Affinity Alloc: Taming ~~Not-So~~ Near-Data Computing

Zhengrong Wang
seanzw@ucla.edu
UCLA, USA

Christopher Liu
chrisliu@cs.ucla.edu
UCLA, USA

Nathan Beckmann
beckmann@cs.cmu.edu
CMU, USA

Tony Nowatzki
tjn@cs.ucla.edu
UCLA, USA

ABSTRACT

To mitigate the data movement bottleneck on large multicore systems, the near-data computing paradigm (NDC) offloads computation to where the data resides on-chip. The benefit of NDC heavily depends on spatial affinity, where all relevant data are in the same location, e.g. same cache bank. However, 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, 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.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; • **Software and its engineering → Allocation / deallocation strategies**.

KEYWORDS

Near-Data Computing, Data Layout, Data Placement, Data Structure Co-Design, Memory Allocation

ACM Reference Format:
Zhengrong Wang, Christopher Liu, Nathan Beckmann, and Tony Nowatzki. 2023. Affinity Alloc: Taming ~~Not-So~~ Near-Data Computing. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3623778

1 INTRODUCTION

As systems scale aggressively in the number of cores and memory channels, data movement has become increasingly the bottleneck for the von Neumann architecture. To mitigate this, architects proposed various near-data computing (NDC) techniques to offload computation to the memory hierarchy, e.g. last-level cache (LLC) [10, 37, 75, 93, 96], on-chip network router [81], memory [5, 6, 41, 43, 44, 56, 58, 60, 89], storage [57, 103], or multiple levels [36, 62]. By not bringing the data all the way to the core, near-data computing can achieve an integer multiple of performance and energy efficiency improvement, and is the key to continue efficient scaling for future systems.

However, 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 1 demonstrates, with Fig 1(a) depicting a conventional system. Fig 1(b) shows an NDC vector addition, where arrays not aligned in memory cause extra communication to collect operands. Fig 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 [5, 20, 30, 44, 81], or requires domain-specific preprocessing (e.g. graph partitioning) [40, 41]. Other work simply is oblivious to the data layout, and falls back to the conventional computing paradigm when NDC is not profitable [6, 43, 62, 75, 88, 96]. 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.

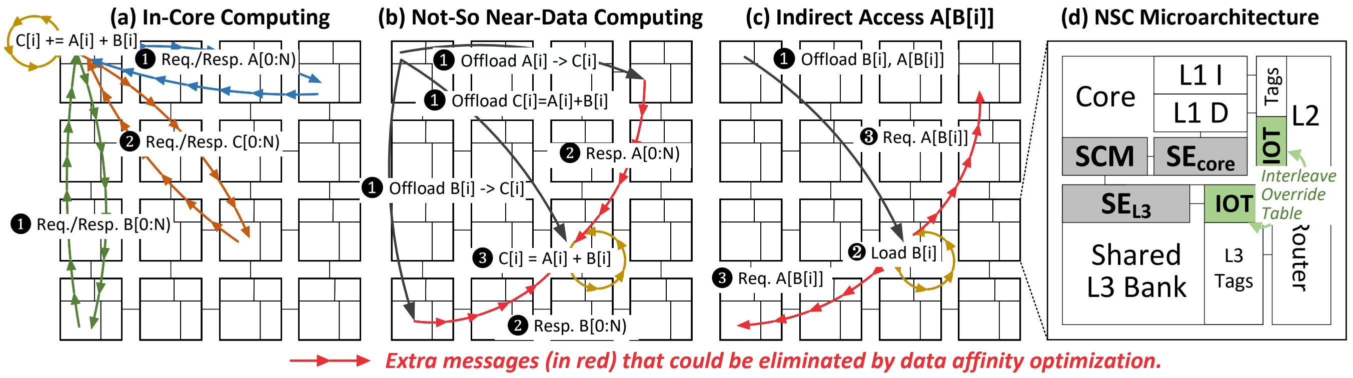


Figure 1: Affinity Optimization Opportunities in Near-Data Computing (View in Color)

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 [66], 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.

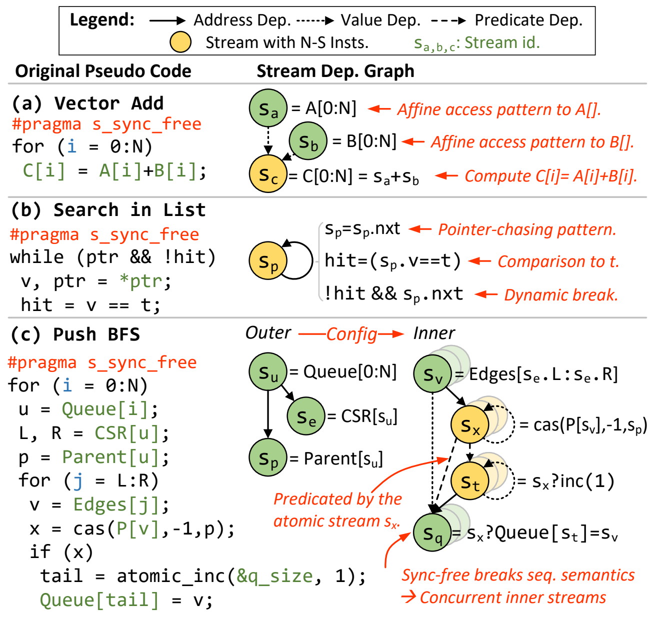
Our Approach: To summarize, we name our approach *affinity alloc*, as it systematically captures and optimizes data affinity for near-data computing. It contains a carefully designed allocator interface to capture the affinity information, a runtime library to lower the alignment constraints to an efficient 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.

In this work, 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 scale in modern CPUs (768MB on AMD EPYC 7773X [1]), and many algorithms can be tiled for locality in the LLC. However, because affinity alloc addresses the fundamental data placement problem, the principles and its implementation can be generalized to other near-data computing levels and techniques, e.g. near memory controller, in HMC die, near storage, etc.

