Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases with Hardware/Software Co-Design

Amirali Boroumand[†] Saugata Ghose[◇] Geraldo F. Oliveira[‡] Onur Mutlu[‡]
[†]*Google* [◇]*Univ. of Illinois Urbana-Champaign* [‡]*ETH Zürich*

A growth in data volume, combined with increasing demand for real-time analysis (using the most recent data), has resulted in the emergence of database systems that concurrently support transactions and data analytics. These hybrid transactional and analytical processing *(HTAP) database systems can support real-time data analysis without the high costs of synchronizing across separate single-purpose databases. Unfortunately, for many applications that perform a high rate of data updates, state-of-the-art HTAP systems incur significant losses in transactional (up to 74.6%) and/or analytical (up to 49.8%) throughput compared to performing only transactional or only analytical queries in isolation, due to (1) data movement between the CPU and memory, (2) data update propagation from transactional to analytical workloads, and (3) the cost to maintain a consistent view of data across the system.*

We propose Polynesia*, a hardware–software co-designed system for in-memory HTAP databases that avoids the large throughput losses of traditional HTAP systems. Polynesia (1) divides the HTAP system into transactional and analytical processing islands, (2) implements new custom hardware that unlocks software optimizations to reduce the costs of update propagation and consistency, and (3) exploits processing-in-memory for the analytical islands to alleviate data movement overheads. Our evaluation shows that Polynesia outperforms three state-of-the-art HTAP systems, with average transactional/analytical throughput improvements of 1.7×/3.7×, and reduces energy consumption by 48% over the prior lowest-energy HTAP system.*

1. Introduction

Data analytics has become popular due to the rapid growth of data generated annually [1]. Many application domains, such as fraud detection [2–4], business intelligence [5–7], healthcare [8, 9], personalized recommendation [10, 11], and IoT [10], have a critical need to perform *real-time data analysis*, where data analysis needs to be performed using the most recent version of data [12, 13]. To enable real-time data analysis, state-of-the-art database management systems (DBMSs) leverage *hybrid transactional and analytical processing* (HTAP) [14–16]. An HTAP DBMS is a single-DBMS solution that supports both transactional and analytical workloads [12, 14, 17–19]. Ideally, an HTAP system should have three properties [18] to guarantee efficient execution of transactional and analytical workloads. First, it should ensure that both transactional and analytical workloads benefit from their own workload-specific optimizations (e.g., algorithms, data structures). Second, it should guarantee data freshness and data consistency (i.e., access to the most recent version of data) for analytical workloads while ensuring that both transactional and analytical workloads have a consistent view of data across the system. Third, it should ensure that the latency and throughput of both the transactional workload and the analytical workload are the same as if each of them were run in isolation.

We extensively study state-of-the-art HTAP systems (§3) and observe two key problems that prevent them from achieving all three properties of an ideal HTAP system. First, these systems experience a drastic reduction in transactional throughput (up to 74.6%) and analytical throughput (up to 49.8%) compared to when transactional and analytical workloads run in isolation. This is because the mechanisms used to provide data freshness and consistency induce a significant amount of *data movement* between the CPU cores and main memory. Second, HTAP systems often fail to provide effective performance isolation. These systems suffer from severe performance interference because of the high resource contention between transactional workloads and analytical workloads. *Our goal* in this work is to develop an HTAP system that overcomes these problems while achieving all three of the desired HTAP properties, with new architectural techniques.

We propose a novel system for in-memory HTAP databases called *Polynesia*. The key insight behind Polynesia is to partition the computing resources into two isolated new custom processing *islands*: *transactional islands* and *analytical islands*. An island is a hardware–software co-designed component specialized for specific types of queries. Each island consists of (1) a replica of data for a specific workload, (2) an optimized execution engine (i.e., the software that executes queries), and (3) a set of hardware resources (e.g., computation units, memory) that cater to the execution engine and its memory access patterns.

Polynesia meets all desired properties from an HTAP system in three ways. First, by employing processing islands, Polynesia enables workload-specific optimizations for both transactional and analytical workloads (*first desired HTAP property*). Second, we design new hardware accelerators to add specialized capabilities to each island, which we exploit to optimize the performance of several key HTAP algorithms. This includes new accelerators and modified algorithms to propagate transactional updates to analytical islands (§5) and to maintain a consistent view of data across the system (§6). Such new components ensure data freshness and data consistency in our HTAP system (*second desired HTAP property*). Third, we tailor the design of transactional and analytical islands to fit the characteristics of transactional and analytical workloads. The transactional islands use dedicated CPU hardware resources (i.e., multicore CPUs and multi-level caches) to execute transactional workloads since transactional queries have cache-friendly access patterns [20–22]. The analytical islands leverage processing-in-memory (PIM) techniques [23–25] due to the large data traffic analytical workloads produce. PIM systems [20–23, 25–107] mitigate data movement bottlenecks by placing computation units nearby or inside memory.[1] We equip the analytical islands with a new PIM-based analytical engine (§7) that includes simple in-order PIM cores added to the logic layer of a 3D-stacked memory [97, 113, 114], software to handle data placement, and runtime task scheduling heuristics. Our new design enables

[1] Memory manufacturers recently introduced *real* PIM systems that target different application domains (e.g., neural networks [95–98, 108], general-purpose computing [109–111]) and memory technologies (e.g., 3D-stacked DRAM [97, 98, 108], 2D DRAM [95, 96, 110, 111], non-volatile memories [112]).

the execution of transactional and analytical workloads at low latency and high throughput (*third desired HTAP property*).

In our evaluations (§10), we show the benefits of each component of Polynesia, and compare its end-to-end performance and energy usage to three state-of-the-art HTAP systems (modeled after Hyper [115], AnkerDB [116], and Batch-DB [18]). Polynesia outperforms all three, with higher transactional throughput (2.20×/1.15×/1.94×; mean of 1.70×) and analytical throughput (3.78×/5.04×/2.76×; mean of 3.74×), while consuming 48% lower energy than the prior lowest-energy HTAP system. We conclude that Polynesia efficiently provides high-throughput real-time data analysis, while meeting all three desired HTAP properties.

We make the following key contributions in this work:

- We comprehensively examine major system- and architecture-level challenges that hinder throughput in HTAP systems.
- We propose Polynesia, an HTAP system composed of heterogeneous hardware–software co-designed components (called *islands*) that are specialized for executing transactional and analytical workloads. For each island, we develop software-based optimizations and design dedicated hardware resources (e.g., processing-in-memory-based accelerators for analytical islands), both of which cater to the memory usage and computational properties of their target workloads.
- To achieve all three desired HTAP properties, we co-design algorithmic modifications and PIM hardware accelerators for update propagation and data consistency, aiming to reduce data movement overheads.
- We tailor data placement and task scheduling schemes to the memory characteristics of HTAP workloads, aiming to fully exploit main memory bandwidth and system utilization.
- We extensively compare Polynesia against three state-of-the-art HTAP systems. We show that Polynesia provides higher transactional and analytical throughput and lower energy compared to the baseline HTAP systems while meeting all three desired HTAP properties.
- We open-source Polynesia and the complete source code of our evaluation [117].

2. HTAP Background

To enable real-time data analysis, where data analysis needs to be performed using the most recent version of data, a DBMS needs to be capable of efficiently executing analytics on fresh (i.e., the most recent) version of data that is ingested by transactional workloads, which is a challenging task. Several works from industry (e.g., [5, 12, 118–120]) and academia (e.g., [17, 18, 115, 121–127] attempt to address issues with data freshness by proposing various techniques to support both transactional and analytical workloads in a *single* database system. This combined approach is known as *hybrid transactional and analytical processing* (HTAP). To enable real-time analysis, an HTAP system should exhibit three key properties [18].

Property 1: Workload-Specific Optimizations. The HTAP system should provide transactional and analytical workloads with optimizations specific to each of them. Transactional and analytical workloads require different algorithms and data structures, based on the workload's memory access patterns, to achieve high throughput and performance. This leads to different and conflicting optimization techniques (e.g., data layout, hardware design) that can be applied to transactional and analytical workloads.

Property 2: Data Freshness and Data Consistency. The HTAP system should provide the analytics workload with the *most recent version* of data, even when transactions keep updating the data at a high rate. Also, the system needs to guarantee data consistency across the entire system, such that analytical queries observe a consistent view of data, regardless of the freshness of the data.

Property 3: Performance Isolation. The HTAP system should ensure that the latency and throughput of either the transactional or analytical workload is not impacted by running them concurrently within the same system.

Meeting all three desired HTAP properties simultaneously is very challenging [18, 122], as transactional and analytical workloads have different underlying algorithms and access patterns, and optimizing for one property can often require a trade-off in another property.

3. Motivation

There are two major types of HTAP systems: (1) single-instance design systems and (2) multiple-instance design systems. In this section, we study both types, and analyze why neither type can meet all three desired properties of an HTAP system (see §2). To illustrate the key challenges faced by the two types of HTAP systems, we assume a relational DBMS (RDBMS), where data is stored in two-dimensional tables, with tuples (rows in the table) representing a set of data related to different attributes (columns in the table) [128].

3.1. Single-Instance Design

Single-instance HTAP systems [17, 115, 118, 121–123] maintain a single instance of the data that both analytics and transactions work on, ensuring that analytical queries access the most recent version of data. While single-instance design enables high data freshness, it suffers from three major challenges:

(1) High Cost of Consistency and Synchronization. Single-instance-based HTAP systems need to ensure that the data is consistent and synchronized, since analytical and transactional workloads work on the same instance of data concurrently. One approach to consistency is to let both transactions and analytics work on the same copy of data, and use locking protocols [129] to maintain consistency across the system. However, locking protocols lead to throughput bottlenecks for both transactional and analytical workloads [130, 131]. To avoid the throughput bottlenecks incurred by locking protocols [129], single-instance HTAP systems resort to either snapshotting [115, 116, 122, 124] or multi-version concurrency control (MVCC) [17, 131]. Unfortunately, both solutions have significant drawbacks.

Snapshotting: Several HTAP systems (e.g., [115, 122, 124]) use a variation of multiversion synchronization, called snapshotting, to provide consistency via snapshot isolation [132, 133]. Snapshot isolation guarantees that all reads in a transaction see a consistent snapshot of the database state, which is the last committed state before the transaction started.

We analyze the effect of state-of-the-art snapshotting [122, 134] on the throughput of an HTAP system with two transactional and two analytical threads (each running on a separate CPU; see §9 for our evaluation methodology). Fig. 1 (left) shows the transactional throughput with snapshotting, normalized to a zero-cost snapshot mechanism (i.e., a hypothetical ideal baseline where snapshotting operations incur zero delay during execution), for three analytical query counts. We make two observations. First, at 128 analytical queries, snapshotting reduces transactional throughput by 43.4%. Second, the throughput loss increases as more analytical queries are being performed, with a loss of 74.6% for 512 analytical queries. We find that the majority of this throughput loss occurs because `memcpy` is used to create each snapshot, which introduces significant interference among the workloads and generates a large amount of data movement between the CPU and main mem-

ory [35, 40]. The resulting high contention for shared hardware resources directly hurts the throughput.

