CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

Amirali Boroumand[†] Saugata Ghose[†] Minesh Patel[⋆] Hasan Hassan[⋆]
Brandon Lucia[†] Rachata Ausavarungnirun[†‡] Kevin Hsieh[†]
Nastaran Hajinazar[⋄†] Krishna T. Malladi[§] Hongzhong Zheng[§] Onur Mutlu[⋆†]

[†]Carnegie Mellon University [⋆]ETH Zürich [‡]KMUTNB
[⋄]Simon Fraser University [§]Samsung Semiconductor, Inc.

ABSTRACT

Specialized on-chip accelerators are widely used to improve the energy efficiency of computing systems. Recent advances in memory technology have enabled *near-data accelerators* (NDAs), which reside *off-chip* close to main memory and can yield further benefits than on-chip accelerators. However, enforcing coherence with the rest of the system, which is already a major challenge for accelerators, becomes more difficult for NDAs. This is because (1) the cost of communication between NDAs and CPUs is high, and (2) NDA applications generate a lot of off-chip data movement. As a result, as we show in this work, existing coherence mechanisms eliminate most of the benefits of NDAs. We extensively analyze these mechanisms, and observe that (1) the majority of off-chip coherence traffic is unnecessary, and (2) much of the off-chip traffic can be eliminated if a coherence mechanism has insight into the memory accesses performed by the NDA.

Based on our observations, we propose CoNDA, a coherence mechanism that lets an NDA *optimistically* execute an NDA kernel, under the assumption that the NDA has all necessary coherence permissions. This optimistic execution allows CoNDA to gather information on the memory accesses performed by the NDA and by the rest of the system. CoNDA exploits this information to avoid performing *unnecessary* coherence requests, and thus, significantly reduces data movement for coherence.

We evaluate CoNDA using state-of-the-art graph processing and hybrid in-memory database workloads. Averaged across all of our workloads operating on modest data set sizes, CoNDA improves performance by 19.6% over the highest-performance prior coherence mechanism (66.0%/51.7% over a CPU-only/NDA-only system) and reduces memory system energy consumption by 18.0% over the most energy-efficient prior coherence mechanism (43.7% over CPU-only). CoNDA comes within 10.4% and 4.4% of the performance and energy of an ideal mechanism with no cost for coherence. The benefits of CoNDA increase with large data sets, as CoNDA improves performance over the highest-performance prior coherence mechanism by 38.3% (8.4x/7.7x over CPU-only/NDA-only), and comes within 10.2% of an *ideal* no-cost coherence mechanism.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322266

1 INTRODUCTION

Modern systems increasingly employ specialized accelerators. Accelerators offer a way to improve system performance and energy efficiency in the face of diminishing returns from process technology scaling [22, 106]. Many recent works (e.g., [24, 31, 34, 37, 41, 42, 54, 67, 75, 77, 78, 125, 131]) propose *on-chip* customized hardware accelerators. These accelerators aim to satisfy the demands of emerging workloads.

Recent advances in 3D-stacked memory technology enable the practical implementation of *near-data processing*, where computational logic is placed close to memory. In particular, *near-data accelerators* (NDAs) can further boost the performance and energy benefits that conventional accelerators promise, by reducing the amount of *data movement* between the processor and the main memory [2–4, 13, 21, 25–27, 35, 47, 48, 60, 76, 88, 103–105, 113, 119, 127, 129, 130].

Despite the significant benefits of accelerators, system challenges remain a main stumbling block to the mainstream adoption of specialized accelerators. The lack of an efficient communication mechanism between CPUs and accelerators creates a significant overhead to synchronize data updates between the two [38, 89, 96, 107, 108, 118, 123]. This inefficiency generates unnecessary data movement, which can negate the benefits of accelerators. In addition, programmers are often required to use custom-designed communication mechanisms between CPU cores and accelerators, which leads to high programming complexity. These challenges can be largely mitigated by making the accelerator *coherent* with the rest of the system, as by doing so: (1) developers can use the conventional shared memory programming model, allowing them to use well-known synchronization mechanisms to coordinate between the accelerators and the CPUs; and (2) accelerators can efficiently share data with each other and with the rest of the system, instead of relying on bulk data transfers [70, 107, 108].

In contrast to coherence for *on-chip* accelerators [70, 96, 118], coherence for NDAs, which reside off-chip, is significantly more challenging for two reasons. First, the energy and performance costs of *off-chip* communication between NDAs and CPUs are very high. For example, for the Hybrid Memory Cube [49], the off-chip serial links consume as much energy to move data as the DRAM array consumes to access the data [5, 52, 95]. In fact, the energy and performance costs of *off-chip* communication between NDAs and CPUs are orders of magnitude greater than the costs of on-chip communication [56]. Second, target applications for NDAs are fundamentally different from those for on-chip accelerators, as NDA applications typically have low computational demand, suffer from poor locality, and generate a large amount of off-chip data movement [3–5, 13, 23, 46, 48, 88, 130]. These applications incur a large number of coherence misses. Given these challenges, it is impractical for an NDA to utilize traditional coherence mechanisms (e.g., MESI [32, 92]), which would need to send off-chip messages for *every cache miss* to coherence management hardware (e.g., a coherence directory) that reside on-chip with the CPU. The high cost and frequency of these messages with a traditional mechanism would eliminate most, if not all, of the benefits of near-data acceleration (Section 3).

While some recent NDA proposals acknowledge the need for NDA–CPU coherence [4, 25, 26, 126], these works largely sidestep the issue by assuming that the NDA and CPU share only a limited amount of data. While this is true for some NDA applications [25, 27, 59, 97, 126], our application analysis indicates that *this is not the case* for many other important NDA applications. We comprehensively analyze two important classes of such applications: graph processing frameworks and hybrid in-memory databases. We make a *key observation* that not all portions of these applications benefit from being executed on the NDA, and the portions that remain on the CPU (called *CPU threads*) often *concurrently* access the same region of data (e.g., graphs, database data structures) as the portions executed on the NDA (called *NDA kernels*), leading to *significant data sharing*. To understand the characteristics of sharing, we delve further into the memory access patterns of the CPU threads and NDA kernels, and we make a *second key observation*: while CPU threads and NDA kernels share the same data regions, they typically *do not* collide concurrently on (i.e., simultaneously access) the same cache lines. Furthermore, in the rare case of collisions, the CPU threads only read those cache lines, meaning that they *rarely* update the same data that an NDA is actively working on.

Using the major insights we obtain from our application analysis, we perform a design space exploration and examine three existing approaches for coherence: non-cacheable regions [3, 21, 25, 27, 88, 97], coarse-grained coherence [25, 26, 74, 121, 126, 129], and fine-grained coherence [13]. We find that (1) all three approaches eliminate a significant portion of the potential benefits of NDAs, as they generate a large amount of off-chip coherence traffic, and in some cases prevent concurrent execution of CPUs and NDAs; and (2) the majority of off-chip data movement (i.e., coherence traffic) generated by these approaches is unnecessary, primarily because the approaches pessimistically assume that *every* memory access needs to acquire coherence permissions or that shared data *cannot* be cached. We find that much of this unnecessary off-chip coherence traffic can be eliminated if the coherence mechanism has insight into what part of the shared data is *actually* accessed by NDA kernels and CPU threads. Unfortunately, this insight is *not* available before execution for many workloads with irregular access patterns. For example, in many pointer chasing or graph processing workloads, the path taken during pointer or graph traversal is *not* known prior to execution [3, 48, 66].

Based on these observations, we find that an *optimistic* approach to coherence can address the challenges of NDA coherence. An optimistic execution model for NDA enables us to gain insight into the memory accesses before any coherence permissions are checked, and thus, enforce coherence with only the *necessary* data movement. To this end, we propose *Coherence for Near-Data Accelerators* (CoNDA), a mechanism that lets the NDA *optimistically* start execution assuming that it has coherence permissions, without issuing *any* coherence messages off-chip. CoNDA executes NDA kernels in portions to keep hardware overheads low (see Section 5.5). While optimistically executing a portion of an NDA kernel, CoNDA records the memory accesses inside the NDA to gain insight into what part of the shared data is *actually* accessed by the kernel. During optimistic execution, CoNDA does not commit any data updates. When CoNDA finishes optimistic execution for the NDA kernel portion, it exploits the recorded information to check which coherence operations are necessary, in order to avoid off-chip data movement for unnecessary coherence operations. If the NDA kernel portion does not need coherence operations for *any* of its data updates, CoNDA commits the data updates. If the NDA kernel portion actually requires any coherence operations, CoNDA invalidates the uncommitted data updates, performs the needed coherence operations, and re-executes the NDA kernel portion.

Our optimistic execution model is inspired by Optimistic Concurrency Control (OCC) [71], which was first proposed in the database community and later harnessed for various purposes (e.g., Transactional Memory [7, 40, 44, 45, 84, 109], enforcing sequential consistency [16], deterministic shared memory [18]). The optimistic execution model fits very well within the context of NDA coherence for three reasons. First, the optimistic approach makes it possible to identify the *necessary* coherence traffic, by gaining insight into the memory accesses performed by the NDA and by the rest of the system before generating off-chip coherence requests. Second, our application analysis shows that CPU threads *rarely* update the same data that an NDA is actively working on (which is also true for other applications that do not have a high degree of data sharing [25, 27, 59, 97, 126]). This behavior leads to a very low re-execution rate of NDA kernels, which is one of our key motivations behind adopting an optimistic execution model for NDA coherence. Third, NDAs are often relatively simple fixed-function accelerators or small programmable cores that lack sophisticated ILP techniques [3, 4, 13, 21, 26, 48, 76, 88, 97, 121, 126]. As a result, the cost of kernel re-execution on an NDA is much lower than that on a sophisticated out-of-order CPU core.

We find that CoNDA is highly effective in efficiently enforcing coherence between the CPU threads and NDAs. Compared to three major existing coherence mechanisms, CoNDA improves performance by 19.6% over the highest performance prior mechanism and reduces memory system energy by 18.0% over the most energy-efficient prior mechanism, on average across our 16-thread workloads operating on modest data set sizes. Over a CPU-only system with no NDAs, CoNDA improves performance by 66.0% and reduces memory system energy consumption by 43.7%. Over an NDA-only system, CoNDA improves performance by 51.8%. These benefits arise because CoNDA eliminates the majority of unnecessary coherence traffic, and is able to retain almost all of the benefits of near-data acceleration. CoNDA comes within 10.4% and 4.4% of the performance and energy, respectively, of an ideal NDA mechanism that incurs no penalty for coherence.