Contributions: Evaluated on parallel workloads with a cycle-level simulator, affinity alloc achieves 2.26× speedup and 1.76× energy efficiency for a state-of-the-art near-data computing technique with 72% NoC traffic reduction, and 7.53× speedup and 4.69× energy efficiency over a wide OOO CPU. We also show that it is critical to codesign the data structure to optimize for data affinity in near-data computing. Specifically, our main contributions are:

- A general allocation interface to capture data alignment constraints for efficient near-data computing.
- A full-system implementation of affinity alloc, with a lightweight runtime library and μarch extension.
- Software co-optimizations that leverage the new interface to fully realize the potential of near-data computing.
- Detailed evaluation of how the new interface helps optimize the data layout and improves near-data computing.

Paper Organization: §2 introduces the baseline NDC. §3 discusses the data layout challenges and overviews our approach. §4 covers the basic interface and extensions to support affine layout, while §5 extends to irregular data layout. Methodology and evaluation are in §6 and §7. Further discussion and related work is in §8 and §9.


Legend: → Address Dep. ⇢ Value Dep. - - → Predicate Dep.
● Stream with N-S Insts.
$s_{a,b,c}$: Stream id.
Original Pseudo Code
Stream Dep. Graph
s_a = A[0:N] ← Affine access pattern to A[].
s_b = B[0:N] ← Affine access pattern to B[].
s_c = C[0:N] = s_a+s_b ← Compute C[i]= A[i]+B[i].
s_p=s_p.nxt ← Pointer-chasing pattern.
hit=(s_p.v==t) ← Comparison to t.
!hit && s_p.nxt ← Dynamic break.
Outer —Config→ Inner
s_u = Queue[0:N]
s_e = CSR[s_u]
s_p = Parent[s_u]
s_v = Edges[s_e.L:s_e.R]
s_x = cas(P[s_v],-1,s_p)
s_t = s_x?inc(1)
s_q = s_x?Queue[s_t]=s_v
Predicated by the atomic stream s_x.
Sync-free breaks seq. semantics → Concurrent inner streams

```
(a) Vector Add
#pragma s_sync_free
for (i = 0:N)
 C[i] = A[i]+B[i];
```

```
(b) Search in List
#pragma s_sync_free
while (ptr && !hit)
 v, ptr = *ptr;
 hit = v == t;
```

```
(c) Push BFS
#pragma s_sync_free
for (i = 0:N)
 u = Queue[i];
 L, R = CSR[u];
 p = Parent[u];
 for (j = L:R)
  v = Edges[j];
  x = cas(P[v],-1,p);
  if (x)
   tail = atomic_inc(&q_size, 1);
   Queue[tail] = v;
```

Figure 2: Example Near-Stream Computing Programs

2 BACKGROUND ON NEAR-DATA BASELINE

In this work, we leverage near-stream computing (NSC) [96] as the state-of-the-art baseline near-data computing framework[i]. Here we give background on this framework and point out opportunities for affinity-aware allocation.

2.1 Basic Near-Stream Computing

Stream Definition: NSC leverages "streams" as the basic unit for near-data computing, which has been widely adopted in general purpose computing [82, 95, 97] and reconfigurable accelerators [28, 61, 71, 98, 99]. Streams are defined by the long-term access pattern to the data structure, e.g. affine pattern `A[i]`, indirect `A[B[i]]` or pointer-chasing, and may contain NDC instructions. They are independently scheduled either at the core or near data at L3 banks.

Affine Stream: Fig 1(a) shows a multicore system with vector add `C[i]=A[i]+B[i]`. Tiles are connected by a mesh network and contain a core, private L1/L2 and a shared L3 cache bank. One major overhead here is the unnecessary traffic to fetch and write back the arrays, which have no reuse at all. Such overhead is only going to be more severe as the system scales up and the data grows.

To mitigate such overhead, in Fig 2(a), the near-stream computing compiler recognizes that there are three affine streams: two load streams s_a=`A[0:N]`, s_b=`B[0:N]`, and one store stream s_c=`C[0:N]`. It also extracts and associates the computation (i.e. addition) with the store stream s_c. This forms a stream dependence graph, in which edges represent the elementwise dependence between streams. In Fig 1(b), all streams are offloaded to the shared L3 banks where the data resides and automatically migrate to the next bank following the access pattern. Stream s_a and s_b directly forward their data to stream s_c. Stream s_c performs SIMD ops on a spare thread of the remote core and then writes directly to L3.

[i]§9 covers other related general near-data computing works and how they can benefit from affinity alloc.

An ideal data layout would colocate corresponding elements of the three arrays in the same bank to eliminate the data forwarding traffic (green and blue arrows in Fig1(b)), which is the goal of this work.

Pointer-Chasing Stream: Fig 2(b) shows a lined list traversal. The pointer-chasing stream s_p=s_p`.nxt` can be offloaded to compare against the target `t`. It also checks the loop condition based on the next pointer and comparison result. If evaluated to false, the stream is terminated, and the final value of `hit` is returned to the core.

An ideal allocator would place neighboring nodes in the same or close banks to reduce pointer-chasing distance.

Indirect Stream: Indirect accesses like `A[B[i]]` can also be offloaded. Fig 2(c) shows a push-based BFS kernel. The inner loop contains an indirect atomic access `P[Edges[i]]` to update the neighboring vertex's parent. Fig 1(c) shows the indirect stream offloaded along with the base stream: it reads the edge array `Edges[]`, generates indirect addresses and sends out indirect requests to target L3 banks. This eliminates the round-trip to the core for address generation.

An affinity-aware allocator would place `B[i]` *closer to the pointed* `A[B[i]]` *to reduce indirect traffic.*

2.2 Near-Stream Computing Details

For completeness, here we include other details of NSC that are not required to understand affinity alloc. Readers should feel free to skip this subsection.

Synchronization: The programmer annotates the loop with `#pragma s_sync_free`, indicating that there is no aliasing between core and stream, and no sequential semantics are needed. This enables the compiler to eliminate the original loop. Synchronization between the core and offloaded streams now relies on a coarsed-grained flow control scheme, i.e. one message contains credits for a few iterations. Similarly, context switch is possible by stopping issuing credits and waiting until all streams have reached the same point. Streams' progress is saved in the architectural state.

Predication: In Fig 2(c), if the vertex `v` has not been visited, i.e. compare and swap (CAS) operation succeeded, it is pushed into the queue for future processing. This push operation is broken into two streams: an atomic stream s_t to increment the tail pointer of the queue, and a store stream s_q to write `v` into the queue. Both streams are predicated by the atomic cas stream s_x, and will be skipped when s_x returns false.