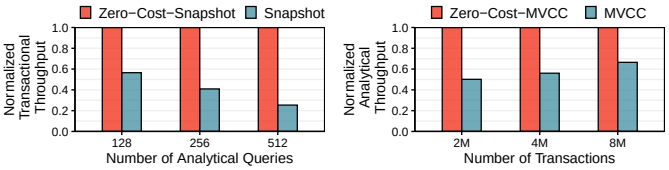

Figure 1: Effect of snapshotting on transactional throughput (left) and MVCC on analytical throughput (right).

MVCC: In MVCC, instead of replacing the old data in a tuple when updates happen (as in the snapshotting approach), the system creates a new timestamped version of the data that is chained together with old versions of the data, forming a pointer-based *version chain*. During execution, instead of reading a separate snapshot, an analytical query can simply use a timestamp to read the most up-to-date version of the data at the time the query starts. Concurrently, a transactional query can insert transactional updates at the end of the version chain with more recent timestamps, without affecting the consistency of the analytical query. As a result, updates never block reads, which is the main reason why many transactional DBMSs have adopted MVCC (e.g., [5, 118, 119]).

However, MVCC is not a good fit for mixed analytical and transactional workloads in HTAP. We study the effect of MVCC on system throughput, using the same hardware configuration that we used for snapshotting. Fig. 1 (right) shows the analytical throughput of MVCC, normalized to a zero-cost version of MVCC (i.e., a hypothetical ideal baseline where MVCC operations incur zero delay during execution), for three transactional query counts. We observe that the analytical throughput significantly decreases (by 42.4% on average across the three transactional query counts) compared to zero-cost MVCC. The root cause is the long version chain caused by frequent updates from the transactional workload. Each version chain is organized as a linked list, which grows with the number of updates from transactional queries. When accessing data in a tuple, the analytical query needs to traverse a lengthy version chain, checking the timestamp in each element of the linked list to locate the most recent version of the data. As analytical queries touch large volumes of data, this generates a large number of random memory accesses, leading to the large throughput loss.

(2) Limited Workload-Specific Optimizations. A single-instance design severely limits workload-specific optimizations, as the instance cannot have different optimizations for each workload. For example, relational transactional engines use a row-wise or N-ary storage model (NSM) for data layout [135], while relational analytics engines employ a column-wise or decomposition storage model (DSM) [136, 137]. It is inherently impossible for a single-instance-based system to efficiently implement both data layouts simultaneously, and many such systems simply choose one of the layouts [115, 122, 123].

(3) Limited Performance Isolation. We evaluate performance interference using the same system configuration that we used for snapshotting and MVCC. Each transactional thread executes 2M queries, and each analytical thread runs 1024 analytical queries. We assume that there is no cost for consistency and synchronization. Compared to running transactional queries in isolation, the transactional throughput drops by 31.3% when the queries run alongside analytics. This is because analytics are very data-intensive and generate a large amount of data movement, which leads to significant contention for shared resources (e.g., the memory system [138–158]). Note that the problem worsens with realistic consistency mechanisms, as they also generate a large amount of data movement.

3.2. Multiple-Instance Design

A second major approach to designing an HTAP system is to maintain multiple instances of the data using replication techniques [134, 159], and specialize each instance to a specific workload (e.g., [5, 18, 19, 119, 120, 124, 125]). Unfortunately, multiple-instance systems suffer from several challenges.

Data Freshness. One of the major challenges in the multiple-instance design is keeping analytical replicas up-to-date even when the transaction update rate is high, without compromising performance isolation [16, 18]. To maintain data freshness, the system needs to propagate transactional updates to analytical replicas (referred as *update propagation*), which requires (1) gathering updates from transactions and shipping them to analytical replicas (*update gathering and shipping*), and (2) performing the necessary format conversion and applying the updates (*update application*). As we discuss below, resource contention and data movement costs become significant performance limiters for multiple-instance HTAP systems.

Update Gathering and Shipping: Given the high update rate of transactions, the frequency of the gathering and shipping process has a direct effect on data freshness. During this process, the system needs to (1) gather updates from different transactional threads, (2) scan them to identify the target memory location corresponding to each update, and (3) transfer each update to the corresponding memory location.

Update Application: The update application process can be challenging due to the need to transform updates from one workload-specific format to another. In RDBMSs, analytical engines use DSM representation to store data [137] and can compress tuples using order-preserving dictionary-based compression (e.g., dictionary encoding [160–162]) to minimize the amount of data that needs to be accessed. In contrast, a single tuple update, stored in the NSM layout by the transactional workload, requires multiple random memory accesses to apply the update in the DSM layout. Compression further complicates this, as columns may need to be decompressed, updated, and recompressed. For compression algorithms that use sorted tuples, such as dictionary encoding, the updates can also lead to expensive shifting of tuples. These operations generate a large amount of data movement and consume many CPU cycles. The challenges are significant enough that some prior works give up on workload-specific optimization to maintain reasonable system performance [18].

We study the effect of update propagation (i.e, update gathering and shipping, and application) on the transactional throughput of a multiple-instance HTAP system (see §9). Fig. 2 shows the transactional throughput for three configurations: (1) a baseline system with zero cost for update propagation (*Zero-Cost-Prop*), (2) a system that performs *only* update gathering and shipping (*Gather-Ship*), and (3) a system that performs update gathering, shipping, *and* application (*Gather-Ship+Apply*). Our system has two transactional threads and two analytical threads (each running on a CPU core) in all three configurations. We make two observations. First, we observe a loss in transactional throughput due to the update gathering and shipping process, which increases as a factor of the update intensity of the transactional query. The transactional throughput of the *Gather-Ship* configuration is 11% lower than that of the *Zero-Cost-Prop* configuration for a 50% update intensity, on average, across different transaction counts. When the transactional queries are more update-intensive (e.g., 80% to 100% updates), the overhead becomes significantly higher, with a throughput loss of 19.9% and 21.2% for 80% and 100% update intensities, respec-

tively. Second, we observe that the update application process leads to an additional loss in transactional throughput. The transactional throughput of the *Gather-Ship+Apply* configuration reduces by 41%, on average, across different transaction counts, compared to that of the *Zero-Cost-Prop* configuration for a 50% update intensity. As the update intensity increases (from 50% to 80%), the loss in transactional throughput further increases (with a 59.0% loss at 80% update intensity).

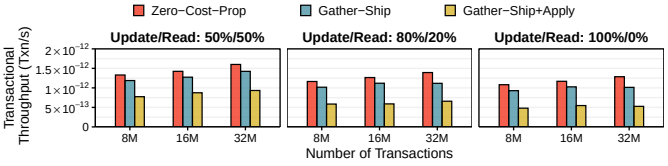

Figure 2: Transactional throughput across different numbers of transactions and different update intensities.

We further analyze the impact of update propagation on the execution time of our HTAP system. Fig. 3 shows the breakdown of execution time during the update propagation process for different numbers of transactions and different update intensities. We make two observations. First, update gathering and shipping accounts for 15.4% of the total execution time, on average, which stems from the large amount of data movement generated by the update gathering and shipping process. Second, the update application process accounts for 23.8% of the execution time, of which 62.6% is spent on (de)compressing columns. Our analysis shows that similar to update gathering and shipping, the update application process also suffers data movement overheads since 30.8% of the total last-level cache misses, on average, are generated during the update application process. We conclude that update propagation accounts for a significant portion of the execution time in our HTAP system (39%, on average), resulting in the loss in transactional throughput that we observe in Fig. 2.

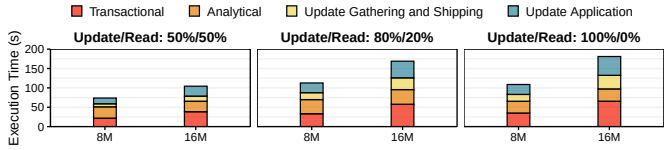

Figure 3: Execution time breakdown across different numbers of transactions update intensities.

Other Major Challenges. We find that maintaining data consistency for multiple instances without compromising performance isolation is very challenging. Updates from transactions are frequently shipped and applied to analytical replicas while analytical queries run. As a result, multiple-instance systems suffer from the same consistency drawbacks that we observe for single-instance systems in §3.1. Another major challenge we find is the limited performance isolation. While separate instances provide partial performance isolation, as transactional queries and analytical queries do not compete for the same copy of data, they still share and contend for underlying hardware resources such as CPU cores and the memory system [138–158].

We conclude that neither single- nor multiple-instance HTAP systems meet the three desired HTAP properties (§2). We, therefore, need a new system that can avoid resource contention and alleviate the data movement costs incurred in HTAP systems.

4. POLYNESIA

We propose Polynesia, which divides the HTAP system into multiple *islands*. An island is a hardware–software co-designed component specialized for specific types of queries. Each island includes (1) a replica of data whose layout is optimized for a specific workload, (2) an optimized execution engine, and (3) a set of hardware resources. Polynesia has two types of islands: (1) a *transactional island*, and (2) an *analytical island*. To avoid the data movement and interference challenges that other multiple-instance HTAP systems face (see §3), we propose to equip each analytical island with (1) *processing-in-memory (PIM) hardware*; and (2) co-designed algorithms and hardware to execute analytical workloads as well as to perform *update propagation* and to guarantee data *consistency*.

Polynesia is a framework that can be applied to many different combinations of transactional and analytical workloads. In this work, we focus on designing an instance of Polynesia that supports relational transactional and analytical workloads.[2] Fig. 4 shows the hardware for our chosen implementation, which includes one transactional island and one analytical island, and is equipped with a 3D-stacked memory similar to the Hybrid Memory Cube (HMC) [113], where multiple vertically-stacked DRAM layers are connected with a *logic layer* using thousands of *through-silicon vias* (TSVs). An HMC chip is split up into multiple *vaults*, where each vault corresponds to a vertical slice of the memory and logic layer.

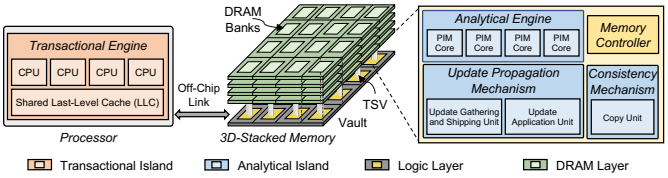

Figure 4: High-level organization of Polynesia hardware.

Polynesia's transactional island uses an execution engine similar to conventional transactional engines [135, 163] to execute a relational transactional workload. The transactional island is equipped with conventional multicore CPUs and multi-level caches, as transactional queries have short execution times, are latency-sensitive, and have cache-friendly access patterns [20–22]. Polynesia's analytical island uses specialized PIM hardware. Inside each vault's portion of the logic layer in memory, we add hardware for the analytical island, including the update propagation mechanism (consisting of the *update gathering and shipping* unit and *update application* unit), the consistency mechanism (*copy unit*), and the *analytical execution engine* (simple programmable in-order PIM cores).[3]

In the next sections, we discuss the detailed design of Polynesia's main components, which includes the update propagation mechanism (§5), the consistency mechanism (§6), and the analytical execution engine (§7). We discuss *both* algorithmic optimizations and novel hardware for each component.