CoNDA is an efficient mechanism whose benefits increase as the application data set size increases. This is because a larger data set results in a higher cache miss rate, which generates more coherence traffic over the off-chip channel and, thus, provides greater optimization opportunities for CoNDA. When we increase the data set sizes by an order of magnitude, we find that CoNDA improves performance by 8.4x over CPU-only, 7.7x over NDA-only, and 38.3% over the best prior coherence mechanism, coming within 10.2% of ideal coherence.

We make the following key contributions in this work:

- We analyze two important classes of applications, and show that (1) there is a *significant* amount of data sharing between CPU threads and NDAs, (2) CPU threads and NDAs often do *not* access the same cache lines *concurrently*, and (3) CPU threads rarely update the same data that NDAs are actively working on.

- We perform an extensive design exploration and show that (1) the poor handling of coherence eliminates much of an NDA's performance and energy benefits, (2) the majority of off-chip data movement (i.e., coherence traffic) generated by a coherence mechanism is unnecessary, and (3) a significant portion of unnecessary coherence traffic can be eliminated by having insight into which memory accesses actually require a coherence operation.
- We propose CoNDA, a new coherence mechanism that *optimistically executes* code on an NDA to gather information on memory accesses. Optimistic execution enables CoNDA to identify and avoid performing *unnecessary* coherence requests. As our evaluation shows, this reduces off-chip data movement, allowing NDA execution under CoNDA to always outperform prior coherence mechanisms (as well as CPU-only and NDA-only execution).

2 BACKGROUND

2.1 Near-Data Processing

Since the early 1970s, many works examined various forms of near-data processing [20, 28, 55, 68, 79, 87, 91, 93, 110, 116]. With the advent of 3D-stacked memories, we have seen a resurgence of near-data processing proposals. Some recent proposals (e.g., [3–5, 13, 21, 26, 27, 35, 36, 47, 48, 59, 88, 94, 126, 127, 129, 130]) add compute units within the logic layer of 3D-stacked memory [49, 53, 64, 73]. These works primarily focus on the design of the underlying logic that is placed within memory, and in many cases propose special-purpose near-data accelerators that cater to a limited set of applications.

2.2 Baseline Architecture

Figure 1 shows the baseline organization of the architecture we assume in this work, which includes programmable or fixed-function near-data accelerators (NDAs). Each NDA executes an *NDA kernel* that is invoked by the CPU threads. In our evaluation, we implement programmable NDAs consisting of *in-order* cores that are ISA-compatible with the CPU cores, but that do not have large caches or any sophisticated ILP techniques. Each NDA includes small private L1 I/D caches. While our evaluations use a programmable NDA, CoNDA can be used with *any* programmable, fixed-function, or reconfigurable NDA.

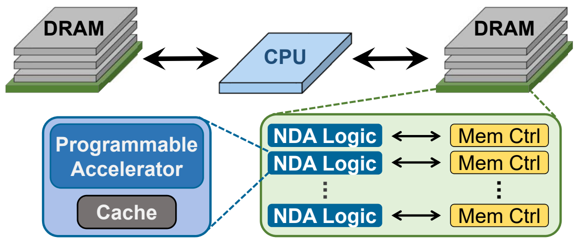

Figure 1: High-level organization of our NDA architecture.

3 MOTIVATION

Enabling coherence for near-data accelerators provides two key benefits: (1) programmers can use the well-known traditional shared memory model to program systems with near-data accelerators, and (2) we can simplify how accelerators communicate and share data with each other and with the rest of the system. In this section, we analyze the data sharing characteristics of several important data-intensive workloads (Section 3.1), and then study how existing coherence mechanisms perform for these workloads (Section 3.2).

3.1 Application Analysis

An application can benefit from near-data acceleration when its memory-intensive parts are offloaded to NDAs. The memory-intensive parts of an application generate a significant amount of data movement, and often exhibit poor temporal locality, leading to high execution times and energy consumption on a CPU. In the NDA, such application parts can benefit from the high-bandwidth, low latency and low-energy memory access available in 3D-stacked memory. However, the compute-intensive parts of the application should *remain on the CPU cores* to maximize performance [4, 13, 26, 47, 88, 104], especially if they can exploit the CPU cache hierarchy well or benefit from sophisticated ILP techniques.

Sharing Data Between NDAs and the CPU. Because many applications have compute-intensive parts that should be executed on the CPU, NDA kernels and CPU threads may share data with each other, depending on how an application is partitioned. While some applications [25, 27, 59, 97, 126] can be partitioned to limit this data sharing, we make a *key observation* that *this is not the case* for many important classes of applications.

A major example is multithreaded graph processing frameworks, such as Ligra [111], where multiple threads operate in parallel on the same shared in-memory graph [65, 111, 128]. Each thread executes a graph algorithm, such as *PageRank* [15]. We rigorously study a number of these algorithms [111] and find that when we carefully convert each of them for NDA execution, only some portions of each algorithm are well-suited for NDA, while the remaining portions perform better if they stay on the CPU. With this partitioning, the CPU portion of each thread executes on the CPU cores while the NDA portions (sometimes concurrently) execute on the near-data accelerators, with all of the threads sharing the graph and other intermediate data structures. For example, we find that for the *Radii* and *PageRank* algorithms running on the *arXiv* input graph, 60.1% and 71.0% of last-level cache (LLC) misses generated by the CPU threads access the shared data (e.g., the graph, intermediate data structures) after we convert these algorithms for NDA execution.

A second major example is modern in-memory databases. Today, analytical and transactional operations are combined into a single hybrid transactional/analytical processing (HTAP) database system [8, 9, 58, 72, 80, 82, 101]. The analytical queries of these hybrid databases are well-suited for NDA execution, as they have long execution times (i.e., the queries run for long enough to amortize the overhead of dispatching the query to the NDA) and touch a large number of rows, leading to a large amount of random memory accesses and data movement [67, 83, 126]. In contrast, even though transactional queries access the same data, they perform better if they stay on the CPU, as they have short execution times, are latency-sensitive, and have cache-friendly access patterns. In such workloads, concurrent accesses from both NDA kernels and CPU threads to shared data structures are inevitable.

Many NDA workloads exhibit the same characteristics that we find in these applications. In several workload domains, recent works show that only parts of an application benefit from NDA execution, while the rest of the application should stay on CPUs [4, 13, 26, 47, 88, 104]. These NDA kernels often share many data structures with the CPU threads. For example, in TensorFlow Mobile [33], prior work shows that two functions (*packing* and *quantization*) should be NDA kernels [13]. These kernels share the neural network data structures (e.g., matrices) with the CPU threads, which execute functions such as convolution and matrix multiplication.

Shared Data Access Patterns. To further understand data sharing, we analyze the memory access patterns of the CPU threads and NDA kernels. We make a *second key observation*: while CPU threads and NDA kernels share data structures, they often *do not concurrently access the same elements* (e.g., a shared graph node or database row) in these structures. For example, when we analyze the *Connected Components* and *Radii* algorithms from Ligra [111] using the *arXiv* input graph, only 5.1% and 7.6%, respectively, of the CPU accesses collide with (i.e., access the same cache line as) accesses from the NDA. Across all of our applications, only 11.2% of these collisions (less than 1% of all accesses) are writes from a CPU thread, and all other collisions are CPU thread reads.

The low rate of collisions with data updates (i.e., writes) is intuitive from a programming perspective, because when programmers offload some of the code to an accelerator, they try to avoid having the CPU thread update the data that the accelerator is working on. Such behavior is a common characteristic of many accelerator-centric applications [24, 50, 51, 54, 67, 77, 99].

3.2 Analysis of NDA Coherence Mechanisms

We explore three types of existing mechanisms that can be used to enforce coherence between CPU threads and NDAs.

Non-Cacheable Approach. One approach to sidestep coherence is to mark any data accessed by the NDA (i.e., the NDA data region) as *non-cacheable* by the CPU [3]. This ensures that any CPU writes are immediately visible to the NDA. While this works well for applications where the CPU *rarely* accesses the NDA data region, it performs poorly for many applications where the CPU accesses the region *often*. For Ligra [111] applications with a representative input graph (*arXiv*), we find that 38.6% of all accesses made by CPU threads to memory are to the NDA data region.

Figures 2 and 3 show the memory system energy consumption and speedup of different coherence mechanisms for a system with NDAs, normalized to a *CPU-only* baseline where the entire application runs on the CPU. In the figures, we also show a mechanism called *Ideal-NDA*, where there is no energy or performance penalty for coherence. We observe from the figures that the non-cacheable approach (*NC*) fails to provide any energy savings (and, in fact, greatly increases energy consumption), and on average performs 6.0% worse than CPU-only. Therefore, NC is a poor fit for applications where the NDA and CPU threads share data, as it is unable to realize any benefit from NDA in our workloads.

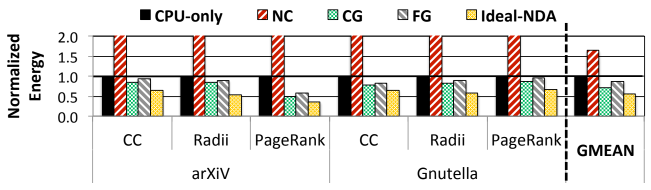

Figure 2: Memory system energy with existing coherence mechanisms.

Coarse-Grained Coherence. A second approach is *coarse-grained coherence*, where there is a *single* coherence permission that applies to the *entire* NDA data region. This works well when there is only a limited amount of data shared between the CPU threads and the NDA kernel, but incurs high overheads when there is a high amount of data sharing and there is no easy way to find what part of the data will be accessed by the NDA before execution (e.g., due to irregular access patterns). The CPU must flush *all* dirty cache lines in the NDA data region *every time* an NDA acquires coherence permissions for the region, *even if the NDA does not access most of the data in the region.* This results in a significant amount of *unnecessary* data movement. For example, with only four CPU threads, *PageRank* flushes 227x the number of cache lines actually accessed by the NDA. While coherence at a smaller granularity (e.g., one entry per page [26]) can reduce this overhead in some cases, the overhead remains high for the many applications that have irregular access patterns. For example, pointer chasing applications [3, 48, 66] access only a few cache lines in each page, but still require *all* changes to the page to be flushed.

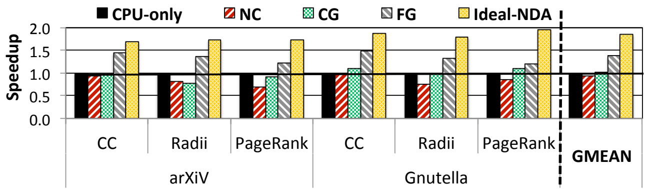

Figure 3: Speedup with existing coherence mechanisms.