Nested Stream: Finally, the inner loop streams in Fig 2c only take parameters from outer loop streams or some loop invariant values. Hence, the compiler can nest the inner loop streams into outer loop streams. Now, every iteration of the outer loop stream can configure an instance of the inner loop stream. Predication to skip the inner loop for certain outer loop iterations is supported. More importantly, this unlocks more parallelism by allowing multiple instances of inner loop streams to be executed concurrently, as no sequential semantics are required here for correctness.

Microarchitecture: Fig 1(d) shows the microarchitecture of near-stream computing, with the new components in gray. Stream engines are hardware units executing streams. The core stream engine (SE_{core}) decides whether to execute the stream in the core (when

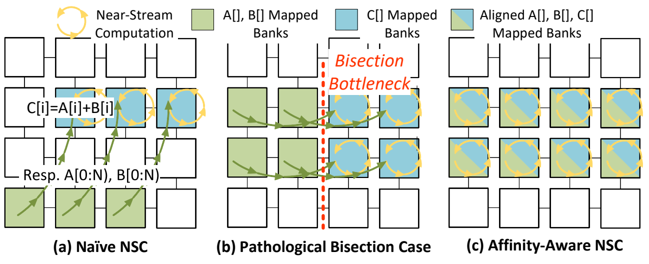


Figure 3: Affine Data Layout for Vec Add

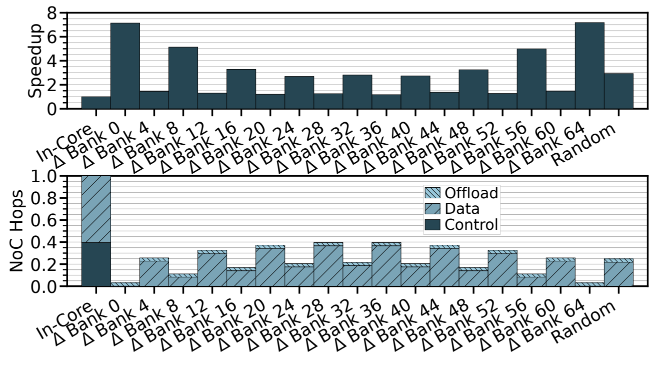


Figure 4: Impact of Affine Data Layout on Vec Add

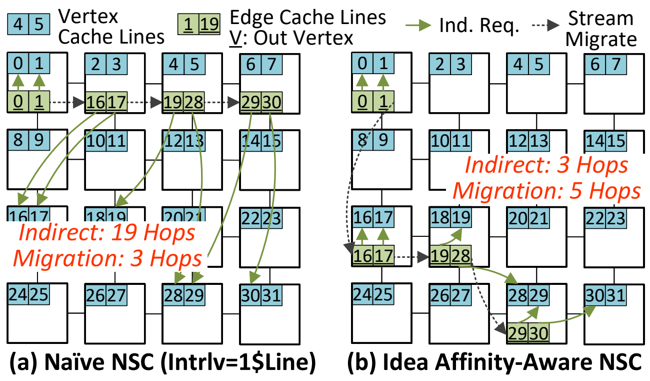


Figure 5: Irregular Data Layout for Graph Edge List

the stream is short or has high reuse in the private cache) or offload it to LLC. If offloading, it sends a configure packet to the L3 stream engine (SE_{L3}), which starts to access the L3 bank and perform NDC. To reduce overheads, synchronization between SE_{core} and SE_{L3} is coarse-grained, i.e. one message for multiple iterations.

Both SE_{core} and SE_{L3} contain ALUs to handle simple scalar operations, e.g. addition, multiplication, comparison, etc. More complex computations are outlined into a separate function and lowered into the native ISA by the compiler (x86 in this work). The stream configuration contains the function pointer, and the stream computing manager (SCM) assigns these functions to lightweight spare simultaneous multithreading (SMT) threads. Since there is no memory access nor control flow in near-stream computation, it can skip the LSQ and branch prediction. Near-stream computation can also be executed by special hardware, e.g. FPGAs [62], but is beyond the scope of this work. For context switch, offloaded streams are terminated with progress recorded in architectural states. When switching back, streams resume execution in SE_{core}.

3 MOTIVATION AND OVERVIEW

Here we first motivate affinity alloc by understanding the critical affine and irregular layout challenges in near-data computing. Then we overview how affinity alloc tackles these two challenges.

3.1 Affine Data Layout

We first consider a simple vector addition: `C[i]=A[i]+B[i]`. As shown in Fig 1(b) and Fig 3(a), When offloaded to the L3 cache, s_a and s_b forward the data to s_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 3(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 3(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 3(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 4 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 $(i + \Delta) \bmod 64$ (methodology in §6). 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.

3.2 Irregular Data Layout

The analogous data layout problems for irregular data structures are even more complicated to solve. Fig 5(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 5(b), if we can place the edges closer to the pointed-to vertices, we can significantly reduce the indirect access traffic to only 3 hops at the cost of a slightly longer migration distance.

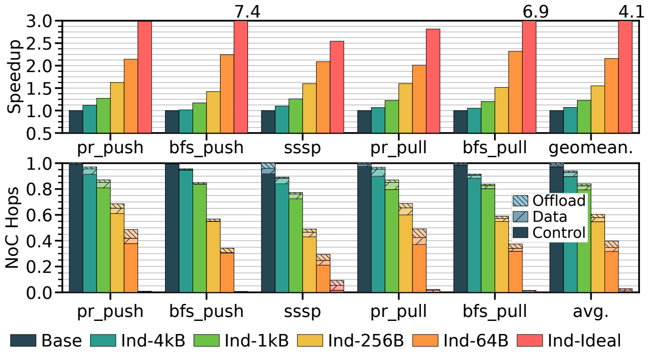


Figure 6: Impact of Irregular Data Layout

To quantify such benefit, Fig 6 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[ii]. 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.