5. Update Propagation Mechanism

We design a new two-part update propagation mechanism. The *update gathering and shipping unit* gathers updates from the transactional island, finds the target location in the analytical island, and frequently pushes these updates to the analytical island. The *update application unit* receives these updates, converts the updates from the transactional to the analytical replica data format, and applies the update to the analytical replica. Our update propagation mechanism leverages novel

[2]Note that our proposed techniques can be applied to other types of analytical workloads (e.g., graphs, machine learning) as well.

[3]The hardware components of Polynesia are a combination of general-purpose processors (GPPs) and fixed-function (ASIC) components. The GPP is responsible for executing transactional queries (at the host processor side) and analytical queries (from within the 3D-stacked DRAM device). The ASIC components are responsible for executing the update propagation and consistency mechanisms we propose. Our current ASIC designs are *not* reconfigurable to ease implementation, but could be extended to be reconfigurable.

algorithms and hardware accelerators tailored to reduce data movement overheads while maintaining data freshness between transactional and analytical islands.

5.1. Update Gathering and Shipping

Algorithm. Our update gathering and shipping mechanism includes three major stages. For each thread in the transactional engine, Polynesia stores an ordered *update log* for the queries performed by the thread. Each update log entry contains four fields: (1) a commit ID (a timestamp used to track the total order of all updates across threads), (2) the type of the update (insert, delete, modify), (3) the updated data, and (4) a record key (e.g., pair of row-ID and column-ID) that links this particular update to a column in the analytical replica. The update gathering and shipping process is triggered when the total number of pending updates reaches the final log capacity, which we set to 1024 entries (see §5.2). Stage 1 scans the per-thread update logs, and merges them into a single *final log*, where all updates are sorted by the commit ID.

Stage 2 finds the memory location of the corresponding column (in the analytical replica) associated with each update log entry. We observe that this stage is one of the major bottlenecks of update gathering and shipping, because the fields in each tuple in the transactional island are distributed across different columns in the analytical island. Since the column size is typically very large, finding the memory location of each update is a very time-consuming process. To overcome this, we maintain a hash index of the stored data on the `(column,row)` key, and use that to find the corresponding column's location for each update in the final log. We use the `modulo` operation as the hash function. Our hash table uses bucket hashing with separate chaining to handle collisions, and hash buckets containing a linked list of keys are stored in main memory. We size our hash table based on the column partition size.[4] We place the updates from the final log for each column in a *column buffer*, based on the output of the hash unit. At the end of this stage, there are multiple column buffers, each corresponding to a column in the analytical replica, which are ready to be shipped (i.e., written) to the analytical island. Stage 3 ships all column buffers to each column in the analytical replica.

Hardware. We find that despite our best efforts to minimize overheads, our algorithm has three major bottlenecks that keep it from meeting data freshness and performance isolation requirements: (1) the scan and merge operation in Stage 1, (2) hash index lookups in Stage 2, and (3) transferring the column buffer contents to the analytical islands in Stage 3. These primitives generate a large amount of data movement and account for 87.2% of our algorithm's execution time. To avoid these bottlenecks, we design a new hardware accelerator, called the *update gathering and shipping unit*, that speeds up the key primitives of the update gathering and shipping algorithm. We add this accelerator to each of Polynesia's in-memory analytical islands.

Fig. 5 shows the high-level architecture of our in-memory update gathering and shipping unit. The update gathering and shipping unit consists of three building blocks: (1) a merge unit, which merges the per-thread sorted update logs into the single final log (Stage 1 of our algorithm); (2) a hash lookup unit, which decouples hash lookup operations into two steps, i.e., bucket address generation and bucket access and traversal (Stage 2 of our algorithm); and (3) a copy unit, which issues concurrent read/write memory requests to main memory (accelerating data copy operations required during hash table indexing in Stage 2 and column buffer shipping in Stage 3 of our algorithm).

The *merge unit* consists of 8 FIFO input queues, where each input queue corresponds to a sorted update log. Each input queue can hold up to 128 updates, which are streamed from DRAM. The *hash lookup unit* consists of a front-end engine (a finite-state machine, or FSM, responsible for bucket memory address generation), four probe units (FSMs responsible for bucket access and traversal), and a small reorder buffer (to track in-flight hash lookups issued to main memory). The hash lookup unit (1) decouples key hashing and bucket address generation from the actual bucket access/traversal to allow for concurrent hashing operations; and (2) guarantees that updates remain in the same order as executed by the transactional engine, by using a small reorder buffer to maintain sequential commit order for completed hash lookups that are sent to the copy engine. The *copy unit* consists of a read/write tracking buffer and multiple fetches and write units (we describe our copy unit in detail in §6).

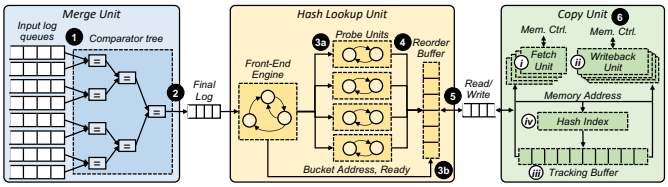

Figure 5: Update gathering and shipping unit architecture.

The update gathering and shipping unit works in five steps. First, the merge unit finds the oldest entry among the input log queue heads using a 3-level comparator tree (❶ in Fig. 5) and adds it to the tail of the final log, which consists of a ninth FIFO queue. Second, the final log (❷) is sent to the hash lookup unit to determine the target column address for each update. Third, the front-end engine in the hash lookup unit performs two operations in parallel: it (i) fetches one hash key from the final update log, computes the hash function, and sends the hash key address to probe units (❸a); and (ii) allocates an entry (with the bucket address and a ready bit) in the reorder buffer, for each hash lookup (❸b). Fourth, a free probe unit (❹) takes the bucket address received from the front-end engine, and issues read requests to the copy unit in order to traverse the bucket's linked list of keys (❺). Once the probe unit reaches the end of the bucket linked list, it will hold the corresponding column for an entry in the final update log. Fifth, the probe unit issues write operations (❻), using the column memory address retrieved from the hash to the copy unit in order to transfer the update in the final log to the target column buffer. This completes the update gathering and shipping process for one entry in the final log.

5.2. Update Application

Similar to other relational analytical DBMSs, our analytical engine uses the DSM data layout (see §3.1) and *dictionary encoding* [118, 137, 160–162, 165, 166]. With dictionary encoding, each column in a table is transformed into a compressed column using encoded fixed-length integer values, and a dictionary stores a sorted mapping of unencoded values to encoded values. As we discuss in §3.2, the layout conversion (our transactional island uses NSM) and column compression make the update application process challenging. We design a new update application mechanism for Polynesia that uses hardware–software co-design to address these challenges.

Algorithm. We first discuss an initial algorithm that we develop for update application. We assume that each column has n entries, and that we have m updates. The algorithm has

[4] Similar to conventional analytical DBMSs, we can use soft partitioning [18, 160, 164] to address scalability issues when the column size increases beyond a certain point. Thus, the hash table size does *not* scale with column size.

four steps. In *Step 1,* the algorithm decompresses the encoded column by scanning the column and looking up in the dictionary to decode each item. This requires *n random accesses* to the dictionary. In *Step 2,* the algorithm applies updates to the decoded column one by one. In *Step 3,* it constructs a new dictionary, by sorting the updated column and calculating the number of fixed-length integer bits required to encode the sorted column. Dictionary construction is computationally expensive ($\mathcal{O}((n+m)\log(n+m))$; where m is the number of updates to a column and n is the number of entries in a column) because we need to sort the entire column. In *Step 4,* the algorithm compresses the new column using our new dictionary. While entry decoding happens in constant time, encoding requires a logarithmic complexity search through the dictionary (since the dictionary is sorted).

This initial algorithm is memory intensive (Steps 1, 2, 4) and computationally expensive (Step 3). Having hardware support is *critical* to enabling low-latency update application and performance isolation. While PIM may be able to help, our initial algorithm is not well-suited for PIM for two reasons, and we optimize the algorithm to address both reasons.

Optimization 1: Two-Stage Dictionary Construction. We eliminate column sorting from Step 3, as it is computationally expensive. Prior work [167, 168] shows that to efficiently sort more than 1024 values in hardware, we should provide a hardware partitioner to split the values into multiple chunks, and then use a sorter unit to sort chunks one at a time. This requires an area of 1.13 mm^2 [167, 168]. Unfortunately, since tables can have millions of entries [165], we would need tens to hundreds of sorter units to construct a new dictionary, easily exceeding the total area budget of 4.4 mm^2 per vault in our baseline 3D-stacked DRAM [52, 75, 76].

To eliminate column sorting, we sort *only* the dictionary, leveraging the fact that (1) the existing dictionary is already sorted, and (2) the new updates are limited to 1024 values. Our optimized algorithm initially builds a sorted dictionary for only the updates, which requires a single hardware sorter (a 1024-value bitonic sorter with an area of only 0.18 mm^2 [168]). Once the update dictionary is constructed, we now have two sorted dictionaries: the old dictionary and the update dictionary. We merge these into a single dictionary using a linear scan ($\mathcal{O}(n+m)$; where m is the number of updates to a column and n is the number of entries in a column), and then calculate the number of bits required to encode the new dictionary.

Optimization 2: Reducing Random Memory Accesses. To reduce the algorithm's memory intensity (which is a result of random memory lookups), we maintain a hash index that links the old encoded value in a column to the new encoded value. This avoids the need to decompress the column and add updates, eliminating data movement and random memory accesses for Steps 1 and 2, while reducing the number of dictionary lookups required for Step 4. The only remaining random memory accesses are for Step 4, which decrease from $\mathcal{O}((n+m)\log(n+m))$ to $\mathcal{O}(n+m)$.

Our optimized algorithm has three steps. First, we sort the updates to construct the update dictionary. Second, we merge the old dictionary and the update dictionary to construct the new dictionary and hash index. Third, we use the hash index and the new dictionary to find the new encoded value for each entry in the column.

Hardware. We design a hardware implementation of our optimized algorithm, called the *update application unit*, and add it to each in-memory analytical island. The unit consists of three building blocks: a *sort unit*, a *hash lookup unit*, and a *scan/merge unit*. Our sort unit uses a 1024-value bitonic sorter, whose basic building block is a network of comparators. These comparators are used to form *bitonic sequences*, sequences where the first half of the sequence is monotonically increasing and the second half is monotonically decreasing. The hash lookup uses a simpler version of the component that we designed for the update gathering and shipping unit. The simplified version does not use a reorder buffer, as there is no dependency between hash lookups for update application. We use the same number of hash units (empirically set to 4), each corresponding to one index structure, to parallelize the compression process. For the merge unit, we use a similar design from our update gathering and shipping unit.

6. Consistency Mechanism

We design a new consistency mechanism for Polynesia in order not to compromise either the throughput of analytical queries or the rate at which updates are applied. This sets two requirements for our mechanism: (1) analytical queries must be able to run continuously without slowdown; and (2) the update application process should not be blocked by long-running analytical queries. This means that our mechanism needs a way to allow analytical queries to run concurrently with updates, without incurring the long-chain read overheads of similar mechanisms such as MVCC (see §3.1). Our consistency unit relies on our novel in-memory hardware *copy unit*, which can fully exploit the large internal memory bandwidth of 3D-stacked memories [114].