Some instances of coarse-grained coherence use *coarse-grained locks* to provide the CPU or NDA with *exclusive* access to a coarse-grained region. Without exclusive access, data can ping-pong between the CPU and the NDA when they concurrently access the NDA data region, increasing data movement. Coarse-grained locks avoid ping-ponging by having the NDA acquire exclusive access to a region for the duration of the NDA kernel. Our analysis shows that coarse-grained locks greatly limit performance when *any* sharing exists, by forcing CPU threads and NDA kernels to serialize. Averaged across all Ligra applications running on a representative input graph (*Gnutella*), coarse-grained locks block 87.9% of the CPU's memory accesses during NDA execution. As Figures 2 and 3 show, the high impact of unnecessary flushes and serialization cause coarse-grained locks (*CG* in the figures) to eliminate a large portion of the energy and performance benefits that Ideal-NDA provides. In fact, *CG* performs 0.4% *worse* than CPU-only, on average.

We conclude that while CG works well in some cases, it is *not* suitable for many important NDA applications.

Fine-Grained Coherence. Traditional, or *fine-grained*, coherence protocols (e.g., MESI [32, 92]) have two major qualities well suited for applications with irregular memory accesses (e.g., graph workloads, databases, pointer chasing). First, fine-grained coherence acquires coherence permissions for only the pointers that are actually traversed during pointer chasing. The path taken during pointer traversal is not known ahead of time. As a result, even though a thread often accesses only a few dispersed pieces of the data structure, a coarse-grained mechanism has no choice but to acquire coherence permissions for the entire data structure. Fine-grained coherence allows the CPU or NDA to acquire permissions for only the pieces of data in the data structure that are actually accessed. Second, fine-grained coherence can ease programmer effort when developing applications for NDAs, as multithreaded programs already use this programming model.

Unfortunately, if an NDA participates in fine-grained coherence with the CPU, it has to exchange coherence messages for *every* *cache miss* with the CPU directory over a narrow pin-limited bus. Since NDA kernels are memory-intensive and exhibit poor temporal locality, this fine-grained message exchange generates large amounts of off-chip data movement. A large amount of those coherence messages are unnecessary, as our analysis from Section 3.1 shows that the vast majority of NDA accesses do not collide with CPU writes and, thus, do not need coherence.

We apply a number of optimizations to fine-grained coherence to reduce the amount of off-chip data movement. We enforce exclusive ownership of each cache line by either the NDA or the CPU. We add a local NDA directory in DRAM to maintain coherence between multiple NDAs. We add a bit to each entry in both the CPU and NDA directories, to mark the cache lines that are owned by an NDA. If a coherence miss occurs for a cache line owned by the NDA, this means that the CPU also does not have a copy of the cache line, and no request needs to be sent to the CPU. Unfortunately, even with these optimizations, fine-grained coherence using the MESI protocol [32, 92] (*FG*) eliminates a significant portion of the energy and performance benefits of Ideal-NDA, as shown in Figures 2 and 3. This is because the optimized FG (1) still needs to send off-chip coherence requests to acquire ownership; (2) is unable to avoid the majority of off-chip coherence requests, as the NDA kernel has a high number of cache misses that generate many coherence requests; and (3) still causes data to ping-pong between the CPU and the NDA, generating many off-chip coherence messages for a single cache line.

We conclude that while FG acquires permissions for only cache lines that are actually accessed, it still causes a lot of unnecessary off-chip data movement.

Limitations of Existing Coherence Mechanisms. The majority of unnecessary off-chip data movement generated by existing coherence mechanisms is due to a lack of *insight* into when coherence requests are actually necessary during NDA kernel execution. Without having any such insight, existing mechanisms preemptively issue coherence requests, many of which are *not needed*, causing significant off-chip data movement to maintain coherence. As a result, none of these mechanisms can exploit the potential energy and performance benefits of NDAs.

4 OPTIMISTIC NDA EXECUTION

As we see in Section 3, existing coherence mechanisms can eliminate the benefits of near-data acceleration for many important application classes. We find that the majority of off-chip data movement generated by these coherence mechanisms is unnecessary, primarily because the mechanisms pessimistically assume that (1) *every* memory operation needs to acquire coherence permissions or (2) data shared between CPUs and NDAs *cannot* be cached. Much of this unnecessary movement can be eliminated if a coherence mechanism has insight on *what part of the shared data* is *actually* accessed. Based on our observations from Section 3, we propose to use *optimistic execution* for NDAs. When executing in optimistic mode, an NDA gains insight into its memory accesses by tracking the accesses *without* issuing any coherence requests. When optimistic execution is done, the NDA uses the tracking information to perform *necessary* coherence requests for *only* the parts of the shared data that were *actually* accessed during execution, which minimizes coherence-related data movement.

In this section, we discuss our optimistic execution model and how it retains existing memory consistency guarantees (Section 4.1), and analyze when coherence messages are necessary in optimistic mode (Section 4.2). We assume a sequentially-consistent memory model in this section, but our execution model can easily be applied to other common memory consistency models, such as the x86-TSO model (Section 5.8).

In Section 5, we propose CoNDA, a coherence mechanism for NDAs that makes use of optimistic execution.

4.1 Execution Model

An application running on the CPU can issue a call to start executing code on an NDA. When the NDA starts executing, the CPU threads may execute concurrently. The NDA executes in optimistic mode, where it assumes that it *always* has coherence permissions on the cache lines that it uses, *without* checking the CPU coherence directory. This avoids the need for off-chip coherence communication during execution. Because the NDA has not actually checked coherence, it ensures that none of its data updates are committed to memory during optimistic execution.

When optimistic execution stops, the system must determine whether it can *commit* the data updates from optimistic execution. This requires the NDA to determine if any coherence requests should have been issued to guarantee correctness. To keep the commit mechanism simple, our execution model makes use of *coarse-grained atomicity*, where *all* memory updates by the NDA are treated as if they all occur *at the moment when optimistic execution stops*. Thus, for all memory operations that take place while the CPU threads and the NDA execute concurrently, the CPU thread memory operations are effectively ordered *before* the NDA memory operations, which is a valid ordering for sequential consistency. When the NDA attempts to commit, the system first checks to see if any coherence violation happened (see Section 4.2). If no violation occurred, the NDA data updates are committed. Otherwise, the NDA must resolve any violations by performing the necessary coherence operations, and re-execute the optimistically-executed code. Because CPU threads and NDA kernels rarely access the same cache lines during concurrent execution (see Section 3.1), re-execution happens rarely, as we show in Section 7.2, making optimistic execution efficient.

We do *not* expose the optimistic execution behavior of the NDA to programmers. Our execution model ensures that memory consistency is not violated, and thus, the programmer can treat the system like a conventional multithreaded system. To make use of near-data acceleration, a programmer simply needs to insert macros to demarcate portions of the application that should be executed on the NDA (see Section 5.1), and can treat the NDA kernel as just another thread. As is the case for concurrent CPU threads, the programmer assumes that (1) instructions across multiple threads can be interleaved *in any order* acceptable under the memory consistency model of the system, and (2) they need to use synchronization primitives if they want to enforce a *specific* ordering. Our mechanism supports synchronization primitives (see Section 5.7).

4.2 Identifying Necessary Coherence Requests

In order to maintain coherence, the NDA must perform any *necessary* coherence operations before committing its uncommitted memory operations. Coherence requests are necessary *only* when the NDA and/or a CPU thread update a cache line that both the NDA and the CPU thread access. Figure 4 shows an example of how CPU and NDA operations are ordered. We discuss three possible interleavings of CPU and NDA memory operations to the same cache

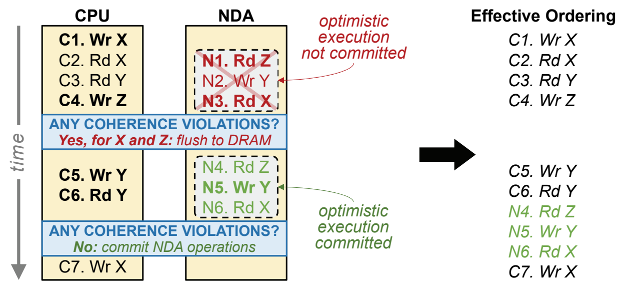

Figure 4: Example timeline of optimistic NDA execution.

line, and show that only one of the interleavings leads to a violation and requires NDA re-execution to ensure correct execution.

Case 1: NDA Read and CPU Write (to the Same Cache Line). *This case triggers re-execution.* As we discuss in Section 4.1, all NDA operations are ordered to take place *once optimistic execution stops.* As a result, any CPU memory accesses that take place *during* optimistic mode are ordered before any NDA operations. For example, the NDA read (e.g., N1 in Figure 4, which reads cache line Z) should be ordered after the CPU write to the same cache line (e.g., C4). However, since the NDA did not issue a coherence request before performing the read operation, it did not read the updated value that was written by the CPU. In order to maintain coherence, the value written by the CPU must be flushed to DRAM, and the NDA must re-execute to ensure correct ordering (e.g., N1–N3 are re-executed as N4–N6).

Note that the CPU may perform the write (e.g., C1 to cache line X) *before* optimistic execution begins. This can still require re-execution if the updated cache line is not written back to memory before the NDA performs the read (e.g., N3).

Case 2: NDA Write and CPU Read (to the Same Cache Line). *This case does not trigger re-execution.* In our execution model, any read to a cache line by the CPU during optimistic mode execution is ordered *before* the NDA's write to the same cache line. As a result, the CPU should *not* read the value written by the NDA. For example, NDA write N5 to cache line Y in Figure 4 is effectively ordered to take place after CPU read C6 to the same cache line. To ensure that the CPU does not read the value written by the NDA, the NDA maintains the updated value in a *data update buffer* until it can confirm that there is no need for it to re-execute instructions. Once this is confirmed, the NDA commits the write to memory, and invalidates any stale copies of the data in the CPU cache.

If the programmer wants to *guarantee* that the CPU read sees the NDA's write, the program must use synchronization primitives to enforce this ordering (see Section 5.7). This is true in conventional CPU multithreading as well.

Case 3: NDA Write and CPU Write (to the Same Cache Line). *This case does not trigger re-execution.* Similar to Case 1, the NDA write (e.g., N5) takes place after the CPU write (e.g., C5), and the NDA holds the update in the data update buffer. However, when the NDA is ready to commit its updates (i.e., there is no need to re-execute), it cannot simply flush the old cache line in the CPU. This is because the CPU and the NDA may have written to *different words* in the same cache line. To ensure that no updates are lost, the data update buffer must include a per-word dirty bit mask, as in [74]. When the system commits the NDA write, it first retrieves the latest version of the cache line from the CPU, and then overwrites only the words in the cache line that were written to by the NDA using the values of those words that are in the data update buffer. Again, if the programmer wishes to enforce a specific ordering of the writes, they must use a synchronization primitive (e.g., a write fence), as in conventional CPU multithreading.