3.3 Affinity Alloc Approach Overview

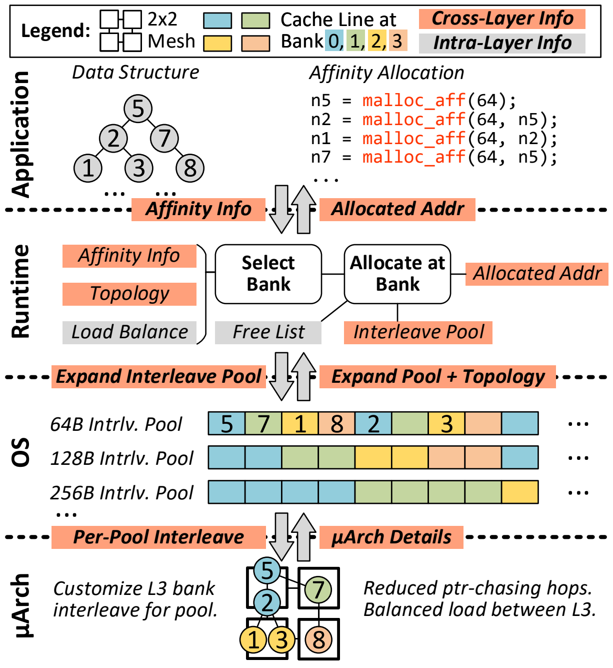
To exploit these opportunities, we propose *affinity alloc*, a systematic data placement solution that optimizes data affinity during allocation for near-data computing. Fig 7 overviews the approach across different system levels.

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 7, 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:

- **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.

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



```
n5 = malloc_aff(64);
n2 = malloc_aff(64, n5);
n1 = malloc_aff(64, n2);
n7 = malloc_aff(64, n5);
...
```

Figure 7: Affinity Alloc Approach Overview

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 7, 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 7). 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 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 [2, 26, 47, 48, 72], 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 §5), affinity alloc provides a general and systematic solution to make NDC computations *truly near data*.

4 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.

4.1 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 [32] 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 7, 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:

$$\text{bank}(vaddr) = \lfloor \frac{vaddr - start}{intrlv} \rfloor \pmod{N_{bank}} \tag{1}$$

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[iii].

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 [11] or RMM [54] supports continuous virtual to physical mapping). To complete the picture, the microarchitecture is extended with an interleave override table (IOT, Table 1) at each L2 and L3 cache controller.

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

Table 1: Interleave Override Table (IOT)

Each entry overrides the interleave for physical addresses within $[start, end)$. The L2/L3 cache controller as well as the $\text{SE}_{\text{core}}/\text{SE}_{\text{L3}}$ 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[iv]. 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. 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 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 numbered or enhancing Eq 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.

4.2 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 8(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 8(a)).

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

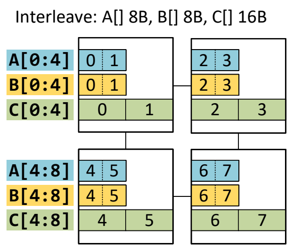
[iv]Physical pages for these interleavings are not continuous and are tracked as 4kB interleaving in the IOT.

(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 **Optimized Layout (8B $Line)**

```
// 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 **Optimized Layout (8B $Line)**

```
// 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});
```

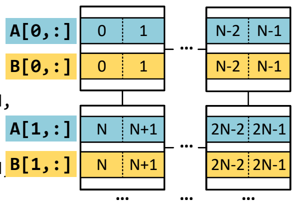


Figure 8: Affine Data Layout Optimizations

Inter-Array Affine Affinity: Fig 8(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 8(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:

$$B[i] \xrightarrow{aligns\ to} A[\frac{align\_p}{align\_q} \times i + align\_x] \quad (2)$$

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:

$$intrlv_B = \frac{elem\_size_B}{elem\_size_A} \times \frac{align\_q}{align\_p} \times intrlv_A \quad (3)$$

By factoring in the ratio of element sizes, the runtime chooses a 16B interleaving for the array double C[N] in Fig 8(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.

Partition Vertexes with Spatial Queue **Optimized Layout**

```
// 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;
```

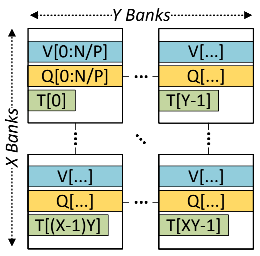


Figure 9: Distribute Partitions (Assume $P = X \times Y$)

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 8(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 8(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[v]. The runtime picks a valid interleaving that minimizes the Manhattan distance between A[i] and A[i+N]. For example, in Fig 8(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.

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 9 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 the push-based BFS in Fig 2(c), 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 9 we allocate a spatially distributed queue, with one sub-queue per partition. The tail pointer and data storage of

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