Algorithm. Our mechanism relies on a combination of snapshotting [115] and versioning [159] to provide snapshot isolation [132, 133] for analytical queries. Our consistency mechanism is based on two key observations: (1) updates are applied at a column granularity, and (2) snapshotting a column is cost-effective using PIM logic. We assume that for each column, there is a chain of snapshots where each chain entry corresponds to a version of the column. Unlike version chains in MVCC, each version is associated with a column, not a tuple.

We adopt a lazy approach (late materialization [169]), where Polynesia does not create a snapshot every time a column is updated. Instead, on a column update, Polynesia marks the column as dirty, indicating that the snapshot chain does not contain the most recent version of the column data. When an analytical query arrives, Polynesia checks the column metadata, and creates a new snapshot only if (1) any of the columns are dirty (similar to Hyper [115]), and (2) no current snapshot exists for the same column (we let multiple queries share a single snapshot). During snapshotting, Polynesia updates the head of the snapshot chain with the new value, and marks the column as clean. This provides two benefits. First, the analytical query avoids the version chain traversals and timestamp comparisons performed in MVCC, as the query only needs to access the head of the version chain at the time of the snapshot. Second, Polynesia uses simple yet efficient garbage collection: when an analytical query finishes, snapshots no longer in use by any query are deleted (except for the head of the snapshot chain).

To maintain high data freshness, our consistency mechanism always allows transactional updates to directly update the main replica using our two-phase update application algorithm (§5.2). In Phase 1, the algorithm constructs a new dictionary and a new column. In Phase 2, the algorithm atomically updates the main replica with pointers to the new column and dictionary.

Hardware. Our algorithm's success at satisfying the first requirement for a consistency mechanism (i.e., no slowdown for analytical queries) relies heavily on its ability to perform fast memory copies to minimize the snapshotting latency. Therefore, we add a custom *copy unit* to each of Polynesia's in-memory

analytical islands. We have two design goals for the unit. First, it needs to be able to issue multiple memory accesses concurrently. This is because (1) we are designing the copy engine for an arbitrarily-sized memory region (e.g., a column), which is often larger than the memory access granularity per vault (8–256 B) in an HMC-like memory; and (2) we want to fully exploit the internal bandwidth of 3D-stacked memory. Second, when a read for a copy completes, the accelerator should immediately initiate the write.

Our copy unit (Fig. 5, right) satisfies both design goals. To issue multiple memory accesses concurrently, we leverage the observation that these memory accesses are independent. We use multiple fetch (ⓘ in Fig. 5) and writeback (ⓘⓘ) units, which read from or write to source/destination regions in parallel. To satisfy the second design goal, we need to track outstanding reads, as they may come back from memory out of order. Similar to prior work on accelerating `memcpy` [170], we use a *tracking buffer* (ⓘⓘⓘ) in our copy unit. The buffer allocates an entry for each read issued to memory. An entry contains a memory address and a ready bit. Once a read completes, we find its corresponding entry in the buffer and set its ready bit to trigger the write.

We find that the buffer lookup limits the performance of the copy unit, as each lookup results in a full buffer scan, and multiple fetch units perform lookups concurrently (generating high contention). To alleviate this, we design a hash index (ⓘⓥ) based on the memory address to determine the location of a read in the buffer. We use a similar design as the hash lookup unit in our update gathering and shipping unit (§5.1).

Our copy unit can be further accelerated by using mechanisms that provide fast in-DRAM data copy support [33, 35, 40, 42, 94, 171, 172].

7. Analytical Engine

The analytical execution engine, whose hardware design is illustrated in Fig. 4, performs the analytical queries. Our analytical engine consists of four simple programmable in-order PIM cores, placed within a vault of our 3D-stacked memory (i.e., a total of 64 PIM cores across the entire analytical island). When an analytical query arrives, the analytical engine parses the query and generates an algebraic query plan consisting of *physical operators* (e.g., scan, filter, join). In the query plan, operators are arranged in a tree where data flows from the bottom nodes (leaves) toward the root, and the result of the query is stored in the root. The analytical execution engine employs the top-down Volcano (Iterator) execution model [173, 174] to traverse the tree and execute operators while respecting dependencies between operators. Analytical queries typically exhibit a high degree of both intra- and inter-query parallelism [160, 164, 175]. To exploit this, the analytical engine decomposes an analytical query into multiple tasks, each of which is a sequence of one or more operators. The analytical engine (task scheduler) then schedules the tasks with the goal of executing multiple independent tasks in parallel.

Efficient analytical query execution strongly depends on (1) data placement, (2) the task scheduling policy, and (3) how each physical operator is executed. Like prior works [76, 176], we find that the execution of physical operators of analytical queries significantly benefit from PIM. However, without an HTAP-aware and PIM-aware data placement strategy and task scheduler, PIM logic for operators alone *cannot* provide significant throughput improvements.

We design a new analytical execution engine based on the characteristics of our in-memory hardware. As we discuss in §4, Polynesia uses 3D-stacked memory [113, 114, 177] that contains multiple vaults. Each vault (1) provides only a fraction (e.g., 8 GB/s) of the total bandwidth available in a 3D-stacked memory; (2) has limited power and area budgets for PIM logic; and (3) can access its own data faster than it can access data stored in other vaults, which take place through a vault-to-vault interconnect (e.g., as in [51, 178–182]). We take these limitations into account as we design our data placement mechanism and task scheduler.

7.1. Data Placement

We evaluate three data placement strategies (shown in Fig. 6) for Polynesia. Our analytical engine uses the DSM layout to store data, and makes use of dictionary encoding [161] for column compression. Our three strategies affect which vaults the compressed DSM columns and dictionary are stored in.

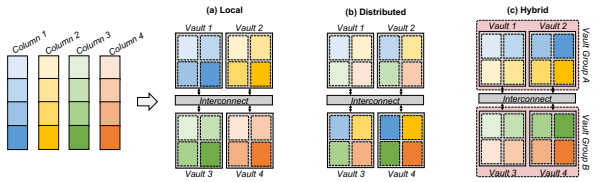

Figure 6: Three data placement strategies.

Strategy 1 (*Local*) stores the entire column (with dictionary) in one vault (Fig. 6a), which improves analytical query throughput by making dictionary lookups and column accesses local to a single vault (out of the 16 vaults we evaluate in §10). It also simplifies the update application process, since each vault has its own update application unit. However, this data placement strategy suffers from the limited area and memory bandwidth available in a single vault since an analytical thread can only execute with the portion of the analytical replica placed at its assigned vault, thus hurting throughput. Strategy 2 (*Distributed*) partitions each column across *all* vaults in a memory chip (Fig. 6b), which allows the analytical engine to exploit the entire internal bandwidth of the 3D-stacked memory and to use all the available PIM logic to serve each analytical query. However, this data placement strategy makes update application challenging due to gather-scatter operations that span all vaults, which reduces throughput.

Strategy 3 (*Hybrid*), which Polynesia employs for data placement, overcomes the challenges of *Local* and *Distributed* by partitioning a set of columns across a *vault group* (Fig. 6c). Each vault group consists of a fixed number of vaults, each of which holds a portion of the column. A group with v vaults provides v times the memory bandwidth and v times the PIM logic power/area budget for a column compared to *Local*. The number of vaults per group is critical for efficiency: too many vaults can complicate the update application process, while not enough vaults can degrade throughput. We empirically find that four vaults per group strikes a good balance (in a 3D-stacked memory with 16 vaults).

The *Hybrid* data placement strategy still needs to perform inter-vault accesses within each vault group. To overcome this, we leverage an observation from prior work [165] that the majority of columns have only a limited (up to 32) number of distinct values. This means that the entire dictionary incurs negligible storage overhead (~2 KB). To avoid inter-vault dictionary accesses during update application, *Hybrid* keeps a copy of the dictionary in each vault. Such an approach is significantly costlier if employed under *Distributed*, as each vault would replicate every other vault's dictionary, generating large storage overheads due to the vault count (e.g., 16).

7.2. Task Scheduler

Polynesia's task scheduler plays a key role in (1) exploiting inter- and intra-query parallelism, and (2) efficiently utilizing

hardware resources. For each query, the task scheduler (1) decides how many tasks to create, (2) finds how to map these tasks to the available resources (*PIM cores*), and (3) guarantees that dependent tasks are executed in order. We first design a *basic task scheduler heuristic* that generates tasks (statically at compile time) by disassembling the operators of the query plan into operator instances (i.e., an invocation of a physical operator on some subset of the input tuples) based on (1) which vault groups the input tuples reside in; and (2) the number of available PIM threads in each vault group, which determines the number of tasks generated. The basic task scheduler heuristic inserts tasks into a global work queue in an order that preserves dependencies between operators, monitors the progress of PIM threads, and assigns each task to a free thread (push-based assignment [183]).

We design an *optimized task scheduler heuristic* by employing three optimizations on top of our basic task scheduler heuristic so our task scheduler can better fit our PIM system. First, we design a pull-based task assignment strategy [183], where PIM threads pull tasks from the task queue at runtime. This eliminates the need for a runtime component and allows PIM threads to dynamically balance their loads. We introduce a local task queue for each vault group. Each PIM thread examines its own local task queue to retrieve its next task. Second, we optimize the heuristic to allow for finer-grained tasks. Instead of mapping tasks statically, we partition input tuples into fixed-size segments (i.e., 1000 tuples) and create an operator instance for each partition. The task scheduler then generates tasks for these operator instances and inserts them into their corresponding task queues (where those tuple segments reside). A large number of tasks increases opportunities for load balancing. Third, we allow a PIM thread to steal tasks from a remote vault if its local queue is empty. This enables us to potentially use all available PIM threads to execute tasks, regardless of the data placement. Each PIM thread first attempts to steal tasks from other PIM threads in its own vault group, because the thread already has a local copy of the full dictionary in its vault, and needs inter-vault accesses only for the column partition. If there is no task to steal in its vault group, the PIM thread attempts to steal a task from a remote vault group.

7.3. Hardware Design

Our analytical engines leverage the design of our new data placement strategy and task scheduler to expose intra-query parallelism and the available vault bandwidth to PIM cores. We add four simple programmable in-order PIM cores [51, 57, 76] to each vault. We run a PIM thread on each core, and we use these cores to execute the task scheduler and other parts of the analytical engine (e.g., the query parser).

We find that our optimized scheduler heuristic significantly increases data sharing between PIM threads. This is because within each vault group, all 16 PIM threads access the same local task queue, and must synchronize their accesses. The problem worsens when other PIM threads attempt to steal tasks from remote vault groups, especially for highly-skewed workloads. To avoid excessive accesses to DRAM and let PIM threads share data efficiently, we implement a simple fine-grained coherence technique (as in [20, 22]), which uses a local PIM-side directory in the logic layer to implement a low-overhead coherence protocol.

8. Integrating Polynesia in the Cloud

We discuss the applicability of our proposal to cloud environments. First, we discuss the benefits of integrating Polynesia in the cloud. Second, we discuss alternative cloud-based hardware solutions for HTAP systems.

