 80–82, New York, NY, USA, 2015. Association for Computing Machinery.

[184] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[185] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):14, 2017.

[186] Benedek Rozemberczki and Rik Sarkar. Twitch gamers: a dataset for evaluating proximity preserving and structural role-based node embeddings, 2021.

[187] Julian McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 539–547, Red Hook, NY, USA, 2012. Curran Associates Inc.

[188] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 133–145, New York, NY, USA, 2023. Association for Computing Machinery.

[189] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 513–527, New York, NY, USA, 2021. Association for Computing Machinery.

[190] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: Efficient Ad-Hoc analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 337–355. USENIX Association, April 2021.

[191] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Tripoline: Generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 17–32, New York, NY, USA, 2021. Association for Computing Machinery.

[192] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International*

Conference on Management of Data, SIGMOD '16, page 417–430, New York, NY, USA, 2016. Association for Computing Machinery.

[193] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1372–1385, New York, NY, USA, 2021. Association for Computing Machinery.

[194] Wole Jaiyeoba and Kevin Skadron. Graphtinker: A high performance data structure for dynamic graph processing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1030–1041, 2019.

[195] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 211–222, New York, NY, USA, 2002. Association for Computing Machinery.

[196] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 455–468, 2006.

[197] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. *SIGARCH Comput. Archit. News*, 37(3):184–195, June 2009.

[198] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 250–261, 2009.

[199] Mainak Chaudhuri. Pagenuca: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 227–238, 2009.

[200] Lei Jin and Sangyeun Cho. Sos: A software-oriented distributed shared cache management approach for chip multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 361–371, 2009.

[201] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 538–550, 2015.

[202] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 213–224, Piscataway, NJ, USA, 2013. IEEE Press.

[203] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 538–550. IEEE, 2015.

[204] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 652–665, New York, NY, USA, 2017. ACM.

[205] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. Nexus: A new approach to replication in distributed shared caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 166–179, 2017.

[206] B.M. Beckmann and D.A. Wood. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 319–330, 2004.

[207] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. Asr: Adaptive selective replication for cmp caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 443–454, 2006.

[208] Haakon Dybdahl and Per Stenstrom. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 2–12, 2007.

[209] Z. Chishti, M.D. Powell, and T.N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 55–66, 2003.

[210] Z. Chishti, M.D. Powell, and T.N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 357–368, 2005.

[211] Javier Merino, Valentin Puente, and Jose A. Gregorio. Esp-nuca: A low-cost adaptive non-uniform cache architecture. In *2010 16th International Symposium on High-Performance Computer Architecture (HPCA'10)*, pages 1–10, 2010.

[212] Brian C. Schwedock and Nathan Beckmann. Jumanji: The case for dynamic nuca in the datacenter. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 665–680, 2020.

[213] Yaosheng Fu, Tri M Nguyen, and David Wentzlaff. Coherence domain restriction on large scale systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 686–698. IEEE, 2015.

[214] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. X-cache: A modular architecture for domain-specific caches. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 396–409, New York, NY, USA, 2022. Association for Computing Machinery.

[215] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. Rethinking the memory hierarchy for modern languages. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 203–216. IEEE Press, 2018.

[216] Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. The execution migration machine: Directoryless shared-memory architecture. *Computer*, 48(9):50–59, sep 2015.

[217] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. A scalable architecture for reprioritizing ordered parallelism. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 437–453, New York, NY, USA, 2022. Association for Computing Machinery.

[218] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. Massive data-centric parallelism in the chiplet era, 2023.

[219] Vidushi Dadu and Tony Nowatzki. Taskstream: Accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1–13, New York, NY, USA, 2022. Association for Computing Machinery.

[220] Quan M. Nguyen and Daniel Sanchez. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1064–1077, New York, NY, USA, 2021. Association for Computing Machinery.

[221] Quan M. Nguyen and Daniel Sanchez. Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1262–1274, 2023.

[222] Mahmut Taylan Kandemir, Xulong Tang, Hui Zhao, Jihyun Ryoo, and Mustafa Karakoy. Distance-in-time versus distance-in-space. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 665–680, New York, NY, USA, 2021. Association for Computing Machinery.

[223] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–921, 2020.

[224] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[225] Daichi Fujiki, Xiaowei Wang, Arun Subramaniyan, and Reetuparna Das. In-/near-memory computing. *Synthesis Lectures on Computer Architecture*, 16:1–140, 08 2021.

[226] Charles Gregory Nelson. *Techniques for Program Verification.* PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[227] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications*, RTA'05, page 453–468, Berlin, Heidelberg, 2005. Springer-Verlag.

[228] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[229] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[230] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 5105–5114, Red Hook, NY, USA, 2017. Curran Associates Inc.

[231] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. Don't forget the memory: Automatic block ram modelling, optimization, and architecture exploration. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 115–124, New York, NY, USA, 2017. Association for Computing Machinery.

[232] Mohsen Imani, Saransh Gupta, and Tajana Rosing. Genpim: Generalized processing in-memory to accelerate data intensive applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1155–1158, 2018.

[233] Jingyao Zhang and Elaheh Sadredini. Inhale: Enabling high-performance and energy-efficient in-sram cryptographic hash for iot. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2022.