5 CONDA ARCHITECTURE

In this section, we describe *Coherence for Near-Data Accelerators* (CoNDA), an efficient coherence mechanism that makes use of optimistic NDA execution (see Section 4) to avoid unnecessary off-chip coherence traffic. Figure 5 shows the high-level operation of CoNDA. In CoNDA, when an application wants to launch an NDA kernel (Section 5.1), the NDA begins executing the kernel in optimistic mode (❶ in Figure 5; Section 5.2). While the NDA kernel executes, all CPU threads continue to execute normally, and *never* make use of optimistic execution.

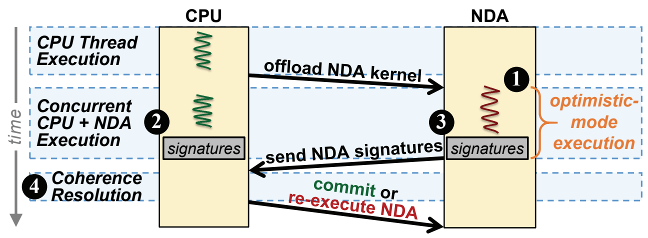

Figure 5: High-level operation of CoNDA.

To gain the insight needed to perform only the necessary coherence requests, CoNDA efficiently tracks the addresses of all NDA reads, NDA writes, and CPU writes during optimistic execution using *signatures* (❷ and ❸; Section 5.3). Once optimistic execution starts, any NDA data updates are initially flagged as uncommitted (Section 5.4). These updates cannot be committed until all necessary coherence requests are performed. When optimistic execution is done (Section 5.5), CoNDA attempts to resolve coherence (❹; Section 5.6). The NDA transmits its signatures to the CPU, and CoNDA compares the NDA signatures with the CPU signatures to identify the necessary coherence requests. If CoNDA detects any coherence violation (see Section 4.2), (1) the NDA invalidates all of its uncommitted updates; (2) the CPU resolves the coherence requests, performing only the necessary coherence operations (including any cache line updates); and (3) the NDA re-executes the uncommitted portion of the kernel. Otherwise, CoNDA performs the necessary coherence operations, clears the uncommitted flag for all data updates in the NDA L1 cache (i.e., any uncommitted data updates are committed), and resumes optimistic execution if the NDA kernel is not finished.

To preserve the conventional multithreaded programming interface, CoNDA correctly supports synchronization primitives (Section 5.7). CoNDA enables optimistic execution, ensures correctness, and reduces unnecessary coherence traffic compared to state-of-the-art coherence mechanisms, all with a low hardware overhead (Section 5.9).

5.1 Program Interface

We provide a simple interface for programmers to port applications to CoNDA. The programmer identifies the portion(s) of the code to execute on an NDA using two macros (`NDA_begin` and `NDA_end`). The compiler converts the macros into instructions that we add to the ISA, which *trigger* and *end* NDA kernel execution, respectively. To reduce tracking overhead, CoNDA needs to know which pages in memory *might* be accessed by an NDA, which we call the *NDA data region*. The NDA data region can be identified by a compiler, or can be manually annotated by the programmer. CoNDA stores this information by adding one bit to each page table entry, which when set indicates that CoNDA needs to track reads from and writes to this page during optimistic execution.

5.2 Starting Optimistic NDA Execution

CoNDA starts optimistic execution when an NDA kernel is launched, or after a successful coherence resolution operation. When an NDA kernel is first launched, the CPU dispatches the kernel's starting PC and any live-in registers to a free NDA. Any time optimistic execution starts, CoNDA takes a checkpoint of the current NDA state, which consists of the NDA's PC and software-visible registers. This checkpoint is used in case commit fails and the NDA needs to re-execute.

5.3 Signatures

CoNDA uses signatures to track whether any coherence requests are necessary during optimistic execution. CoNDA uses this information once optimistic execution ends to perform *only* the necessary coherence requests.

Implementation. To reduce the amount of storage needed to track coherence requests without missing any addresses that need to be checked for correct coherence, we implement signatures in CoNDA using fixed-length parallel Bloom filters [11]. In a parallel Bloom filter, an N-bit signature is partitioned into M segments. Each segment in the signature employs a unique hash function (H_3 [100]). When an address is added to the signature, each segment's hash function maps the address to a single bit in the segment, which we call the *hashed value*, and the bit is set to 1. Using a parallel Bloom filter, CoNDA can perform three operations. First, it can check if a filter contains a given memory address, by generating the hashed values for the address and checking if they are set in *all* M segments. Second, it can retrieve a list of all addresses in the filter, by using the signature expansion technique [16, 17]. Third, it can quickly compare two filters to see if both filters might contain one or more of the same addresses, by taking the bitwise AND of the two signatures to generate the filter intersection and checking that each segment in the intersection has at least one bit set (indicating at least one matching address that is shared by both filters).

Once a bit is set in a parallel Bloom filter segment, the bit remains set until the filter is reset. As a result, there are *no false negatives*. This ensures that CoNDA checks all of the addresses that are added to the signatures during optimistic execution. Due to aliasing of some hashed values, Bloom filters can introduce a limited number of false positives (see below), which may lead to unnecessary re-executions without affecting correctness. The parallel Bloom filters in CoNDA are reset every time optimistic execution starts.

Tracking Memory Operations During NDA Execution. CoNDA maintains three sets of signatures during optimistic execution, as shown in Figure 6. The `NDAReadSet` records the addresses of all cache lines read from by the NDA. The `NDAWriteSet` records the addresses of all cache lines written to by the NDA. The `CPUWriteSet` records the addresses of all cache lines *in the NDA data region* that (1) a CPU thread writes to during optimistic execution, or (2) have dirty copies in a CPU cache before an NDA kernel starts. The CPU scans the caches before launching an NDA kernel to find dirty cache lines that reside in the NDA data region. Our evaluation shows that across the entire program, the total time spent on scanning accounts for less than 1% of the overall execution time (Section 7), including TLB overheads. This is because the NDA kernels are very data intensive, and require orders of magnitude more time to execute than scanning the CPU L1 cache tag stores, which can be done in parallel across caches.

In the unlikely case that scanning becomes a performance bottleneck (which we never observe in our experiments), we can optimize the scan operation by introducing a Dirty-Block Index [102] to track dirty NDA data.

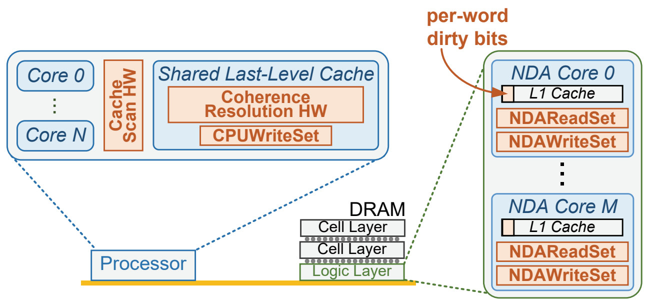

Figure 6: Hardware additions to support CoNDA.

Signature Size. Each Bloom filter is composed of a fixed-length register. The size of the register determines the maximum number of addresses can be added to the filter without exceeding a given false positive rate. Once the maximum number of addresses is reached in any signature, CoNDA stops optimistic execution and begins resolving necessary coherence requests for the NDA kernel. We size the signatures in CoNDA to balance the storage overhead, the false positive rate (as false positives can cause unnecessary re-executions), how frequently optimistic execution must be stopped, and the amount of data movement needed to transfer signatures from the NDA to the CPU when CoNDA performs any necessary coherence requests.

The `NDAReadSet` and `NDAWriteSet` each use one 256 B register that is split into four segments. The `CPUWriteSet` uses *eight* 256 B Bloom filters. This is because the `CPUWriteSet` does not need to be transmitted off-chip during coherence resolution, so we can use multiple filters to increase the addresses that the `CPUWriteSet` can hold. We use a round robin policy to select which of the filters an address is added to. We target a 20% false positive rate (worst case) in CoNDA, which allows us to hold up to 250 addresses in each 256 B filter. Our Bloom filter analysis is based on mathematical derivations in prior work [100].

5.4 Buffering Uncommitted Values

Each NDA includes a small private L1 data cache (see Section 2.2). During optimistic execution, when an NDA performs a write operation, the write value cannot be committed to memory until CoNDA can perform the necessary coherence operations. As a result, the NDA buffers the updated data value in the L1 cache, using techniques similar to prior works on speculative execution [7, 16, 39, 40, 44, 45, 69, 71, 84, 115]. We add a per-word dirty bit mask [74] to each cache line, as shown in Figure 6, to mark all uncommitted data updates.

5.5 Ending Optimistic NDA Execution

CoNDA dynamically determines when to end optimistic execution. Execution in optimistic mode stops when one of three events occurs:

(1) the NDA reaches the end of the NDA kernel, (2) the NDA L1 data cache needs to evict a speculative cache line, or (3) one of the signatures cannot hold more addresses without exceeding the false positive rate. To determine if a signature cannot hold more addresses, CoNDA maintains a counter for the number of addresses added to each signature since optimistic execution started.

5.6 Identifying Necessary Coherence Requests

Once optimistic execution ends, CoNDA uses the signatures to check for and resolve any necessary coherence operations, and attempts to commit any uncommitted NDA data updates. The `NDAReadSet` and `NDAWriteSet` are sent from the NDA to *coherence resolution* logic residing in the CPU. The coherence resolution logic then calculates the intersection of the `NDAReadSet` and the `CPUWriteSet`. There are two cases, depending on whether the intersection contains a match.

Case 1 (Conflict). If the intersection contains a match, a coherence violation *may* have occurred (i.e., an NDA read and a CPU write may have been to the same address; see Section 4.2). This means that the pending NDA data updates cannot be committed, as they *may* violate correct memory ordering. Instead, CoNDA must perform the necessary coherence operations and then re-execute the NDA operations. To do so, the CPU flushes any dirty cache lines that match addresses in the `NDAReadSet` to DRAM, and places a copy of these cache lines in the NDA L1 cache. The CPU uses the signature expansion method [16, 17] to decode the `CPUWriteSet` into corresponding addresses, and then checks if those addresses are present within the `NDAReadSet`. Once the dirty cache line flush completes, the coherence resolution logic sends a message to the NDA that the commit attempt failed. The NDA invalidates all uncommitted data in its L1 cache, rolls back to its checkpointed state, and restarts optimistic execution. If the same portion of the NDA kernel fails commit *N* times (we empirically set *N* to three), CoNDA guarantees forward progress of the NDA by acquiring a lock for each cache line in the `NDAReadSet`(e.g., by temporarily setting the page table read-only bit for any page that contains a cache line in the `NDAReadSet`). This ensures that the kernel does not roll back anymore, and avoids livelock.