Integrating Polynesia in the Cloud. We believe that cloud providers would be willing to add extra ASICs into their environment as long as doing so provides potential improvements to a wide enough range of cloud workloads. Several cloud providers already integrate ASIC accelerators in their datacenters. For example, Google Cloud, Microsoft Azure and Amazon AWS use dedicated hardware for neural network inference (Google Cloud TPU [184, 185], Habana Goya [186]), neural network training (AWS Trainium [187], Habana Gaudi [186]), video transcoding (Google VCU [188]), and compression/encryption/data authentication (Microsoft's Project Corsica [189]). Recently, different commercial PIM designs [95–98, 108–111, 190], which target large cloud systems, have been proposed.[1] Since Polynesia's hardware can provide performance benefits (§10.1) across a wide range of applications that depend on large amounts of data and analytics, we believe it is an attractive architecture for cloud providers. Polynesia's energy savings (§10.6) are also attractive to a cloud environment, as it can significantly lower operating costs. As prior works show [188, 191], ASIC accelerators are a viable solution for cloud environments, depending on the scale of the computation. Even though we cannot accurately predict the price of integrating Polynesia into a cloud system due to unknown parameters (e.g., non recurring engineering expenses), we believe it would be a beneficial solution for cloud systems due to the widespread use of the database applications it targets.

Alternative Hardware Solutions. A straightforward alternative solution to improve performance for a cloud HTAP system is to scale up hardware resources (especially core count). However, as we demonstrate in this paper, several key bottlenecks in HTAP systems are not compute-bound but are instead memory-bound and cannot benefit simply from more cores (as memory bandwidth, latency, and data movement remain as bottlenecks with more cores). PIM hardware can overcome the memory bandwidth bottlenecks by tapping into the significantly higher internal memory bandwidth of 3D-stacked memories and also improving latency and energy efficiency with reduced data movement. Therefore, our custom hardware has distinct benefits unachievable by additional cores.

9. Methodology

We use and heavily extend state-of-the-art transactional and analytical engines to implement various single- and multiple-instance HTAP configurations. We use DBx1000 [163, 192] as the starting point for our transactional engine, and we implement an in-house analytical engine similar to C-store [137]. Our analytical engine supports key physical operators for relational analytical queries (select, filter, aggregate and join), and supports both NSM and DSM layouts, and dictionary encoding [160–162]. For consistency, we implement both snapshotting (similar to software snapshotting [134], with snapshots taken only when dirty data exists) and MVCC (adopted from DBx1000 [163]).

Our baseline single-instance HTAP system stores the single data replica in main memory. Each transactional query randomly performs reads or writes on a few randomly-chosen tuples from a randomly-chosen table. Each analytical query uses select and join on randomly-chosen tables and columns. Our baseline multiple-instance HTAP system models a similar system as our single-instance baseline, but provides the transactional and analytical engines with separate replicas (using the NSM layout for transactions, and DSM with dictionary encoding for analytical queries). Across all baselines, we have 4 transactional and 4 analytical worker threads.

We simulate Polynesia using gem5 [193], integrated with DRAMSim2 [194] to model an HMC-like 3D-stacked

DRAM [113]. Table 1 shows our system configuration. For the analytical island, each vault of our 3D-stacked memory contains four PIM cores and three fixed-function accelerators (update gathering and shipping unit, update application unit, copy unit). For the PIM core, we model a core similar to the ARM Cortex-A7 [195], as in prior works [44, 196].

Table 1: Evaluated system configuration.

Processor (Transactional Island)	4 OoO cores, each with 2 HW threads, 8-wide issue; *L1 I/D Caches*: 64 kB private, 4-way assoc.; *L2 Cache:* 8 MB shared, 8-way assoc.; *Coherence*: MESI [197]
PIM Cores (Analytical Engine)	4 in-order cores per vault, 2-wide issue, *L1 I/D Caches*: 32 kB private, 4-way assoc. *Coherence*: MESI [197]
3D-Stacked Memory	4 GB cube, 16 vaults per cube; *Internal Bandwidth:* 256 GB/s; *Off-Chip Channel Bandwidth:* 32 GB/s

We model the in-order cores and specialized accelerators in gem5 using a methodology similar to [20–22, 52]. Similar to prior works [20–22], we implement a simple fine-grained coherence technique between PIM cores. Updates between islands are propagated using our update propagation mechanism and shared memory. The updates are stored in shared memory. To allow coordination, ordering, and synchronization between different parts of islands, we need to provide coherence between CPU cores and PIM logic. We employ a simple fine-grained coherence technique (MESI [197]), which uses a local PIM-side directory [198] in the logic layer to maintain coherence between PIM cores and to enable low-overhead fine-grained coherence between PIM logic and the CPUs.

We open-source Polynesia and the complete source code of our evaluation [117].

Experimental Setup Validation. We validate our experimental setup, including our workloads and gem5-based simulation, in two ways. First, to implement our HTAP system, we adopt *prior* transactional (DBx1000) and analytical (C-store) engines. Similarly, we tailor our consistency models based on prior works [134, 163]. We validate the correctness of each model modification we make by comparing the outputs *after* modifications with the outputs *before* modifications. Second, to model the hardware components of our system, we use an already validated simulator (i.e., gem5). While we expect to see similar results when Polynesia is implemented on top of high-end HTAP systems, it is currently very challenging for us to confirm that in real hardware since Polynesia requires hardware modifications that are costly to realize or prototype using available tools and frameworks.[5]

9.1. Simplifying Assumptions About Realistic HTAP

We had to make simplifying assumptions to capture most of the key properties of a real HTAP system in an architectural simulator. We list such assumptions next.

Scheduling Transactional and Analytical Queries. We assume that the partitioning between transactional and analytical queries is made by the DBMS. We do not propose a specific dynamic scheduling mechanism to identify the query type and map the query to the appropriate island. We believe that such a dynamic scheduling mechanism is orthogonal to our work, and can be an extension of existing mechanisms [200–202].

Modern Transactional and Analytical Engines. Our goal in this paper is to (1) provide insights about the major challenges in HTAP systems, (2) propose a framework that can address these challenges, and (3) evaluate such a framework as a case study. To achieve this goal, we employ transactional and analytical models that may not necessarily represent the most recent implementations of OLTP and OLAP workloads, but are enough to demonstrate the challenges and drawbacks that most HTAP systems face due to data movement overheads. The transactional and analytical models we use for our reference implementation of Polynesia have the following advantages. First, our transactional engine (DBX1000) is a simple, in-memory DBMS that provides several concurrency control models, including MVCC and the optimistic concurrency control model, a similar but improved version of the concurrency control algorithm Microsoft's Hekaton [203] employs. Many works [204–207] build on top of DBX1000, proposing both hardware [207] and software [204–206] optimizations. Second, our analytical engine is a column-based analytical engine adopted from C-store, which employs a Volcano-style processing model [208]. While modern analytical engines may employ more efficient processing models (e.g., based on vectorization [209] or pushing tuples [210]), our Volcano-style processing model is still present in many relational DBMSs [211]. To conclude, we use such transactional and analytical models since they are enough to reach our goals. We believe that the challenges we identify in HTAP systems are also present in systems with more modern transactional and analytical engines, but we leave detailed studies of large-scale systems to future works.

Assumptions About the Encodings/Formats. Our analytical engine uses dictionary encoding, similarly to prior works [161, 162]. However, dictionary encoding might not be beneficial to a particular real system if there are not enough common values across columns to employ dictionary compression.

10. Evaluation

We demonstrate the advantages of Polynesia by evaluating (i) its end-to-end performance benefits compared against state-of-the-art HTAP systems using both synthetic and real-world queries; (ii) the individual performance of the three major components of our proposal (i.e., our update propagation technique, consistency mechanism, and analytical engine); (iii) how Polynesia performs as the dataset size grows; and (iv) the energy savings Polynesia provides.

10.1. End-to-End System Performance Analysis

Fig. 7 (left) shows the transactional throughput of six DBMSs: (1) Single-Instance-Snapshot (*SI-SS*; modeled after the Hyper HTAP system [115] with software snapshotting [134]); (2) Single-Instance-MVCC (*SI-MVCC*; modeled similar to AnkerDB [116]); (3) *MI+SW*, an improved version of Multiple-Instance, modeled similar to Batch-DB [18] and including all of our software optimizations for Polynesia (except those specifically targeted for PIM); (4) *MI+SW+HB*, a hypothetical version of *MI+SW* with 8× its main memory bandwidth (256 GB/s), equal to the internal memory bandwidth of a commercially available 3D-stacked memory (HBM 2.0 [177]); (5) *PIM-Only*, a hypothetical version of *MI+SW* that uses general-purpose PIM cores to run both transactional and analytical workloads; and (6) *Polynesia*, our full hardware–software proposal. Each one of these baselines (i) isolates one of our new components and shows how much benefit we get out of them; (ii) is modeled after state-of-the-art (software-only) HTAP systems. Note that we do not compare Polynesia against hardware-based HTAP systems since no prior work has proposed to use tailored hardware accelerators for HTAP. We normalize throughput to an ideal transaction-only DBMS (*Ideal-Txn*) for three transaction counts. *Ideal-Txn* indicates the peak transactional throughput if we run the transactional workload in isolation.

[5] While PiDRAM [94] and MEG [199] are potential promising FPGA-based platforms to emulate the functional correctness of proposed PIM designs, they are not useful for our studies because (1) neither can model the performance and energy usage of PIM, (2) PiDRAM does not support the 3D-stacked memories used by Polynesia, and (3) the MEG framework is not available at the time of writing.

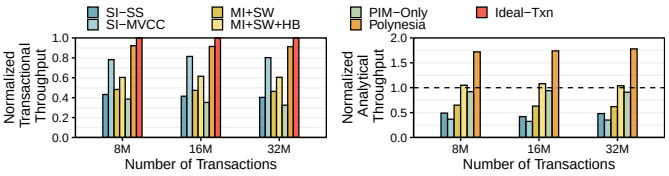

Figure 7: Normalized transactional (left) and analytical (right) throughput for end-to-end HTAP systems.

We make five observations from Fig. 7 (left). First, Polynesia improves transactional throughput by 51.0% over *MI+SW+HB*, and by 14.6% over *SI-MVCC*, while achieving 91.6% of the transactional throughput of *Ideal-Txn*, on average across the three transaction counts. Polynesia's higher transactional throughput stems from (1) using custom PIM logic for analytical queries and update propagation and consistency mechanisms that reduce resource contention significantly, and (2) reducing off-chip memory bandwidth contention by decreasing data movement. Second, of the single-instance DBMSs, *SI-MVCC* performs best, achieving 80.0% of the throughput of *Ideal-Txn*, on average across the three transaction counts. Its use of MVCC over snapshotting overcomes the high performance penalties incurred by *SI-SS*. Third, *SI-MVCC* significantly outperforms the two software-only multiple-instance DBMSs (*MI+SW* and *MI+SW+HB*), even though the latter two enable software optimizations and greatly increase the memory bandwidth. Such performance degradation is due to the lack of performance isolation in both *MI+SW* and *MI+SW+HB*, and in the case of *MI+SW*, the large data movement overhead of update propagation. This demonstrates that we *cannot* achieve the three desired properties of an HTAP system without co-designing software with hardware. Fourth, *MI+SW+HB* cannot mitigate data movement and contention on shared resources, even with 8× the main memory bandwidth. As a result, *MI+SW+HB* transactional throughput is 58.8% of that of *Ideal-Txn*. Fifth, *PIM-Only* significantly hurts transactional throughput (by 67.6% vs. *Ideal-Txn*), and performs 7.6% worse than *SI-SS*. This happens because transactional queries have cache-friendly memory access patterns [20–22], thus benefiting from a deep cache hierarchy, which is not present in *PIM-Only*.