```
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;
```

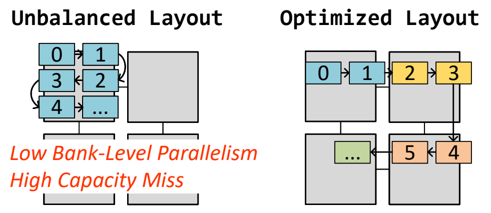


Figure 10: Irregular Data Layout API

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 B (i.e. P≠B), but having them equal yields better load balancing and higher performance. Priority queues, e.g. MultiQueues [79], 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.

5 IRREGULAR DATA LAYOUT

5.1 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 10 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 §5.2), 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.

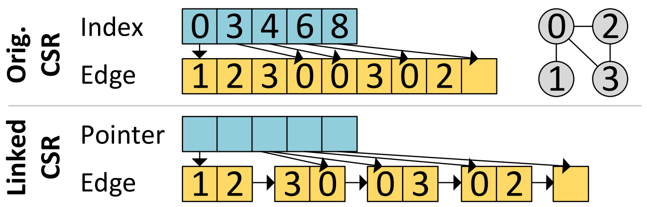


Figure 11: Linked CSR Format

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 microarchitecture only need to handle coarse-grained interleave pools and physical address ranges.

5.2 Bank Select Policy

Simply optimizing for data affinity may result in pathological unbalanced layout. For example, in the bottom left of Fig 10, 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:

$$score = avg\_hops + H \times (\frac{load}{avg\_load} - 1) \quad (4)$$

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 AB-NDP [89] to optimize task scheduling, while here we extend it for data allocation. We evaluate the sensitivity to *H* in §7.

5.3 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 in page 5). In this work, we focus on codesigning graph representations to optimize data affinity.

Fig 11 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

System	2.0GHz, 8x8 Cores
OOO8 CPU (8-issue)	64 IQ, 72 LQ, 56 SQ+SB 348 Int/FP RF, 224 ROB
Func. Units	8 Int ALU/SIMD (1 cy.) 4 Int Mult/Div (3/12 cy.) 4 FP ALU/SIMD (4 cy.) 4 FP Div (12 cy.)
L1 D/I TLB L2/SE$_{L3}$ TLB	64-entry, 8-way 2k/1k-entry, 16-way, 8 cy.
L1 I/D \$ Priv. L2 \$	32KB, 8-way, 2 cy. 256KB, 16-way, 16 cy.
Replacement L1 Bingo Pf. L2 Stride Pf.	Bimodal RRIP, $p = 0.03$ 8kB PHT, 2kB region 16 streams, 16 pf./stream
NoC	8x8 mesh topology 32B 1 cy. bidirection link 5-stage router, multicast X-Y routing, 4 mem. ctrls
Shared L3 \$	20 cycles, MESI Static NUCA, 1kB interleave 16-way, 64 banks, 1MB/bank total 64MB
DRAM	3200MHz DDR4 25.6 GB/s 4 channels at corners
SE$_{core}$	2kB FIFO, 12 streams
SE$_{L3}$	768 streams, 64kB buf. 4 cy. compute init. lat.
IOT	16 regions

Table 2: System and μarch Parameters (cy.: cycle)

Benchmark	Layout	Parameters
pathfinder [22]	Affine	1.5M entries, 8 iters
srad [22]	Affine	1k×2k, 8 iters
hotspot [22]	Affine	2k×1k, 8 iters
hotspot3D [22]	Affine	256×1k×8, 8 iters
bfs [13]	Linked CSR	Kronecker generated
pr_push [13]	Linked CSR	128k nodes 4M edges
sssp [13]	Linked CSR	A/B/C: 0.57/0.19/0.19
pr_pull [13]	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 3: Workloads Parameters

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 11), 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 5 (page 4). 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). Such co-optimization is the key to unlocking the full potential of near-data computing and can be applied to other domains and near-data computing systems.

6 METHODOLOGY

Compiler and Runtime: We extend the open-source LLVM-based near-stream computing compiler [96] to support predication on streams and dynamic loop bounds (§2). Programs are compiled to x86 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+ [63] 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 to expand interleave pools in gem5. We leverage McPAT [59] to estimate the energy and area with the 22nm process.

Parameters and Configurations: Table 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 [8], but no computation is offloaded (labelled as **In-Core** in §7). For near-memory computing, **Near-L3** offloads streams and the associated computation to SE$_{L3}$, 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 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 [13], 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 [12]. We discuss the tradeoffs between pushing/pulling and the heuristics we used in §7.

7 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.

7.1 General Evaluation

Overall Performance: Fig 12 shows the overall performance for all benchmarks. The speedup and energy efficiency are normalized

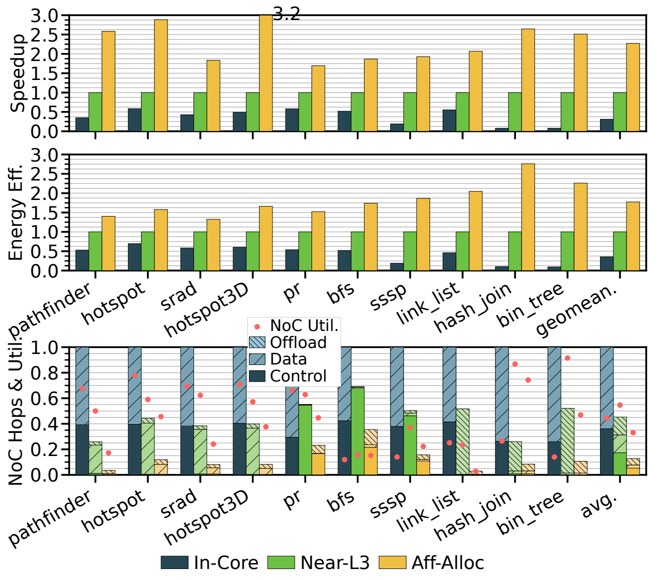


Figure 12: Overall Performance and Traffic Reduction

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 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 [9], 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 13 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 $H = 0$ in Eq 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

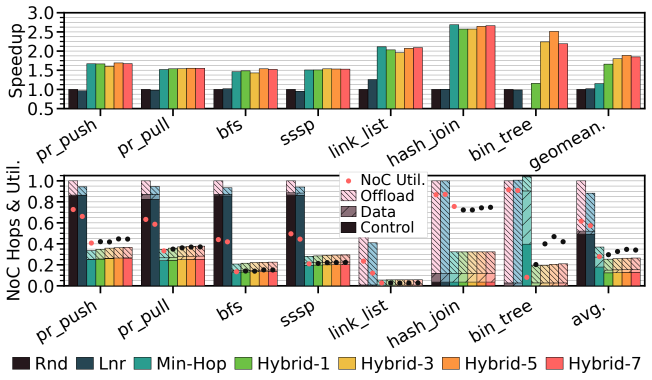


Figure 13: Sensitivity on Irregular Layout Policies

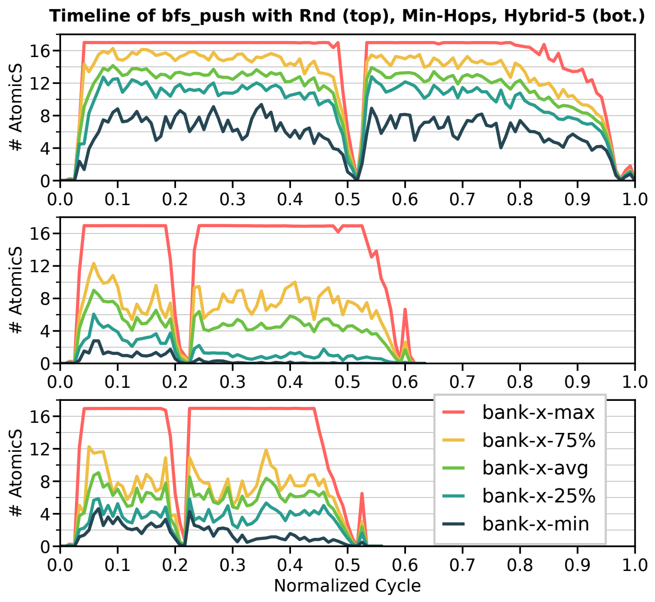


Figure 14: Distribution of Atomic Stream in BFS-Push

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 13), it dramatically increases the miss rate to that L3 bank and results in a huge slowdown.

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 14 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.

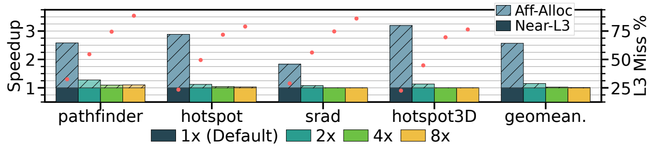


Figure 15: Speedup of Affine Layout on Large Inputs

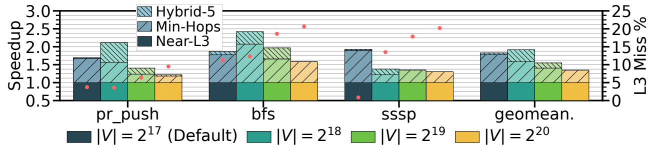


Figure 16: Speedup of Linked CSR on Large Graphs

Large Input Size: Fig 15 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 16 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 [1]), 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.

7.2 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.

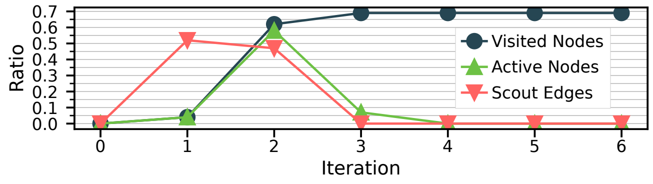


Figure 17: BFS Iteration Characteristic

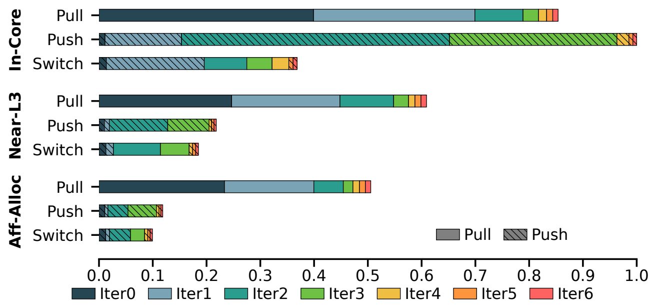


Figure 18: BFS Push vs. Pull Timeline

However, in `bfs`, each iteration has different characteristics and may benefit from per-iteration choices between pushing and pulling [12]. Fig 17 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 18 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 18) 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 default `bfs` implementation in GAP suite [13] 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 18, 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 17. 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* $\longrightarrow$ *Pull*: Visited Node > 40% and Scout Edge > 6%.
- *Pull* $\longrightarrow$ *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 partitioning divides

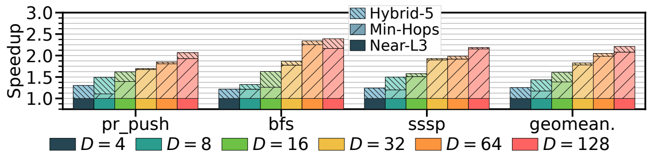