Case 2 (No Conflict). If no match is found in the intersection of the `NDAReadSet` and the `CPUWriteSet`, then no memory ordering violation exists, and the commit process starts to perform any necessary coherence operations. First, the coherence resolution hardware computes the intersection of the `NDAWriteSet` and the `CPUWriteSet`. If an overlap is found, CoNDA uses signature expansion (see Section 5.3) to identify which cache lines need to be merged (see Section 4.2), and sends these cache lines to the NDA for merging, which ensures write-after-write coherence. Second, all cache lines in the CPU cache that match an address in the `NDAWriteSet` are invalidated. Third, a message is sent to the NDA, which clears the uncommitted flag on all NDA cache lines and allows the lines to be written to DRAM. Finally, if the NDA kernel is not finished, the NDA continues executing the kernel by starting optimistic execution from the instruction after the commit.

During coherence resolution, CPU threads continue to execute, but all coherence directory entries for cache lines in the NDA data region are locked to ensure atomicity. If a thread accesses any cache line in the NDA region, the thread stalls until the coherence resolution completes.

5.7 Support for Synchronization Primitives

CoNDA allows a programmer to use traditional synchronization primitives to manually order memory operations and provide atomicity. When an NDA reaches a synchronization primitive (e.g., `Acquire`, `Release`), it performs three steps. First, CoNDA ends optimistic execution, and commits any pending speculative updates before the primitive. This ensures that there are no remaining memory operations that *must* be ordered before the synchronization primitive, preserving the memory ordering expected by the programmer. Second, the NDA executes the primitive *non-speculatively*, performing any coherence operations necessary for the primitive. By executing the primitive non-speculatively, CoNDA guarantees that the primitive *cannot be rolled back*, allowing the primitive to work exactly as it does in conventional CPU multithreading. Third, after the primitive has been performed, the NDA resumes optimistic execution.

5.8 Support for Weaker Consistency Models

In Section 4.1, we assume a sequentially-consistent memory model, but CoNDA can support other common memory consistency models. We illustrate this using a brief case study on how CoNDA works with the x86-TSO (total store ordering) consistency model. CoNDA can support x86-TSO by waiting to add store addresses to signatures until the addresses are issued to the memory system. To support x86-TSO, each CPU and NDA needs to include a FIFO write buffer that can hold issued stores until they are written to the memory system. Since the write buffer is architecturally invisible, CoNDA ensures that the `CPUWriteSet` signatures do not record an address that is in the write buffer. CoNDA records a write address in the `CPUWriteSet` only when the address leaves the write buffer and is sent to the memory system. The coherence resolution logic in CoNDA remains unchanged, as the write must become visible to the entire system when it is completed by the memory system. For example, if the CPU writes to memory address A, and an NDA reads from memory address A, there is no conflict as long as the write stays in the write buffer, and the NDA read can complete, which is an expected memory ordering in x86-TSO. Once the write from the write buffer is issued to the memory, the address is recorded in the `CPUWriteSet` and becomes visible to the entire system.

5.9 Hardware Overhead

Each NDA uses 512 B to store signatures, while the CPU uses 2 kB in total. Aside from the signatures, CoNDA's overhead consists mainly of (1) 1 bit per page in the page table (0.003% of DRAM capacity) and 1 bit per TLB entry for the page table flag bits (Section 5.1); (2) a 1.6% increase in NDA L1 data cache size for the per-word uncommitted data bit mask (Section 5.4); and (3) two 8-bit counters per NDA to track the number of addresses stored in each signature.

6 METHODOLOGY

We implement CoNDA in the gem5 simulator [10]. We perform all simulations in full-system mode using the x86 ISA, and modify the integrated DRAMSim2 [19] DRAM timing simulator to model 3D-stacked HMC DRAM [29, 30, 49] available to the NDAs. To accurately model the coherence mechanisms, we modify the Ruby memory model in gem5. We include a local coherence directory for the NDAs, and set the CPU coherence directory as the main point of coherence for the system. Both the NDA and CPU directories use the MESI protocol. Table 1 shows our system configuration.

Processor	16 cores, 8-wide issue, 2 GHz frequency, out-of-order *L1 I/D Caches*: 64 kB private, 4-way, 64 B blocks *L2 Cache* 4 MB shared, 8-way, 64 B blocks *Coherence*: MESI
Near-Data Accelerator	16 NDAs per stack, 1-wide issue, 2 GHz frequency, in-order *L1 I/D Caches*: 64 kB private, 4-way, 64 B blocks *Coherence*: MESI
HMC [49]	one 4 GB cube, 16 vaults per cube, 16 banks per vault
Memory	DDR3-1600, 4 GB, FR-FCFS scheduler

Table 1: Evaluated system configuration.

Our simulation platform models all of the overheads of CoNDA. During coherence resolution, these overheads include (1) 20 cycles to send each signature from the NDA to the CPU, which is a conservative estimate given that it takes 3 cycles to transfer a 256 B signature across the HMC link; (2) 2 cycles to compare a CPU signature to an NDA signature, which is more conservative than prior work [16, 17]; (3) 8 cycles to invalidate a CPU cache line on a matching address; and (4) 12 cycles to transfer a cache line from the CPU to the NDA to merge writes. We model the full overhead of NDA kernel re-execution, which involves (1) invalidating uncommitted cache lines and erasing signatures; (2) rolling back the NDA to a checkpoint, (3) resolving coherence, and (4) re-running the kernel. We assume that NDA rollback takes 8 cycles, since the NDA is a small core with a small cache, making the rollback significantly cheaper than for large out-of-order CPUs.

We report *memory system energy* using an energy model similar to prior work [13], which accounts for the total energy consumed by the DRAM, on-chip and off-chip interconnects, and all caches. We use detailed simulator statistics to drive this model. We model the 3D-stacked DRAM energy as the energy consumed per bit, leveraging estimates and models from prior work [52]. We estimate the energy consumption of all L1 and L2 caches using CACTI-P 6.5 [86], assuming a 22 nm process. We model the off-chip interconnect using the method used by prior work [26], which estimates the HMC SerDes energy consumption as 3 pJ/bit for data packets.

Applications. We study two classes of applications that are well-suited for NDA. For these applications, we use two types of input datasets: (1) a modest size dataset, which we use to perform the majority of our evaluations; and (2) a large size dataset, which we use in Section 7.5 to show how CoNDA's benefits scale as the input sizes increase.

We evaluate three graph applications from Ligra [111] (a lightweight multithreaded graph framework): *Connected Components* (*CC*), *Radii*, and *PageRank* (*PR*). The modest size dataset for our graph applications consists of input graphs from three real-world networks [114]: *Enron* email communication network (73384 nodes, 367662 edges), *arXiv* General Relativity (10484 nodes, 28984 edges), and peer-to-peer *Gnutella25* (45374 nodes, 109410 edges). The large size dataset consists of input graphs from two real-world networks [114], which are on average 14.8x larger than the graphs used in the modest size dataset: (1) the *Amazon* product network (334863 nodes, 925872 edges), and (2) the *DBLP* collaboration network (317080 nodes, 1049866 edges)).

We also evaluate an in-house prototype of an in-memory database (IMDB) that supports HTAP workloads [82, 101, 117]. Our IMDB uses a state-of-the-art, highly-optimized hash join kernel [120]. For the modest size dataset, we simulate an IMDB system with 64 tables, 64K tuples per table, and 32 randomly-populated integer fields per table. Our transactional workload consists of 64K transactions, where each transaction reads from or writes to 1–3 randomly-chosen database tuples. Our two analytical workloads consist of 128 or 256 analytical queries (*HTAP-128* and *HTAP-256*) that use the *select* and *join* operations. For the large size dataset, we increase the size of the IMDB system to 64 tables with 640K tuples per table, with 256K transactions in the transactional workload and 1024 queries in the analytical workload (*HTAP-1024*).

Identifying NDA Kernels in Applications. Similar to prior works [4, 13], we identify candidate NDA kernels that are memory-intensive and not cache friendly. Using hardware performance counter data, we consider a function to be an NDA kernel candidate when (1) it is memory intensive (i.e., its LLC misses per kilo instruction, or MPKI, is greater than 20 [62, 63, 85]); (2) the majority of the function's execution time is spent on data movement; and (3) it is one of the top three functions in the workload in terms of execution time. From this set of candidate kernels, we select kernels for NDA execution such that we minimize the amount of data sharing that occurs between CPU threads and NDA kernels. A lower amount of data sharing leads to higher performance under existing coherence mechanisms (i.e., NC, CG, FG), and more modest performance improvements with CoNDA. We manually annotate the NDA data region, by replacing all `malloc` calls to data in the region with a custom memory allocator called `nda_alloc`, which notifies gem5 that the data belongs to the region.

In most Ligra applications, we select the *edgeMap* function as an NDA kernel. This function processes and updates a subset of edges for each vertex [111], which generates many random memory accesses. In our *HTAP* workloads, we select the analytical queries (i.e., *select* and *join* operations) as NDA kernels.

7 EVALUATION

We show results normalized to a *CPU-only* baseline, and compare CoNDA to NDA execution using fine-grained coherence (*FG*), coarse-grained locks (*CG*), non-cacheable NDA data (*NC*), or ideal coherence (*Ideal-NDA*), as described in Section 3.2.

7.1 Off-Chip Data Movement

Figure 7 shows the normalized off-chip data movement (which we measure as bytes transferred between the NDAs and the CPU) of the NDA coherence mechanisms for a system with 16 CPU cores and 16 NDAs. We make three observations from the figure. First, CoNDA significantly reduces the *overall* data movement compared to all prior NDA coherence mechanisms, with an average reduction of 30.9% over the next best mechanism, CG. Compared to CG, which has to flush every dirty cache line in each region acquired by the NDA, CoNDA uses its insight on NDA memory accesses to greatly reduce the number of lines flushed (e.g., by 92.2% for Radii using arXiv). Second, NC's data movement is very high because *all* processor accesses to the NDA data region must go to DRAM. Third, CoNDA reduces data movement by 86.3% over CPU-only, because it successfully allows memory-intensive portions of the applications to no longer consume off-chip bandwidth.