Fig. 7 (right) shows the analytical throughput of the six DBMSs. We normalize the analytical throughput at each transaction count to a baseline where analytical queries are running alone on the system (*Base-Anl*). We make four observations. First, Polynesia *improves* analytical throughput over *Base-Anl* by 63.8%, on average across the three transaction counts. The performance benefits of Polynesia come from eliminating data movement, reducing the latency of memory accesses, and using custom logic for update propagation and consistency. Second, while *SI-MVCC* is the best software-only DBMS when considering transactional throughput, it *degrades* analytical throughput by 63.2% compared to *Base-Anl* due to its lack of workload-specific optimizations and pointer-chasing intensive consistency mechanism (MVCC; see §3.1). Third, *MI+SW+HB* improves analytical throughput by 41.2% over *MI+SW*. However, it still suffers an analytical throughput loss of 35.5% compared to *Base-Anl*, on average, even though *MI+SW+HB* has 8× the main memory bandwidth of *MI+SW*. Fourth, the analytical throughput of *PIM-Only* is 11.4% lower than that of *MI+SW+HB*, on average, as *PIM-Only* suffers from resource contention caused by co-running transactional and analytical queries.

We conclude that Polynesia's island-based hardware-software co-design leads to significant performance benefits compared to all evaluated software-only implementations. Averaged across all transaction counts in Fig. 7, Polynesia has both a higher transactional throughput (2.20× over *SI-SS*, 1.15× over *SI-MVCC*, and 1.94× over *MI+SW*; mean of 1.70×) *and* a higher analytical throughput (3.78× over *SI-SS*, 5.04× over *SI-MVCC*, and 2.76× over *MI+SW*; mean of 3.74×).

Real Workload Analysis. To model more complex queries, we evaluate Polynesia using a mixed workload from TPC-C [212] (for our transactional workload) and TPC-H [213] (for our analytical workload). TPC-C's schema includes nine relations (tables) that simulate an order processing application. We simulate two transaction types defined in TPC-C, *Payment* and *New order*, which together account for 88% of the TPC-C workload [163] and touch all nine tables defined by TPC-C. We vary the number of warehouses from 1 to 4, and we assume that our transactional workload includes an equal number of transactions from both *Payment* and *New order*. For TPC-H, we use three of its queries in our experiments, each of which displays different behavior and operates over six TPC-H tables (out of the eight total tables in the TPC-H schema). The six tables are *LINEITEM*, *PART*, *SUPPLIER*, *PARTSUPP*, *ORDERS*, and *NATION*, with a cardinality (i.e., number of rows) of 6M, 200K, 10K, 800K, 1.5M, and 25, respectively. The three TPC-H queries we evaluate are: (i) Query 1 (*Q1*), an aggregation-heavy query [214] that generates a pricing summary report over the *LINEITEM* table; (ii) Query 6 (*Q6*), a selection-heavy query [214] that computes a forecast revenue change over the *LINEITEM* table; and (iii) Query 9 (*Q9*), a join-heavy query [214] that measures the profit for a given product type over all six tables.

We evaluate the transactional and analytical throughput for Polynesia and for three baselines: (1) *SI-SS*, (2) *SI-MVCC*, (3) *MI+SW*. We find that, averaged across all warehouse counts, Polynesia has a higher throughput than all three baselines. More specifically, Polynesia achieves both a higher transactional throughput (2.31× over *SI-SS*, 1.19× over *SI-MVCC*, and 1.78× over *MI+SW*; mean of 1.76×) and a higher analytical throughput for *Q1* (2.84× over *SI-SS*, 4.12× over *SI-MVCC*, and 2.4× over *MI+SW*; mean of 3.04×), for *Q6* (3.41× over *SI-SS*, 4.85× over *SI-MVCC*, and 2.2× over *MI+SW*; mean of 3.48×), and for *Q9* (3.67× over *SI-SS*, 4.51× over *SI-MVCC*, and 1.95× over *MI+SW*; mean of 3.18×).

We conclude that Polynesia's ability to meet all three HTAP properties enables better transactional and analytical performance over all three evaluated state-of-the-art systems.

10.2. Effect of the Update Propagation Technique

Fig. 8 shows the transactional throughput for Polynesia's update propagation mechanism and *Multiple-Instance*, normalized to a multiple-instance baseline with zero cost for update propagation (*Ideal*). We assume that each thread of the analytical workload executes 128 queries, and vary both the number of transactional queries per thread and the transactional query update-to-read ratio. To isolate the impact of different update propagation mechanisms, we use a zero-cost consistency mechanism, and ensure that the level of interference between transactional and analytical threads remains the same for all mechanisms. We make two observations from Fig. 8.

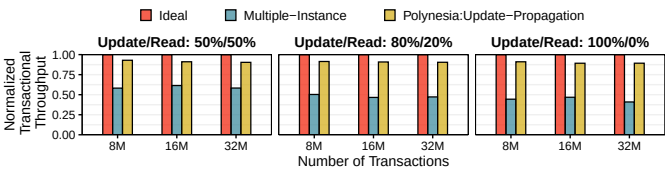

Figure 8: Effect of update propagation mechanisms on transactional throughput.

First, Polynesia's update propagation mechanism improves transactional throughput by 1.8× compared to *Multiple-*

Instance, and comes within 9.2% of *Ideal*, on average across all transaction counts and update-to-read ratios. The improvement comes from (1) significantly reducing data movement by offloading the update propagation process to PIM, (2) freeing up CPUs from performing update propagation by using a specialized hardware accelerator, and (3) co-designing the hardware and software for update propagation. Overall, our mechanism reduces the latency of update propagation by 1.9× compared to *Multiple-Instance* (not shown). Second, we find that *Multiple-Instance* degrades transactional throughput, on average, by 49.5% compared to *Ideal*, as it severely suffers from resource contention (e.g., at shared caches and main memory) and data movement cost. We observe that 27.7% of *Multiple-Instance*'s transactional throughput degradation comes from the update gathering and shipping latencies associated with (i) data movement and (ii) merging updates from multiple transactional threads. The remaining 21.8% transactional throughput degradation is due to the update application process, where the major bottlenecks are column compression/decompression and dictionary reconstruction.

We conclude that Polynesia's update propagation mechanism provides data freshness (i.e., low update latency) while maintaining high transactional throughput (i.e., performance isolation).

10.3. Effect of the Consistency Mechanism

Fig. 9 (left) shows the transactional throughput for Polynesia's consistency mechanism and Single-Instance-Snapshot (*Snapshot*), normalized to a single-instance baseline with zero-cost snapshotting (*Ideal-Snapshot*). Each thread performs 1M transactional queries and we vary the analytical query count. We make two observations. First, Polynesia improves transactional throughput by 2.2× over *Snapshot*, and comes within 6.1% of *Ideal-Snapshot*, on average, because it snapshots at a column granularity and leverages PIM for fast snapshotting. Second, *Snapshot* reduces transactional throughput, on average, by 59% compared to *Ideal-Snapshot*. This is because of expensive `memcpy` operations needed to create each snapshot, resulting in significant memory bandwidth contention.

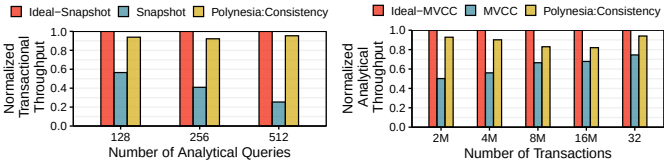

Figure 9: Effect of consistency mechanisms on transactional (left) and analytical (right) throughput.

Fig. 9 (right) shows the analytical throughput of Polynesia's consistency mechanism and of Single-Instance-MVCC (*MVCC*), normalized to a single-instance baseline with zero cost for MVCC (*Ideal-MVCC*, a hypothetical ideal baseline where MVCC operations incur zero delay during execution) for five transaction counts. We assume each thread of the analytical workload executes 128 queries, and we vary the transactional query count per analytical workload thread. For a fair comparison, we implement our consistency mechanism in a single-instance system. We make two observations. First, Polynesia's consistency mechanism improves analytical throughput by 1.4× compared to *MVCC*, and comes within 11.7% of *Ideal-MVCC*, on average, since Polynesia's consistency mechanism allows analytical queries to avoid scanning lengthy version chains when accessing each tuple. Second, *MVCC* degrades analytical throughput, on average, by 37.0% compared to *Ideal-MVCC*, as it forces each analytical query to traverse a lengthy version chain and perform expensive timestamp comparisons to locate the most recent version of the data.

We conclude that Polynesia's consistency mechanism maintains consistency without compromising performance isolation, leading to high analytical and transactional throughput.

10.4. Effect of the Analytical Engine

We study the effect of each of our data placement strategies from §7.1: (i) *Local*; (ii) *Distributed*; (iii) our *Hybrid* strategy, where all use the basic scheduler heuristic; and (iv) our hybrid strategy combined with our optimized task scheduler heuristic (labeled as *Hybrid-Sched*). For these studies, analytical queries address the same column.

Fig. 10 (left) shows the analytical throughput, normalized to a *CPU-Only* baseline where one core services all queries to the same column, for different number of analytical queries. We make four observations. First, *Local* reduces throughput by 23.9%, on average, over *CPU-Only*, because in *Local*, each analytical query can only use (1) the PIM cores in the local vault, which cannot issue enough memory requests concurrently to saturate the vault's memory bandwidth and exploit memory-level parallelism (MLP) [62, 76, 141, 215–222], and (2) a single vault's memory bandwidth. In contrast, *CPU-Only* leverages out-of-order cores that can issue many memory requests in parallel to multiple memory channels, exploiting MLP and higher memory bandwidth. Second, *Distributed* improves analytical throughput by 4.1×/3.1× over *Local*/*CPU-Only*, on average. This is because under *Distributed*, each column is partitioned across all vaults, allowing the analytical engines to service each analytical query using (1) all PIM cores, and (2) the entire internal memory bandwidth of the 3D-stacked memory. However, *Distributed* increases the update application latency, on average, by 45.8% (Fig. 10, right), and thus, degrades data freshness. This is because of the high update application costs (§7.1), which *Local* does *not* incur. Third, *Hybrid* addresses the shortcomings of *Local*, improving analytical throughput by 57.2% over *CPU-Only*, while having a similar update application latency (0.7 ms). This is because the local dictionary copies eliminate most of the remote accesses. However, the throughput under *Hybrid* is 49.8% lower than *Distributed*, because each query is serviced only using resources (memory bandwidth and PIM cores) available in the local vault group, leading to resource underutilization. Fourth, *Hybrid-Sched* overcomes *Hybrid*'s resource underutilization issues by enabling task stealing, which makes idle resources in remote vaults available for analytical queries. *Hybrid-Sched* comes within 3.2% of *Distributed*, while maintaining the same update application latency as *Hybrid*.

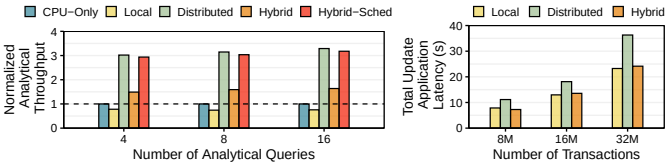

Figure 10: Effect of data placement and task scheduling on analytical throughput (left) and update application latency (right).