[234] Jingyao Zhang, Hoda Naghibijouybari, and Elaheh Sadredini. Sealer: In-sram aes for high-performance and low-overhead memory encryption. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '22, New York, NY, USA, 2022. Association for Computing Machinery.

[235] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. Overgen: Improving fpga usability through domain-specific overlay generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 35–56, 2022.

[236] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564, 2022.

[237] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[238] Jisung Park, Roknoddin Azizi, Geraldo F. Oliveira, Mohammad Sadrosadati, Rakesh Nadig, David Novo, Juan Gómez-Luna, Myungsuk Kim, and Onur Mutlu. Flash-cosmos: In-flash bulk bitwise operations using inherent computation capability of nand flash memory. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 937–955, 2022.

[239] Xin Xin, Youtao Zhang, and Jun Yang. Roc: Dram-based processing with reduced operation cycles. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[240] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 100–113, New York, NY, USA, 2019. Association for Computing Machinery.

[241] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Fracdram: Fractional values in off-the-shelf dram. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 885–899, 2022.

[242] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: A framework for bit-serial simd processing using

dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 329–345, New York, NY, USA, 2021. Association for Computing Machinery.

[243] Xiangjun Peng, Yaohua Wang, and Ming-Chang Yang. Chopper: A compiler infrastructure for programmable bit-serial simd processing using memory in dram. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1275–1288, 2023.

[244] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, mar 1997.

[245] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *12th Hot Chips Conference*, August 2000.

[246] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for pim based on commercial dram technology : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56, 2021.

[247] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.

[248] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 802–815, New York, NY, USA, 2019. Association for Computing Machinery.

[249] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *IEEE Journal of Solid-State Circuits*, 55(1):76–86, 2020.

[250] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 27–39. IEEE Press, 2016.

[251] Weitao Li, Pengfei Xu, Yang Zhao, Haitong Li, Yuan Xie, and Yingyan Lin. Timely: Pushing data movements and interfaces in pim accelerators towards local and in time

domain. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 832–845. IEEE Press, 2020.

[252] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 14–26. IEEE Press, 2016.

[253] Saransh Gupta, Mohsen Imani, Harveen Kaur, and Tajana Simunic Rosing. Nnpim: A processing in-memory architecture for neural network acceleration. *IEEE Trans. Comput.*, 68(9):1325–1337, sep 2019.

[254] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2019.

[255] Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy John. Wave-pim: Accelerating wave simulation using processing-in-memory. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[256] Yinan Li and Jignesh M. Patel. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 289–300, New York, NY, USA, 2013. Association for Computing Machinery.

[257] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. Bitfunnel: Revisiting signatures for search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, page 605–614, New York, NY, USA, 2017. Association for Computing Machinery.

[258] Nvidia. Nvidia H100 tensor core gpu architecture, 2022.

[259] Intel. Intel data streaming acceleartor architecture specification, 2022.

[260] Albert Segura, Jose-Maria Arnau, and Antonio González. Scu: A gpu stream compaction unit for graph processing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 424–435, New York, NY, USA, 2019. Association for Computing Machinery.

[261] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.

[262] Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu. Simplepim: A software framework for productive and efficient processing-in-memory. In *Proceedings of the 2023 International Conference on Parallel Architectures and Compilation*, PACT '23, New York, NY, USA, 2023. Association for Computing Machinery.

[263] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 790–803. IEEE Press, 2020.

[264] Byeongho Kim, Jongwook Chung, Eojin Lee, Wonkyung Jung, Sunjung Lee, Jaewan Choi, Jaehyun Park, Minbok Wi, Sukhan Lee, and Jung Ho Ahn. Mvid: Sparse matrix-vector multiplication in mobile dram for accelerating recurrent neural networks. *IEEE Trans. Comput.*, 69(7):955–967, jul 2020.

[265] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. Abc-dimm: Alleviating the bottleneck of communication in dimm-based near-memory processing with inter-dimm broadcast. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 237–250, 2021.

[266] Zhe Zhou, Cong Li, Fan Yang, and Guangyu Sun. Dimm-link: Enabling efficient inter-dimm communication for near-memory processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 302–316, 2023.

[267] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. Trim: Enhancing processor-memory interfaces with scalable tensor reduction in memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 268–281, New York, NY, USA, 2021. Association for Computing Machinery.

[268] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385, 2020.

[269] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. Toward standardized near-data processing with unrestricted data placement for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[270] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Neurostream: Scalable and energy efficient deep learning with smart memory cubes. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):420–434, 2018.

[271] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.

[272] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: Graph semantics aware ssd. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 116–128, 2019.

[273] Fangxin Liu, Wenbo Zhao, Yongbiao Chen, Zongwu Wang, Zhezhi He, Rui Yang, Qidong Tang, Tao Yang, Cheng Zhuo, and Li Jiang. Pim-dh: Reram-based processing-in-memory architecture for deep hashing acceleration. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 1087–1092, New York, NY, USA, 2022. Association for Computing Machinery.

[274] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. Enabling scientific computing on memristive accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 367–382, 2018.

[275] Shaahin Angizi and Deliang Fan. Graphide: A graph processing accelerator leveraging in-dram-computing. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.