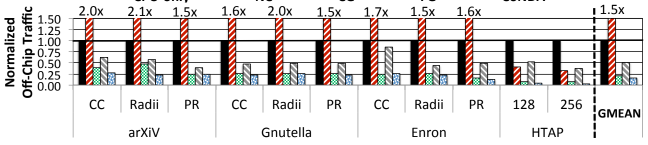

Figure 7: Normalized off-chip data movement.

7.2 Performance

Figure 8 shows the performance improvement for our 16-core system. We make four observations from the figure. First, with no coherence overhead, Ideal-NDA shows that there is significant potential for speedup (on average 1.84x) in our applications when we use NDAs. Second, we observe that CG and NC experience drastic performance losses compared to Ideal-NDA, due to the high costs they have in maintaining coherence. Both CG and NC lead to on average 0.4% and 6.0% performance loss over CPU-only. Third, while FG provides reasonable performance improvements, it still falls far short of Ideal-NDA, achieving only 44.9% of Ideal-NDA's performance benefits. Fourth, unlike the other mechanisms, CoNDA's efficient approach to coherence allows it to retain most of the performance benefits of Ideal-NDA, coming within 10.4% on average. CoNDA improves performance over CPU-only by 66.0%, and over the best previous mechanism, FG, by 19.6%.

We also evaluate the effect of offloading the entire application to NDAs. Running an entire application on NDAs eliminates the need for coherence between the CPU and NDAs. However, our analysis shows that executing these applications entirely on NDAs hurts performance significantly, eliminating on average 82.2% of Ideal-NDA's performance improvement. CoNDA performs 51.7% better than NDA-only, and NDA-only performs only 8.7% better than CPU-only. The reason is that the NDA cannot afford to incorporate complicated logic and large caches due to power and area constraints [13, 21, 27], which, for NDA-only, significantly slows down the parts of the application that are better suited for CPU execution.

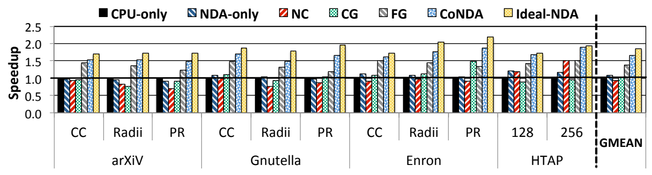

Figure 8: Speedup.

CoNDA's execution time consists of three major parts: (1) NDA kernel execution, (2) coherence resolution overhead, and (3) re-execution overhead. Our analysis shows that for our applications, the coherence resolution and re-execution overheads take 3.3% and 8.4% of the entire execution time, respectively.

Coherence Resolution Overhead. We find that the coherence resolution overhead is low because (1) CPU threads do *not* stall during resolution unless they access the NDA data region; (2) the `NDAWriteSet` typically contains only a small number of addresses (6 on average), which limits the number of CPU-side invalidations and NDA writebacks (see Section 5.6); and (3) resolution mainly involves sending signatures and checking for any necessary coherence operations, which altogether take less than 50 cycles.

Re-Execution Overhead. We find that the overhead of re-execution is small for two reasons. First, as we mention in Section 3.1, the collision rate (i.e., the fraction of commit attempts that require re-execution) is low (13.4% on average) for our applications, limiting the number of times an NDA must re-execute and the number of cache lines that must be flushed to DRAM. Second, the re-execution of the NDA kernel portion is significantly *faster* than its original execution. This is because the majority of data and instructions needed by the NDA during re-execution are already in the NDA L1 cache from the previous execution, and CoNDA places a copy of each cache line flushed from the CPU during coherence resolution into the NDA L1 cache.

7.3 Memory System Energy

Figure 9 shows the memory system energy consumption of the NDA coherence mechanisms. We observe that CoNDA reduces energy consumption over all prior mechanisms, by 18.0% on average over the best mechanism for memory system energy (CG). CoNDA achieves nearly all of the energy reduction potential of Ideal-NDA, coming within 4.4%, and reduces energy consumption by 43.7% over CPU-only. This is because CoNDA successfully reduces off-chip traffic, and eliminates much of the unnecessary coherence traffic of existing coherence mechanisms.

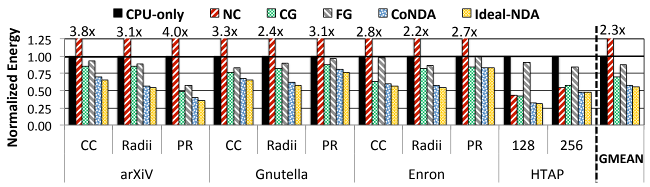

Figure 9: Normalized memory system energy.

FG, CG, and NC each introduce coherence overheads that undermine their potential for energy savings. FG exchanges a very high number of off-chip coherence messages between the CPU and the NDAs. While CG reduces memory system energy over CPU-only, it induces a large number of writebacks, which increase DRAM energy consumption by 18.9% CPU-only. The additional DRAM energy cancels out some of the off-chip interconnect energy savings (a 49.1% reduction over CPU-only) that CG provides. We find that CG is more energy efficient than FG because it generates less off-chip coherence traffic. CG performs writebacks only at the beginning of NDA kernel execution while FG generates off-chip coherence traffic on every single coherence miss. NC forces all CPU accesses to the NDA data region to bypass the CPU caches and go to DRAM, which increases the interconnect and DRAM energy over CPU-only by 3.1x and 4.5x, respectively.

7.4 Multiple Memory Stacks

In systems that need a large main memory, there can be multiple memory stacks each with NDAs. We evaluate how CoNDA's benefits scale as multiple stacks require coherence, using a representative workload (*PageRank* with the *arXiv* input graph) as a case study. In this study, we assume that there are four CPU cores and four NDAs per stack, and that stacks are connected together using the *processor-centric topology* [61]. We use the local NDA directory in each NDA stack (Section 6) to maintain coherence between different stacks (i.e., similar to the distributed directory in NUMA machines [1, 43]) using the MESI coherence protocol.

Off-Chip Traffic. Figure 10 shows the normalized off-chip traffic for each mechanism as the number of stacks increases. The NDA and CPU core count increases with stack count, and so does the off-chip memory traffic. Unlike existing NDA coherence mechanisms, off-chip traffic under CoNDA scales well and remains significantly lower than CPU-only as we increase the stack count. FG and CG do *not* scale as well as CoNDA. For FG, increasing the stack count leads to a significantly larger number of coherence misses, and thus, generates more off-chip coherence messages. For CG, the number

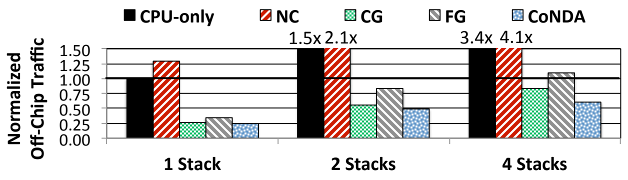

Figure 10: Effect of stack count on data movement (lower is better) for *PageRank* with *arXiv* input graph.

of writebacks required scales *superlinearly* with stack count (6.2x from 1 to 4 stacks; not shown). NC generates more traffic than CPU-only across all stack counts. In contrast, even with 4 stacks, CoNDA reduces off-chip traffic by 82% over CPU-only, again significantly reducing the cost of NDA coherence.

Performance. Figure 11 shows the normalized speedup for each mechanism as the number of stacks increases. Across all stack counts, we find that CoNDA consistently outperforms CPU-only, FG, CG, and NC. Compared to FG, the best existing mechanism at 4 stacks, CoNDA improves performance by 21.5%. This is because there is more off-chip traffic at higher stack counts, which CoNDA reduces, as we see in Figure 10. Unlike CoNDA, as the stack count increases, CG (at 4 stacks) and NC (at 2 and 4 stacks) actually perform *worse* than CPU-only. CG performs worse for two reasons: (1) the high number of flushes; and (2) the increased probability of blocking CPU threads during concurrent execution, with the threads stalled for up to 73.1% of the execution time. FG still outperforms CPU-only as the stack count increases, but its high amount of off-chip coherence requests prevents it from coming close to CoNDA's performance.

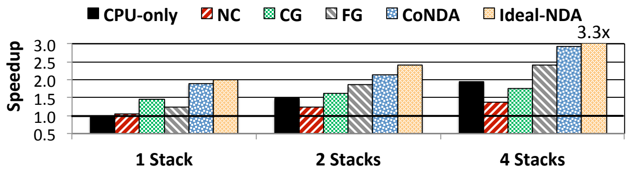

Figure 11: Effect of stack count on speedup for *PageRank* with *arXiv* input graph.

7.5 Effect of Larger Data Sets

Increasing the data set size significantly increases the benefits of CoNDA. This is because the larger data sets result in a significantly larger number of cache misses than the smaller data sets. As a result, more coherence traffic is generated, which, thus, provides more opportunities for CoNDA to eliminate overheads. To demonstrate this, we evaluate three of our applications with larger data sets: (1) *Connected Components* and (2) *Radii*, and (3) *HTAP-1024*. Figure 12 shows the performance improvement normalized to the CPU-only baseline. We find that Ideal-NDA outperforms CPU-only by 9.2x, averaged across these three workloads. CoNDA retains most of the performance benefits of Ideal-NDA, coming within 10.2% on average, and improves performance by 8.4x over CPU-only, 7.7x over NDA-only, and 38.3% over FG (the best prior coherence mechanism). Similar to our observation for the modest size data sets, executing these applications entirely on NDAs (shown by NDA-only) hurts performance significantly, eliminating on average 88.7% of Ideal-NDA's performance improvement. We conclude that as we scale to larger data sets, CoNDA retains its effectiveness at eliminating unnecessary coherence traffic and preserves almost all of the benefits of NDA execution.

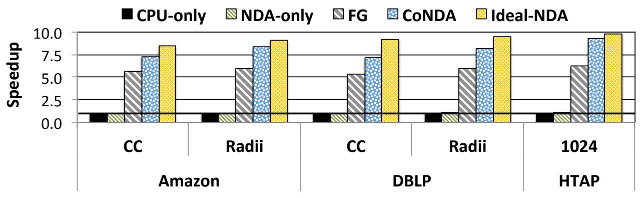

Figure 12: Speedup for larger data sets.