We conclude that our hybrid data placement along with our optimized task scheduling heuristic can provide high analytical throughput while fully leveraging the memory and computation resources of Polynesia's analytical island.

10.5. Effect of the Dataset Size

Fig. 11 (left) shows how Polynesia performs as the dataset size grows. To accommodate the larger data, we increase the number of memory stacks, doubling the dataset size as we double the stack count. We use a workload with 32M transactional and 60K analytical queries, and analyze analytical

throughput normalized to *Multiple-Instance*, as a case study. We assume stacks are connected together using a processor-centric topology [20, 63]. To provide a fair comparison, we double the number of cores available to the analytical threads in the *Multiple-Instance* baseline as we double the number of stacks, to compensate for the doubling of hardware resources available to Polynesia (since there are twice as many vaults).

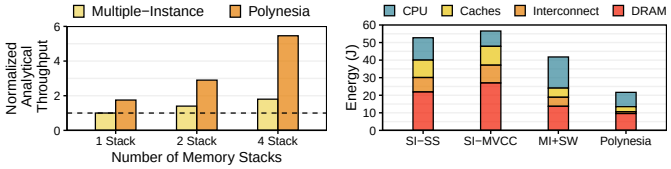

Figure 11: Number of memory stacks vs. analytical throughput (left). System energy (right).

We make two observations. First, Polynesia significantly outperforms *Multiple-Instance* (up to 3.0×) and scales well as we increase the stack count. This is because, as we increase the stack count, columns can be distributed more evenly across vault groups, which reduces the probability of multiple queries colliding in the same vault group. Second, with increasing dataset size, the overheads of consistency mechanism, update propagation and analytical query execution are all higher for *Multiple-Instance*, which hurts its scalability. The transactional throughput (not shown) decreases by 54.4% at four stacks for *Multiple-Instance*, compared to one stack, but decreases by *only* 8.8% for Polynesia.

We conclude that Polynesia scales well with the dataset size and thus the number of 3D-stacked memories used in the HTAP system. We believe that further performance can be achieved by levering interconnection topologies that favor memory-centric architectures [42, 51, 77, 223].

10.6. Energy Analysis

We model system energy similar to prior work [20, 22, 52, 224], which sums the energy consumed by the CPU cores, all caches (modeled using CACTI-P 6.5 [225]), DRAM, and all on-chip and off-chip interconnects. Fig. 11 (right) shows the total system energy across three HTAP DBMSs. We make two observations. First, Polynesia consumes only 0.41×/0.38×/0.51× the energy of *SI-SS*/*SI-MVCC*/*MI+SW*. Polynesia *significantly* reduces energy consumption compared to the three HTAP DBMSs since it eliminates a large fraction (30%) of off-chip accesses and uses custom logic and simple in-order PIM cores for its analytical islands. Second, the energy consumption of *MI+SW* is 0.8× and 0.7× that of *SI-SS* and *SI-MVCC*. However, *MI+SW* still consumes more energy than Polynesia since it (i) cannot reduce the large number of memory accesses to off-chip memory, and (ii) uses large and power-hungry CPU cores and caches. Such sources of energy consumption cannot be eliminated by simply providing high memory bandwidth to CPU cores. We conclude that Polynesia is an energy-efficient HTAP DBMS.

10.7. Area Analysis

We use previously reported data [195] to determine the area of our PIM cores (used by the analytical islands), and Calypto Catapult [226] to determine the area of our ASIC components, i.e, the update propagation unit and the copy unit (used by our consistency mechanism) for a 22 nm process. Four PIM cores require 1.8 mm^2, based on ARM Cortex-A7 (0.45 mm^2 each) [195]. The area Calypto Catapult reports for our ASIC components is 0.7 mm^2 for the update propagation unit and 0.2 mm^2 for the copy unit for our consistency mechanism. This brings Polynesia's total hardware area to 2.7 mm^2 per vault.

We conclude that our design can fit completely within the unused area in the logic layer of 3D-stacked memory (4.4 mm^2 per vault [52, 75, 76]). As a comparison, a 4-core Intel i5 processor with a 6 MB L3 cache has a die area of 160 mm^2 [227].

11. Related Work

To our knowledge, this is the first work that (1) proposes specialized hardware–software co-designed accelerators to cater for heterogeneous workload demands in HTAP systems, (2) describes an HTAP system that meets all three desired HTAP properties, and (3) uses processing-in-memory to alleviate data movement overheads in HTAP systems. We briefly summarize related works.

HTAP Systems. Several works from industry (e.g., [5, 12, 118–120]) and academia (e.g., [17, 18, 115, 121–127, 228]) propose techniques to support HTAP. Many of them use a single-instance design [17, 115, 118, 121–123], while others are multiple-instance [18, 120, 229]. All of these proposals suffer from the drawbacks we highlight in §3, and none can fully meet the three desired HTAP properties (§2).

Analytical Query Acceleration. Various prior works focus solely on analytical workloads [76, 168, 176, 230–232]. Some of these works propose to use specialized on-chip accelerators [168, 230, 231] while others propose to use PIM to speed up analytical operators [76, 176, 232]. However, none of these works study the effect of data placement or task scheduling for the analytical workload in the context of PIM or HTAP systems.

Processing-in-Memory (PIM). Many works [22, 42, 44, 45, 47, 49–52, 54, 57, 58, 60, 63, 64, 69–75, 77–80, 92, 97, 98, 102–104, 107, 207, 233–240] add compute units to the logic layer of 3D-stacked memory to accelerate various workloads. None of these works are designed for HTAP systems, and are largely orthogonal. Prior PIM-based DBMS proposals [57, 76, 81, 176, 232, 241] solely focus on analytical workloads. None of them study the effect of data placement or task scheduling for the analytical workload in the context of PIM or HTAP systems.

12. Conclusion

We propose Polynesia, a novel HTAP system that makes use of workload-optimized transactional and analytical islands (i.e., co-designed hardware/software units) to enable real-time analytical queries without sacrificing throughput. Our analytical islands alleviate the data movement and workload interference overheads incurred in state-of-the-art HTAP systems, while ensuring that data replicas for analytical workloads are kept up-to-date with the most recent version of the transactional data replicas. Polynesia outperforms three state-of-the-art HTAP systems (with a 1.7×/3.7× higher transactional/analytical throughput on average), while consuming less energy (48% lower than the best) and meeting all three desired HTAP properties. We conclude that Polynesia is an effective and efficient architecture for HTAP systems. We hope that Polynesia inspires future research and development in hardware/software co-designed HTAP systems that take advantage of processing-in-memory.

Acknowledgments

We thank the anonymous reviewers of MICRO 2019/2020, ASPLOS 2021, and ICDE 2022 for feedback. We thank SAFARI Research Group members for valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts of our industrial partners, especially Google, Huawei, Intel, Microsoft, and VMware. This research was partially supported by the Semiconductor Research Corporation and the ETH Future Computing Laboratory.

References

[1] "Cisco Global Cloud Index: Forecast and Methodology, 2016-2021," 2016. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html

[2] S. Cao *et al.*, "TitAnt: Online Real-Time Transaction Fraud Detection in Ant Financial," arXiv:1906.07407 [cs.LG], 2019.

[3] X. Qiu *et al.*, "Real-Time Constrained Cycle Detection in Large Dynamic Graphs," *Proc. VLDB Endow.*, 2018.

[4] J. T. Quah and M. Sriganesh, "Real-Time Credit Card Fraud Detection Using Computational Intelligence," *Expert Systems with Applications*, 2008.

[5] P.-Å. Larson *et al.*, "Real-Time Analytical Processing with SQL Server," *PVLDB*, 2015.

[6] J. Ramnarayan *et al.*, "SnappyData: Streaming, Transactions, and Interactive Analytics in a Unified Engine," in *CIDR*, 2016.

[7] B. Sahay and J. Ranjan, "Real Time Business Intelligence in Supply Chain Analytics," *Information Management & Computer Security*, 2008.

[8] S. Chisholm, "Adopting Medical Technologies and Diagnostics Recommended by NICE: The Health Technologies Adoption Programme," *Annals of the Royal College of Surgeons of England*, 2014.

[9] V.-D. Ta *et al.*, "Big Data Stream Computing in Healthcare Real-Time Analytics," in *ICCCBDA*, 2016.

[10] R. Barber *et al.*, "WiSer: A Highly Available HTAP DBMS for IoT Applications," arxiv:1908.01908 [cs.DB], 2019.

[11] J. Zhou *et al.*, "Kunpeng: Parameter Server Based Distributed Learning Systems and Its Applications in Alibaba and Ant Financial," in *SIGKDD*, 2017.

[12] P.-A. Larson *et al.*, "Real-Time Analytical Processing with SQL Server," *Proc. VLDB Endow.*, 2015.

[13] D. Huang *et al.*, "TiDB: A Raft-Based HTAP Database," *Proc. VLDB Endow.*, 2020.

[14] M. Pezzini *et al.*, "Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation," 2013. [Online]. Available: https://www.gartner.com/en/documents/2657815

[15] V. Sikka *et al.*, "SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform," *Proc. VLDB Endow.*, 2013.

[16] J. Giceva and M. Sadoghi, "Hybrid OLTP and OLAP," in *Encyclopedia of Big Data Technologies*. Springer, Cham, 2018.

[17] J. Arulraj *et al.*, "Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads," in *SIGMOD*, 2016.

[18] D. Makreshanski *et al.*, "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications," in *SIGMOD*, 2017.

[19] F. Özcan *et al.*, "Hybrid Transactional/Analytical Processing: A Survey," in *SIGMOD*, 2017.

[20] A. Boroumand *et al.*, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," in *ISCA*, 2019.

[21] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2017.

[22] A. Boroumand, "Practical Mechanisms for Reducing Processor-Memory Data Movement in Modern Workloads," Ph.D. dissertation, Carnegie Mellon University, 2020.

[23] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," arXiv:2012.03112 [cs.AR], 2021.

[24] O. Mutlu *et al.*, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MICPRO*, 2019.

[25] S. Ghose *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, 2019.

[26] W. H. Kautz, "Cellular Logic-in-Memory Arrays," *IEEE TC*, 1969.

[27] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.

[28] P. M. Kogge, "EXECUBE: A New Architecture for Scaleable MPPs," in *ICPP*, 1994.

[29] M. Gokhale *et al.*, "Processing in Memory: The Terasys Massively Parallel PIM Array," *Computer*, 1995.

[30] D. Patterson *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, 1997.

[31] M. Oskin *et al.*, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.

[32] J. Draper *et al.*, "The Architecture of the DIVA Processing-in-Memory Chip," in *ICS*, 2002.

[33] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.

[34] V. Seshadri *et al.*, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988 [cs:AR], 2016.

[35] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[36] V. Seshadri *et al.*, "RowClone: Accelerating Data Movement and Initialization Using DRAM," arXiv:1805.03502 [cs.AR], 2018.

[37] S. Angizi and D. Fan, "GraphidDe: A Graph Processing Accelerator Leveraging In-DRAM-Computing," in *GLSVLSI*, 2019.