Figure 19: Speedup vs. Avg. Node Degree

Input Graph	Type	|Vertex|	|Edge|	Avg. Degree
twitch-gamers [80]	Power Law	168,114	13,595,114	81
gplus [64]	Power Law	107,614	13,673,453	127

Table 4: Real World Graphs

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 19 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. Table 4 lists the detailed information. These power-law graphs have a high average degree and are hard to partition. Fig 20 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.

8 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 [3, 33, 45, 50, 85] 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 [46, 74], 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.

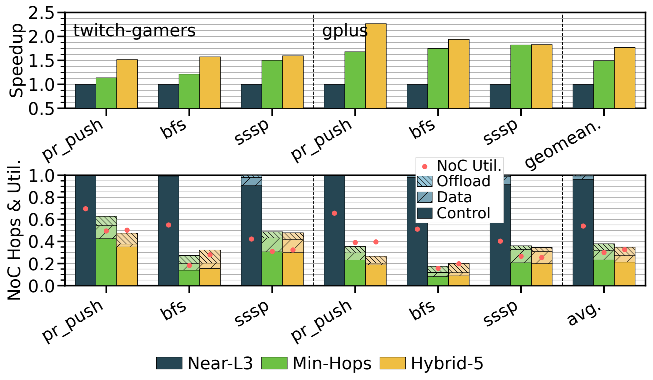


Figure 20: Performance on Real World Graphs

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 [54] has 32 range entries vs. 7 interleave pools in this work).

9 RELATED WORK

Multicore Caching and Dynamic Data Layout: Multicore caches are physically distributed, giving rise to non-uniform cache access (NUCA) [55]. Many dynamic NUCA (D-NUCA) designs have been proposed that change the data layout to reduce data movement[7, 14–18, 21, 23–25, 31, 39, 51, 65, 83, 90, 91]. 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[7, 16, 21, 25, 39, 51]. 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[7, 16–18, 35, 83, 84, 90, 91]. 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). Hotpad [92] designs a scratchpad hierarchy for managed languages (e.g. Java), but does not optimize for data affinity among banks.

Whirlpool [66] 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 [86], near LLC [10, 75, 96, 97], on-chip network router [81], memory [4–6, 20, 29, 30, 34, 40–44, 56, 58, 68, 89] and storage [57, 77, 103], and even across multiple levels or substrates [19, 36, 52, 62, 93, 94]. 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 [2, 48, 49, 76, 87] that use task hints to map tasks near-data [47, 100]. Several prior multicore accelerators [5, 72, 73] and reconfigurable architectures [26, 27, 69, 70] 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. [6, 29, 43, 52, 62, 75, 88, 96]. Other techniques require manual data placement using imperative APIs, e.g. [5, 20, 30, 34, 44, 81]. Hong et al. [40] organize the linked list nodes into the same HMC vault, and Gearbox [58] performs hybrid partition on SpMV and SpMSpV. These techniques are limited to a specific domain or affine workloads. Another line of work [52, 88] 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 [53] 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 [2, 5, 26, 38, 67, 72, 73, 78, 101, 102]. One use case is for vertex-centric graph processing, in which vertex-updates are scheduled near vertex properties storage [2, 5, 26, 38, 72, 73]. Our results suggest that our codesigned linked CSR format plus affinity alloc would be effective for them.

10 CONCLUSION

This work 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.

ACKNOWLEDGMENTS

This work was supported by funding from NSF grant CCF-2200831. We sincerely thank our anonymous shepherd and reviewers for their insights and suggestions.

REFERENCES

[1] 2023. AMD EPYC 7773X. https://www.amd.com/en/products/cpu/amd-epyc-7773x

[2] Maleen Abeydeera and Daniel Sanchez. 2020. 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).*

[3] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/3575693.3575713

[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 336–348.

[5] Kapil Arya, Yury Baskakov, and Alex Garthwaite. 2014. Tesseract: Reconciling Guest I/O and Hypervisor Swapping in a VM. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Salt Lake City, Utah, USA) *(VEE '14)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2576195.2576198

[6] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *HPCA*.

[7] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. 2009. 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*. 250–261. https://doi.org/10.1109/HPCA.2009.4798260

[8] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 399–411. https://doi.org/10.1109/HPCA.2019.00053

[9] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 14.

[10] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. 2022. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1160–1177. https://doi.org/10.1109/MICRO56248.2022.00083

[11] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 237–248. https://doi.org/10.1145/2508148.2485943

[12] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 12, 10 pages.

[13] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[14] B.M. Beckmann and D.A. Wood. 2004. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. 319–330. https://doi.org/10.1109/MICRO.2004.21

[15] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. 2006. ASR: Adaptive Selective Replication for CMP Caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 443–454. https://doi.org/10.1109/MICRO.2006.10

[16] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable Software-defined Caches. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) *(PACT '13)*. IEEE Press, Piscataway, NJ, USA, 213–224. http://dl.acm.org/citation.cfm?id=2523721.2523752

[17] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 538–550. https://doi.org/10.1109/HPCA.2015.7056061

[18] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 538–550.

[19] 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. 2021. 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* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 282–297. https://doi.org/10.1145/3466752.3480133

[20] 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. 2020. 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)*. https://doi.org/10.1109/MICRO50266.2020.00081

[21] Mainak Chaudhuri. 2009. 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*. 227–238. https://doi.org/10.1109/HPCA.2009.4798258

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[23] Z. Chishti, M.D. Powell, and T.N. Vijaykumar. 2003. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 55–66. https://doi.org/10.1109/MICRO.2003.1253183

[24] Z. Chishti, M.D. Powell, and T.N. Vijaykumar. 2005. Optimizing replication, communication, and capacity allocation in CMPs. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 357–368. https://doi.org/10.1109/ISCA.2005.39

[25] Sangyeun Cho and Lei Jin. 2006. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 455–468. https://doi.org/10.1109/MICRO.2006.31