7.6 Effect of Optimistic Execution Duration

As we discuss in Section 5.3, the duration of optimistic execution is determined by the number of addresses that can be held in the signatures without exceeding a target false positive rate, for a fixed signature length. Figure 13 shows how varying the number of addresses affects execution time and off-chip traffic (normalized to CPU-only) for two representative workloads: *Connected Components* with the *Enron* input graph, and *HTAP-128*. We make two observations from the figure. First, as the duration increases from 150 addresses to 350 addresses, the total execution time *increases* by 5.7% and 10.5% for *Connected Components* and *HTAP-128*, respectively. This is because when the duration is longer, there are more opportunities for the CPU and NDA to collide on the same cache line. As a result, the conflict rate (i.e., the fraction of coherence resolution attempts that require re-execution) increases by 18.3% and 41.2%, respectively. Second, as the duration increases, the off-chip traffic *decreases*, by 25.7% and 15.5%, respectively. Every time CoNDA performs coherence resolution, the NDA sends its signatures to the CPU across the off-chip interconnect, and a longer duration requires less frequent coherence resolution.

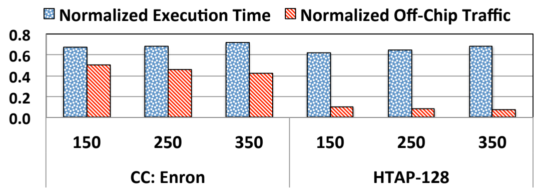

Figure 13: Effect of optimistic execution duration.

Given our signature size (256 B) and target false positive rate (20%), we conclude that a duration of 250 addresses strikes a good balance between execution time and off-chip traffic.

7.7 Effect of Signature Size

We repeat the study from Section 7.6, but this time hold the duration of optimistic execution constant at 250 addresses, and instead vary the size of each signature. Figure 14 shows how the signature size affects execution time and off-chip traffic (normalized to CPU-only). We make two observations from the figure. First, when the signature size increases from 256 B to 1 kB, the execution time *decreases* by 10.1% for *Connected Components* and 10.9% for *HTAP-128*. Increasing the signature size has a similar effect to decreasing the duration of optimistic execution. In this case, the conflict rate decreases mainly because the false positive rate is lower, by 31.4% and 40.5% for the respective workloads. Second, when the signature size increases from 256 B to 512 B and to 1 kB, the off-chip traffic *increases*, by 32.7% and 31.4%, respectively.

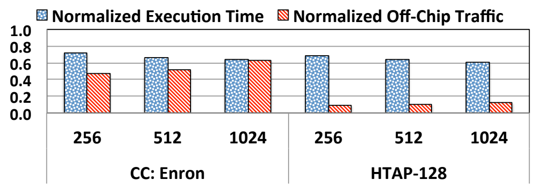

Figure 14: Effect of signature size.

We conclude that for a 250-address duration, using a smaller 256 B signature strikes a good balance between storage overhead, execution time, and off-chip traffic across all of our workloads.

7.8 Effect of Data Sharing Characteristics

The amount of data shared by CPU threads and NDA kernels, and the rate at which these two collide on the same cache lines during concurrent execution, are intrinsic properties of the application and how it is partitioned. We can group applications into three categories: (1) limited sharing; (2) high amount of sharing, infrequent collisions; (3) high amount of sharing, frequent collisions. So far, our work has shown that several important applications fit into Category 2, and CoNDA is much more effective for these applications than existing coherence mechanisms. In this section, we explore how CoNDA performs for applications in the *other two* categories (1 & 3).

Many NDA applications fall under Category 1, and prior works on NDA often assume this behavior (e.g., [21, 25, 27, 59, 97, 126]). To show how CoNDA compares to CG, NC, and FG for such applications, we construct two representative benchmarks inspired by prior works [5, 13, 104]: (1) a matrix tiling operation that is offloaded to an NDA, and (2) a kernel where the NDA performs `memcpy()` and `memset()` on a large region of memory. During NDA execution, the CPU is idle. For these benchmarks, we find that CG, NC, and CoNDA all perform comparably, outperforming CPU-only by 1.83x, 1.85x, and 1.82x, respectively (not shown). All three achieve near-ideal performance, with CoNDA coming within 2.8%. In contrast, FG performs relatively poorly, only coming within 21.1% of Ideal-NDA, due to its high coherence traffic. Even though CG and NC especially cater to such applications, CoNDA still performs competitively, as *very few* rollbacks occur when sharing is limited. Unlike CG and NC, CoNDA provides coherence at a fine granularity (i.e., per cache line) for these applications, making the programming model simpler.

Unlike applications in Categories 1 and 2, applications in Category 3 may not benefit significantly from the concurrent execution of CPU threads and NDA kernels. For such applications, most coherence messages are necessary, and the system would have to spend a large fraction of the total execution time on performing these requests. We develop a representative microbenchmark for this category, where the NDA performs a tiling operation on a matrix while the CPU threads concurrently update and read from the matrix. FG, CG, and NC all hurt the performance of the benchmark, performing 33.4% worse (averaged across the three coherence mechanisms) than CPU-only (not shown) for the same reasons that we discuss in Section 3.2. While CoNDA does significantly better, performing only 2.1% worse than CPU-only, it still falls significantly short of Ideal-NDA, by 51.2%, due to frequent rollbacks. We note, however, that applications where a large portion of the execution is done on an accelerator [24, 50, 51, 54, 67, 77, 99] rarely exhibit this kind of behavior where the CPU and the accelerator collide frequently on shared data.

We conclude that for the vast majority of accelerator-centric applications, we expect that CoNDA (1) either significantly outperforms or performs competitively with existing NDA coherence mechanisms, and (2) achieves near-ideal performance.

8 RELATED WORK

To our knowledge, this is the first work to (1) perform an extensive design exploration of state-of-the-art mechanisms for NDA–CPU coherence, and (2) propose an efficient mechanism for coherence at a fine granularity between NDA kernels and CPU threads. Most recent NDA proposals assume that there is only a limited amount of data sharing between NDA kernels and the CPU threads, and employ naive solutions for coherence, such as (1) assuming that data is never accessed concurrently [3, 48], (2) making NDA data non-cacheable in the CPU [3, 21, 25, 27, 88, 97], (3) flushing *all* dirty cache lines from CPU caches before starting an NDA kernel [26, 74, 121, 129], (4) using coarse-grained coherence [25, 126], or (5) using traditional fine-grained coherence protocols [13]. We show that these approaches are not effective for several important application domains (see Sections 3 and 7).

Other works aim to provide efficient coherence support in non-NDA systems. HSC [96] reduces coherence traffic between the CPU and GPU. However, HSC assumes that both the CPU and GPU are on the same chip, and thus can benefit from CG, unlike many NDA systems. FUSION [70] employs MESI (fine-grained, or FG) coherence between on-chip accelerators and the CPU. As we show in Sections 3 and 7, FG eliminates the majority of the benefits of an NDA due to the high cost of *off-chip* traffic.

Several works [6, 57, 90, 112] reduce on-chip communication between GPU cores, and are also not well-suited for NDA–CPU coherence. For example, the DeNovo protocol [112] communicates at a fine granularity when (1) a cache miss occurs, (2) the protocol registers a write (i.e., informs the directory of the core that holds the modified data), or (3) a cache line is evicted. Since NDA kernels often have poor cache locality, this would cause DeNovo to generate a large amount of off-chip traffic. Furthermore, protocols like DeNovo are not readily compatible with existing coherence protocols, and require us to implement the new protocol across the entire system [112], which is a barrier to adoption. In contrast, CoNDA is designed with the poor cache locality of NDA kernels in mind, and does not require the modification of existing coherence protocols elsewhere in the system. We believe these works [6, 57, 90, 112] can be used to optimize *intra*-NDA coherence.

Our execution model is inspired by Optimistic Concurrency Control (OCC) [71]. Many prior works harness OCC for various purposes [7, 16–18, 40, 44, 45, 81, 84, 98, 109, 115, 122]. CoNDA uses optimistic execution in a different context (NDA–CPU communication) with a different goal (efficient off-chip coherence) from these works. Optimistic execution in CoNDAobserves memory accesses to gain insight into what part of the data will be accessed, and is key to eliminating coherence requests that are unnecessary. Optimistic execution can also be used to enable further optimizations for NDA-based systems, such as if a kernel should execute on an NDA or on the CPU. We leave such optimizations as future work.

While our execution model shares similarities with Bulk-style mechanisms [16, 17, 122, 124, 132] (i.e., mechanisms that speculatively execute chunks of instructions and make use of speculative memory access information to correctly track potential data conflicts during speculation), there is a key distinction. Bulk assumes that *all* compute units are speculative, including the CPU, and, thus, a core must compare all of its memory accesses with those of *all other cores*. In CoNDA, only *some* of the compute units (i.e., NDAs) are speculative. This key distinction leads to a completely different definition of conflicts, which in turn changes many aspects of the system (e.g., the commit protocol, coherence resolution logic, signature implementation). Importantly, CPUs in CoNDA *never* roll back, do *not* require checkpointing (which has a high storage overhead in Bulk), and do *not* need to track any writes to non-NDA data or any reads. As a result, CoNDA is less costly than Bulk. Our execution model also shares similarities with works that use transactional memory (TM) semantics (e.g., [7, 40, 44, 45, 84, 109]), but there are two key differences. First, while any CPU can bundle multiple memory accesses into an atomic transaction in TM, only the NDA bundles multiple memory accesses into a single commit operation in CoNDA. Second, while programmers must express the transaction boundaries in TM, they do *not* need to do so in CoNDA, as CoNDA dynamically determines the duration of optimistic execution, transparently to the programmer.

9 CONCLUSION

Many applications can harness near-data accelerators (NDAs) to gain significant performance and energy benefits. However, enforcing coherence between an NDA and the rest of the system is a major challenge. We extensively analyze NDA applications and existing coherence mechanisms, and observe that (1) a majority of off-chip coherence traffic is unnecessary, and (2) a significant portion of off-chip data movement can be eliminated if a coherence mechanism has insight into NDA memory accesses. Based on our observations, we propose CoNDA, a coherence mechanism that lets an NDA optimistically execute code assuming that it has coherence permissions. Optimistic execution enables CoNDA to gather information on memory accesses, and exploit the information to minimize unnecessary off-chip data movement for coherence. Our results show that CoNDA improves performance and reduces energy consumption compared to existing coherence mechanisms, and comes close to the energy and performance of a no-cost ideal coherence mechanism. We conclude that CoNDA is an effective coherence mechanism for NDAs, and hope that this work encourages the development of other mechanisms for NDA coherence.

ACKNOWLEDGMENTS

An earlier version of this work was published in IEEE CAL [12], and an earlier preprint of this paper was posted on arxiv.org [14]. We thank our industrial partners, especially Alibaba, Facebook, Google, Huawei, Intel, Microsoft, and VMware, for their generous donations. We thank the SAFARI group members for valuable feedback and the stimulating environment. This research was partially supported by the Semiconductor Research Corporation.