[26] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[27] Vidushi Dadu and Tony Nowatzki. 2022. 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3503222.3507706

[28] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[29] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or Not for Emerging General Purpose Processing in DDR Memory Systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 231–244. https://doi.org/10.1145/3470496.3527431

[30] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. 2017. The mondrian data engine. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3079856.3080233

[31] Haakon Dybdahl and Per Stenstrom. 2007. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2–12. https://doi.org/10.1109/HPCA.2007.346180

[32] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3302424.3303977

[33] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527. https://doi.org/10.1145/3448016.3457263

[34] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. MeNDA: A near-Memory Multi-Way Merge Solution for Sparse Transposition and Dataflows. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 245–258. https://doi.org/10.1145/3470496.3527432

[35] Yaosheng Fu, Tri M Nguyen, and David Wentzlaff. 2015. Coherence domain restriction on large scale systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 686–698.

[36] Daichi Fujiki, Alireza Khadem, Scott Mahlke, and Reetuparna Das. 2022. Multi-Layer In-Memory Processing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 920–936. https://doi.org/10.1109/MICRO56248.2022.00068

[37] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2019. Duality Cache for Data Parallel Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 397–410. https://doi.org/10.1145/3307650.3322257

[38] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[39] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 184–195. https://doi.org/10.1145/1555815.1555779

[40] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating Linked-List Traversal Through Near-Data Processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2967938.2967958

[41] Kevin Hsieh, Eiman Ebrahim, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. 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.

[42] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 25–32. https://doi.org/10.1109/ICCD.2016.7753257

[43] Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum, and Eun Jung Kim. 2019. Active-Routing: Compute on the Way for Near-Data Processing. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 674–686.

[44] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. 2020. DUAL: Acceleration of Clustering Algorithms using Digital-based Processing In-Memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 356–371. https://doi.org/10.1109/MICRO50266.2020.00039

[45] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 337–355. https://www.usenix.org/conference/nsdi21/presentation/iyer

[46] Wole Jaiyeoba and Kevin Skadron. 2019. GraphTinker: A High Performance Data Structure for Dynamic Graph Processing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1030–1041. https://doi.org/10.1109/IPDPS.2019.00110

[47] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-Centric Execution of Speculative Parallel Programs. In

Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49).

[48] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2015. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 228–241. https://doi.org/10.1145/2830772.2830777

[49] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez. 2018. Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 217–230. https://doi.org/10.1109/MICRO.2018.00026

[50] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21).* Association for Computing Machinery, New York, NY, USA, 17–32. https://doi.org/10.1145/3447786.3456226

[51] Lei Jin and Sangyeun Cho. 2009. SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques.* 361–371. https://doi.org/10.1109/PACT.2009.14

[52] Mahmut Taylan Kandemir, Jihyun Ryoo, Xulong Tang, and Mustafa Karakoy. 2021. Compiler Support for near Data Computing. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21).* Association for Computing Machinery, New York, NY, USA, 90–104. https://doi.org/10.1145/3437801.3441600

[53] Mahmut Taylan Kandemir, Xulong Tang, Hui Zhao, Jihyun Ryoo, and Mustafa Karakoy. 2021. Distance-in-Time versus Distance-in-Space. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021).* Association for Computing Machinery, New York, NY, USA, 665–680. https://doi.org/10.1145/3453483.3454069

[54] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15).* Association for Computing Machinery, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[55] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated on-Chip Caches. *SIGARCH Comput. Archit. News* 30, 5 (oct 2002), 211–222. https://doi.org/10.1145/635506.605420

[56] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016).

[57] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17).* Association for Computing Machinery, New York, NY, USA, 219–231. https://doi.org/10.1145/3123939.3124553

[58] Marzieh Lenjani, Alif Ahmed, Mircea Stan, and Kevin Skadron. 2022. 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* (New York, New York) *(ISCA '22).* Association for Computing Machinery, New York, NY, USA, 218–230. https://doi.org/10.1145/3470496.3527402

[59] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. [n. d.]. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO '09.*

[60] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao. 2018. Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* https://doi.org/10.1109/MICRO.2018.00059

[61] 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. 2022. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* 35–56. https://doi.org/10.1109/MICRO56248.2022.00018

[62] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20).* Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3373376.3378497

[63] 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. 2020. The gem5 Simulator: Version 20.0+. In *CoRR*, Vol. abs/2007.03152. https://arxiv.org/abs/2007.03152

[64] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12).* Curran Associates Inc., Red Hook, NY, USA, 539–547.

[65] Javier Merino, Valentin Puente, and Jose A. Gregorio. 2010. ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture. In *2010 16th International Symposium on High-Performance Computer Architecture (HPCA'10).* 1–10. https://doi.org/10.1109/HPCA.2010.5416641

[66] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. 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* (Atlanta, Georgia, USA) *(ASPLOS '16).* Association for Computing Machinery, New York, NY, USA, 113–127. https://doi.org/10.1145/2872362.2872363

[67] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA).* IEEE.

[68] 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. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (2015). https://doi.org/10.1147/JRD.2015.2409732

[69] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21).* Association for Computing Machinery, New York, NY, USA, 1064–1077. https://doi.org/10.1145/3466752.3480048

[70] Quan M. Nguyen and Daniel Sanchez. 2023. Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* 1262–1274. https://doi.org/10.1109/HPCA56546.2023.10071026

[71] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17).* ACM, New York, NY, USA, 416–429. https://doi.org/10.1145/3079856.3080255

[72] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi. 2023. Dalorex: A Data-Local Program Execution and Architecture for Memory-bound Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE Computer Society, Los Alamitos, CA, USA, 718–730. https://doi.org/10.1109/HPCA56546.2023.10071089

[73] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. 2023. Massive Data-Centric Parallelism in the Chiplet Era. arXiv:2304.09389 [cs.DC]

[74] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21).* Association for Computing Machinery, New York, NY, USA, 1372–1385. https://doi.org/10.1145/3448016.3457313

[75] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in gpu architectures. In *Proceedings of the 46th International Symposium on Computer Architecture.* 210–223.

[76] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. 2022. A Scalable Architecture for Reprioritizing Ordered Parallelism. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New

York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 437–453. https://doi.org/10.1145/3470496.3527387

[77] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-Accelerated Near-Storage Sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 106–114. https://doi.org/10.1109/FCCM51124.2021.00020

[78] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 908–921. https://doi.org/10.1109/MICRO50266.2020.00078

[79] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 80–82. https://doi.org/10.1145/2755573.2755616

[80] Benedek Rozemberczki and Rik Sarkar. 2021. Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings. arXiv:2101.03091 [cs.SI]

[81] Karthik Sangaiah, Michael Lui, Ragh Kuttappa, Baris Taskin, and Mark Hempstead. [n. d.]. SnackNoC: Processing in the Communication Layer. ([n. d.]).

[82] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2019. Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores. *arXiv preprint arXiv:1911.08356* (2019).

[83] Brian C. Schwedock and Nathan Beckmann. 2020. Jumanji: The Case for Dynamic NUCA in the Datacenter. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 665–680. https://doi.org/10.1109/MICRO50266.2020.00061

[84] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. 2022. X-Cache: A Modular Architecture for Domain-Specific Caches. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 396–409. https://doi.org/10.1145/3470496.3527380

[85] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/2882903.2882950

[86] Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. 2015. The Execution Migration Machine: Directoryless Shared-Memory Architecture. *Computer* 48, 9 (sep 2015), 50–59. https://doi.org/10.1109/MC.2015.263

[87] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

[88] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 730–744. https://doi.org/10.1145/3123939.3123954

[89] Boyu Tian, Qihang Chen, and Mingyu Gao. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 3–17. https://doi.org/10.1145/3582016.3582026

[90] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 652–665. https://doi.org/10.1145/3079856.3080214

[91] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Nexus: A New Approach to Replication in Distributed Shared Caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 166–179. https://doi.org/10.1109/PACT.2017.42

[92] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking the Memory Hierarchy for Modern Languages. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 203–216. https://doi.org/10.1109/MICRO.2018.00025

[93] Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 359–375. https://doi.org/10.1145/3582016.3582032

[94] Zhengrong Wang, Christopher Liu, and Tony Nowatzki. 2022. Infinity Stream: Enabling Transparent and Automated In-Memory Computing. *IEEE Computer Architecture Letters* 21, 2 (2022), 85–88. https://doi.org/10.1109/LCA.2022.3203064

[95] Zhengrong Wang and Tony Nowatzki. 2019. Stream-Based Memory Access Specialization for General Purpose Processors. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, USA, 736–749. https://doi.org/10.1145/3307650.3322229

[96] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-Stream Computing: General and Transparent Near-Cache Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 331–345. https://doi.org/10.1109/HPCA53966.2022.00032

[97] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. 2021. Stream Floating: Enabling Proactive and Decentralized Cache Optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 640–653. https://doi.org/10.1109/HPCA51647.2021.00060

[98] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[99] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 703–716. https://doi.org/10.1109/HPCA47549.2020.00063

[100] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. 2020. T4: Compiling sequential code for effective speculative parallelization in hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 159–172.

[101] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2018.00053

[102] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.*

[103] Chen Zou and Andrew A. Chien. 2022. ASSASIN: Architecture Support for Stream Computing to Accelerate Computational Storage. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 354–368. https://doi.org/10.1109/MICRO56248.2022.00035