REFERENCES

[1] M. E. Acacio *et al.*, "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors," *TPDS*, 2005.
[2] A. Agrawal *et al.*, "Leveraging Near Data Processing for High-Performance Checkpoint/Restart," in *SC*, 2017.
[3] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[4] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[5] B. Akin *et al.*, "Data Reorganization in Memory using 3D-Stacked DRAM," in *ISCA*, 2015.
[6] J. Alsop *et al.*, "Lazy Release Consistency for GPUs," in *MICRO*, 2016.
[7] C. S. Ananian *et al.*, "Unbounded Transactional Memory," in *HPCA*, 2005.
[8] R. Appuswamy *et al.*, "The Case for Heterogeneous HTAP," in *CIDR*, 2017.
[9] J. Arulraj *et al.*, "Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads," in *SIGMOD*, 2016.
[10] N. Binkert *et al.*, "The gem5 Simulator," *Comp. Arch. News*, 2011.
[11] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, 1970.
[12] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2017.
[13] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[14] A. Boroumand *et al.*, "LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures," arXiv:1706.03162 [cs.AR], 2017.
[15] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *WWW*, 1998.
[16] L. Ceze *et al.*, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, 2007.
[17] L. Ceze *et al.*, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *ISCA*, 2006.
[18] J. Devietti *et al.*, "DMP: Deterministic Shared Memory Multiprocessing," in *ASPLOS*, 2009.
[19] DRAMSim2, http://www.eng.umd.edu/ blj/dramsim/.
[20] J. Draper *et al.*, "The Architecture of the DIVA Processing-in-Memory Chip," in *SC*, 2002.
[21] M. Drumond *et al.*, "The Mondrian Data Engine," in *ISCA*, 2017.
[22] H. Esmaeilzadeh *et al.*, "Dark Silicon and the End of Multicore Scaling," in *ISCA*, 2011.
[23] B. Falsafi *et al.*, "Near-Memory Data Services," *IEEE Micro*, 2016.
[24] Y. Fang *et al.*, "UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More," in *MICRO*, 2017.
[25] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
[26] M. Gao *et al.*, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[27] M. Gao *et al.*, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.
[28] S. Ghose *et al.*, "The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption," in *Beyond-CMOS Technologies for Next Generation Computer Design*, 2019.
[29] S. Ghose *et al.*, "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.
[30] S. Ghose *et al.*, "Understanding the Interactions of Workloads and DRAM Types: A Comprehensive Experimental Study," arXiv:1902.07609 [cs.AR], 2019.
[31] V. Gogte *et al.*, "HARE: Hardware Accelerator for Regular Expressions," in *MICRO*, 2016.
[32] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," in *ISCA*, 1983.
[33] Google LLC, "TensorFlow: Mobile," https://www.tensorflow.org/mobile/.
[34] D. Gope *et al.*, "Architectural Support for Server-Side PHP Processing," in *ISCA*, 2017.
[35] Q. Guo *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[36] R. Hadidi *et al.*, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-in-Memory," *ACM TACO*, 2017.
[37] T. J. Ham *et al.*, "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics," in *MICRO*, 2016.
[38] T. J. Ham *et al.*, "DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures," in *MICRO*, 2015.
[39] L. Hammond *et al.*, "Data Speculation Support for a Chip Multiprocessor," in *ASPLOS*, 1998.
[40] L. Hammond *et al.*, "Transactional Memory Coherence and Consistency," in *ISCA*, 2004.
[41] M. Hashemi *et al.*, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.
[42] M. Hashemi *et al.*, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
[43] J. Hennessy *et al.*, "Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges," *Proc. IEEE*, 1999.
[44] M. Herlihy *et al.*, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *ISCA*, 1993.
[45] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," *ISCA*, 1993.
[46] B. Hong *et al.*, "Accelerating Linked-List Traversal Through Near-Data Processing," in *PACT*, 2016.
[47] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[48] K. Hsieh *et al.*, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[49] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 2.1," 2015.
[50] Intel Corp., "Intel Chipset 89xx Series," http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf.
[51] Z. Istvan *et al.*, "Histograms As a Side Effect of Data Movement for Big Data," in *SIGMOD*, 2014.
[52] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSIT*, 2012.
[53] JEDEC Solid State Technology Assn., "JESD235: High Bandwidth Memory (HBM) DRAM," October 2013.
[54] S. Kanev *et al.*, "Mallacc: Accelerating Memory Allocation," in *ASPLOS*, 2017.
[55] Y. Kang *et al.*, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 2012.
[56] S. W. Keckler *et al.*, "GPUs and the Future of Parallel Computing," *IEEE Micro*, 2011.
[57] J. H. Kelm *et al.*, "WAYPOINT: Scaling Coherence to Thousand-Core Architectures," in *PACT*, 2010.
[58] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots," in *ICDE*, 2011.
[59] D. Kim *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[60] G. Kim *et al.*, "Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs," in *SC*, 2017.
[61] G. Kim *et al.*, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *PACT*, 2013.
[62] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[63] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[64] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
[65] H. Kimura *et al.*, "Janus: Transaction Processing of Navigation and Analytic Graph Queries on Many-Core Servers," in *CIDR*, 2017.
[66] O. Kocberber *et al.*, "Asynchronous Memory Access Chaining," *Proc. VLDB Endow.*, 2015.
[67] O. Kocberber *et al.*, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," in *MICRO*, 2013.
[68] P. M. Kogge, "EXECUBE: A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
[69] V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor," in *ICS*, 1998.
[70] S. Kumar *et al.*, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *ISCA*, 2015.
[71] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *TODB*, 1981.
[72] T. Lahiri *et al.*, "Oracle Database In-Memory: A Dual Format In-Memory Database," in *ICDE*, 2015.
[73] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.
[74] J. Lee *et al.*, "Automatically Mapping Code on an Intelligent Memory Architecture," in *HPCA*, 2001.
[75] K. Lim *et al.*, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *ISCA*, 2013.
[76] J. Liu *et al.*, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*, 2018.
[77] J. V. Lunteren *et al.*, "Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator," in *MICRO*, 2012.
[78] M. Mahmoud *et al.*, "IDEAL: Image Denoising Accelerator," in *MICRO*, 2017.
[79] K. Mai *et al.*, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.
[80] D. Makreshanski *et al.*, "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications," in *SIGMOD*, 2017.
[81] J. F. Martínez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," in *ASPLOS*, 2002.
[82] MemSQL, Inc., "MemSQL," http://www.memsql.com/.
[83] N. Mirzadeh *et al.*, "Sort vs. Hash Join Revisited for Near-Memory Execution," in *ASBD*, 2007.
[84] K. E. Moore *et al.*, "LogTM: Log-Based Transactional Memory," in *HPCA*, 2006.
[85] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning," in *MICRO*, 2011.
[86] N. Muralimanohar *et al.*, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.
[87] O. Mutlu *et al.*, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MICPRO*, 2019.
[88] L. Nai *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[89] L. E. Olson *et al.*, "Crossing Guard: Mediating Host-Accelerator Coherence Interactions," in *ASPLOS*, 2017.
[90] M. S. Orr *et al.*, "Synchronization Using Remote-Scope Promotion," in *ASPLOS*, 2015.
[91] M. Oskin *et al.*, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
[92] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *ISCA*, 1984.
[93] D. Patterson *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
[94] A. Pattnaik *et al.*, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[95] J. Picorel *et al.*, "Near-Memory Address Translation," in *PACT*, 2017.
[96] J. Power *et al.*, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO*, 2013.
[97] S. H. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
[98] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *MICRO*, 2001.
[99] B. Reagen *et al.*, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures," in *IISWC*, 2014.
[100] D. Sanchez *et al.*, "Implementing Signatures for Transactional Memory," in *MICRO*, 2007.
[101] SAP SE, "SAP HANA," http://www.hana.sap.com/.
[102] V. Seshadri *et al.*, "The Dirty-Block Index," in *ISCA*, 2014.
[103] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
[104] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[105] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[106] J. M. Shalf and R. Leland, "Computing Beyond Moore's Law," *Computer*, 2015.
[107] Y. S. Shao *et al.*, "Co-Designing Accelerators and SoC Interfaces Using gem5-Aladdin," in *MICRO*, 2016.
[108] Y. S. Shao *et al.*, "Towards Cache-Friendly Hardware Accelerators," in *SCAW*, 2015.
[109] N. Shavit and D. Touitou, "Software Transactional Memory," *Distributed Computing*, 1997.
[110] D. E. Shaw *et al.*, "The NON-VON Database Machine: A Brief Overview," *IEEE Database Eng. Bull.*, 1981.
[111] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *PPoPP*, 2013.
[112] M. D. Sinclair *et al.*, "Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models," in *MICRO*, 2015.
[113] G. Singh *et al.*, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.
[114] Stanford Network Analysis Project, http://snap.stanford.edu/.
[115] J. G. Steffan *et al.*, "A Scalable Approach to Thread-Level Speculation," *ISCA*, 2000.
[116] H. S. Stone, "A Logic-in-Memory Computer," *IEEE Trans. Comput.*, 1970.
[117] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS." *IEEE Data Eng. Bull.*, 2013.
[118] J. Stuecheli *et al.*, "CAPI: A Coherent Accelerator Processor Interface," *IBM JRD*, 2015.
[119] X. Tang *et al.*, "Data Movement Aware Computation Partitioning," in *MICRO*, 2017.
[120] J. Teubner *et al.*, "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware," in *ICDE*, 2013.
[121] P. Tsai *et al.*, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *MICRO*, 2018.
[122] E. Vallejo *et al.*, "Implementing Kilo-Instruction Multiprocessors," in *ICPS 2005*, 2005.
[123] J. Vesely *et al.*, "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," in *ISPASS*, 2016.
[124] T. F. Wenisch *et al.*, "Mechanisms for Store-Wait-Free Multiprocessors," in *ISCA*, 2007.
[125] L. Wu *et al.*, "Q100: The Architecture and Design of a Database Processing Unit," in *ASPLOS*, 2014.
[126] S. L. Xi *et al.*, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.
[127] C. Xie *et al.*, "Processing-in-Memory Enabled Graphics Processors for 3D Rendering," in *HPCA*, 2017.
[128] J. Xue *et al.*, "Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing," in *HPDC*, 2014.
[129] D. P. Zhang *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
[130] M. Zhang *et al.*, "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.
[131] Y. Zhu and V. J. Reddi, "WebCore: Architectural Support for Mobile Web Browsing," in *ISCA*, 2014.
[132] C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization," in *MICRO*, 2002.