[38] J. Kim *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.

[39] J. Kim *et al.*, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.

[40] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

[41] N. Hajinazar *et al.*, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.

[42] S. H. S. Rezaei *et al.*, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *CAL*, 2020.

[43] Y. Wang *et al.*, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.

[44] C. Giannoula *et al.*, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.

[45] I. Fernandez *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *ICCD*, 2020.

[46] M. Alser *et al.*, "Accelerating Genome Analysis: A Primer on an Ongoing Journey," *IEEE Micro*, 2020.

[47] D. S. Cali *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.

[48] J. S. Kim *et al.*, "GRIM-Filter: Fast Seed Filtering in Read Mapping Using Emerging Memory Technologies," arXiv:1708.04329 [q-bio.GN], 2017.

[49] J. S. Kim *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.

[50] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[51] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[52] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.

[53] A. Boroumand *et al.*, "LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures," arXiv:1706.03162 [cs:AR], 2017.

[54] G. Singh *et al.*, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.

[55] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.

[56] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.

[57] M. Gao *et al.*, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.

[58] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.

[59] B. Gu *et al.*, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *ISCA*, 2016.

[60] Q. Guo *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.

[61] M. Hashemi *et al.*, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.

[62] M. Hashemi *et al.*, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.

[63] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[64] D. Kim *et al.*, "NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.

[65] G. Kim *et al.*, "Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs," in *SC*, 2017.

[66] J. H. Lee *et al.*, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *PACT*, 2015.

[67] Z. Liu *et al.*, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.

[68] A. Morad *et al.*, "GP-SIMD Processing-in-Memory," *ACM TACO*, 2015.

[69] L. Nai *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.

[70] A. Pattnaik *et al.*, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.

[71] S. H. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.

[72] D. P. Zhang *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.

[73] Q. Zhu *et al.*, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.

[74] B. Akin *et al.*, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.

[75] M. Gao *et al.*, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[76] M. Drumond *et al.*, "The Mondrian Data Engine," in *ISCA*, 2017.

[77] G. Dai *et al.*, "GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing," *IEEE TCAD*, 2018.

[78] M. Zhang *et al.*, "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.

[79] Y. Huang *et al.*, "A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing," in *IPDPS*, 2020.

[80] Y. Zhuo *et al.*, "GraphQ: Scalable PIM-Based Graph Processing," in *MICRO*, 2019.

[81] P. C. Santos *et al.*, "Operand Size Reconfiguration for Big Data Processing in Memory," in *DATE*, 2017.

[82] S. Ghose *et al.*, "A Workload and Programming Ease Driven Perspective of Processing-in-Memory," arXiv:1907.12947 [cs:AR], 2019.

[83] M. Besta *et al.*, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO*, 2021.

[84] J. D. Ferreira *et al.*, "pLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation," arXiv:2104.07699 [cs.AR], 2021.

[85] A. Olgun *et al.*, "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAMs," in *ISCA*, 2021.

[86] S. Lloyd and M. Gokhale, "In-Memory Data Rearrangement for Irregular, Data-Intensive Computing," *Computer*, 2015.

[87] D. G. Elliott *et al.*, "Computational RAM: Implementing Processors in Memory," *IEEE Design & Test of Computers*, 1999.

[88] J. Landgraf *et al.*, "Combining Emulation and Simulation to Evaluate a Near Memory Key/Value Lookup Accelerator," 2021.

[89] A. Rodrigues *et al.*, "Towards a Scatter-Gather Architecture: Hardware and Software Issues," in *MEMSYS*, 2019.

[90] S. Lloyd and M. Gokhale, "Near Memory Key/Value Lookup Acceleration," in *MEMSYS*, 2017.

[91] M. Gokhale *et al.*, "Near Memory Data Structure Rearrangement," in *MEMSYS*, 2015.

[92] R. Nair *et al.*, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM JRD*, 2015.

[93] O. Mutlu and J. Gómez-Luna, "Exploring the Processing-in-Memory Paradigm for Future Computing Systems (Fall 2021)," https://safari.ethz.ch/projects_and_seminars/fall2021/doku.php?id=processing_in_memory.

[94] A. Olgun *et al.*, "PiDRAM: A Holistic End-to-End FPGA-Based Framework for Processing-in-DRAM," arXiv:2111.00082 [cs.AR], 2021.

[95] S. Lee *et al.*, "A 1ynm 1.25V 8Gb, 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, 2022.

[96] L. Ke *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2021.

[97] Y.-C. Kwon *et al.*, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.
[98] S. Lee *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021.
[99] J. M. Herruzo *et al.*, "Enabling Fast and Energy-Efficient FM-Index Exact Matching Using Processing-Near-Memory," *The Journal of Supercomputing*, 2021.
[100] G. Singh *et al.*, "FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications," *IEEE Micro*, 2021.
[101] G. Singh *et al.*, "Accelerating Weather Prediction Using Near-Memory Reconfigurable Fabric," *ACM TRETS*, 2021.
[102] G. F. Oliveira *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, 2021.
[103] A. Boroumand *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," in *PACT*, 2021.
[104] A. Boroumand *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," arXiv:2109.14320 [cs.AR], 2021.
[105] A. Denzler *et al.*, "Casper: Accelerating Stencil Computation Using Near-Cache Processing," arXiv:2112.14216 [cs.AR], 2021.
[106] N. M. Ghiasi *et al.*, "GenStore: A High-Performance and Energy-Efficient In-Storage Computing System for Genome Sequence Analysis," in *ASPLOS*, 2022.
[107] G. F. Oliveira *et al.*, "NIM: An HMC-Based Machine for Neuron Computation," in *ARC*, 2017.
[108] D. Niu *et al.*, "184QPS/W 64Mb/mm^2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.
[109] F. Devaux, "The True Processing in Memory Accelerator," in *HCS*, 2019.
[110] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," arXiv:2105.03814 [cs.AR], 2021.
[111] J. Gómez-Luna *et al.*, "Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-in-Memory Hardware," in *CUT*, 2021.
[112] Mythic, Inc., "Mythic — Power-Efficient AI Acceleration for the Edge," https://www.mythic.ai/.
[113] Hybrid Memory Cube Consortium, "HMC Specification 2.0," 2014.
[114] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.
[115] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots," in *ICDE*, 2011.
[116] A. Sharma *et al.*, "Accelerating Analytical Processing in MVCC Using Fine-Granular High-Frequency Virtual Snapshotting," in *SIGMOD*, 2018.
[117] SAFARI Research Group, "Polynesia — GitHub Repository," https://github.com/CMU-SAFARI/Polynesia/, 2022.
[118] F. Färber *et al.*, "The SAP HANA Database – An Architecture Overview," *IEEE Data Eng. Bull.*, 2012.
[119] T. Lahiri *et al.*, "Oracle Database In-Memory: A Dual Format In-Memory Database," in *ICDE*, 2015.
[120] A. K. Goel *et al.*, "Towards Scalable Real-Time Analytics: An Architecture for Scale-Out of OLxP Workloads," *Proc. VLDB Endow.*, 2015.
[121] M. Grund *et al.*, "HYRISE: A Main Memory Hybrid Storage Engine," *Proc. VLDB Endow.*, 2010.
[122] R. Appuswamy *et al.*, "The Case For Heterogeneous HTAP," in *CIDR*, 2017.
[123] M. Sadoghi *et al.*, "L-Store: A Real-Time OLTP and OLAP System," in *EDBT*, 2018.
[124] T. Mühlbauer *et al.*, "ScyPer: A Hybrid OLTP and OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics," in *BTW*, 2013.
[125] V. Arora *et al.*, "Janus: A Hybrid Scalable Multi-Representation Cloud Datastore," *IEEE Transactions on Knowledge and Data Engineering*, 2018.
[126] H. Kimura *et al.*, "Janus: Transaction Processing of Navigation and Analytic Graph Queries on Many-Core Servers," in *CIDR*, 2017.
[127] J. Lee *et al.*, "Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads," *PVLDB*, 2017.
[128] E. F. Codd, *The Relational Model for Database Management: Version 2*. Addison-Wesley, 1990.
[129] K. P. Eswaran *et al.*, "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM*, 1976.
[130] F. Chirigati *et al.*, "Virtual Lightweight Snapshots for Consistent Analytics in NoSQL Stores," in *ICDE*, 2016.
[131] T. Neumann *et al.*, "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems," in *SIGMOD*, 2015.
[132] H. Berenson *et al.*, "A Critique of ANSI SQL Isolation Levels," *SIGMOD Rec.*, 1995.
[133] M. J. Cahill *et al.*, "Serializable Isolation for Snapshot Databases," in *SIGMOD*, 2008.
[134] D. Šidlauskas *et al.*, "A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-Intensive Workloads," in *DaMoN*, 2012.
[135] R. Kallman *et al.*, "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," *PVLDB*, 2008.
[136] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model," *SIGMOD*, 1985.
[137] M. Stonebraker *et al.*, "C-Store: A Column-oriented DBMS," in *VLDB*, 2005.
[138] L. Subramanian *et al.*, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
[139] R. Das *et al.*, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.
[140] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[141] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[142] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
[143] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.
[144] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[145] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
[146] O. Mutlu, "Main Memory Scaling: Challenges and Solution Directions," in *More Than Moore Technologies for Next Generation Computer Design*. Springer-Verlag, 2015.
[147] L. Subramanian *et al.*, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
[148] L. Subramanian, "Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management," Ph.D. dissertation, Carnegie Mellon University, 2015.
[149] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[150] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[151] T. M. O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[152] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems With Hardware Accelerators," *TACO*, 2016.
[153] L. Subramanian *et al.*, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.
[154] D. Lee *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[155] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[156] B. Grot *et al.*, "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees," in *ISCA*, 2011.
[157] K. J. Nesbit *et al.*, "Fair Queuing Memory Systems," in *MICRO*, 2006.
[158] K. J. Nesbit *et al.*, "Multicore Resource Management," *IEEE Micro*, 2008.
[159] P. A. Bernstein and N. Goodman, "Multiversion Concurrency Control—Theory and Algorithms," *ACM Trans. Database Syst.*, 1983.
[160] I. Psaroudakis *et al.*, "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement," *VLDB Endowment*, 2015.
[161] C. Binnig *et al.*, "Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores," in *SIGMOD*, 2009.
[162] C. Lemke *et al.*, "Speeding Up Queries in Column Stores: A Case for Compression," in *DaWak*, 2010.
[163] X. Yu *et al.*, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," *VLDB Endow.*, 2014.
[164] V. Leis *et al.*, "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age," in *SIGMOD*, 2014.
[165] J. Krueger *et al.*, "Fast Updates on Read-Optimized Databases Using Multi-Core CPUs," *Proc. VLDB Endow.*, 2011.
[166] W. Fang *et al.*, "Database Compression on Graphics Processors," *PVLDB*, 2010.
[167] S. H. Pugsley *et al.*, "Fixed-Function Hardware Sorting Accelerators for Near Data MapReduce Execution," in *ICCD*, 2015.
[168] L. Wu *et al.*, "Q100: The Architecture and Design of a Database Processing Unit," in *ASPLOS*, 2014.
[169] D. J. Abadi *et al.*, "Materialization Strategies in a Column-Oriented DBMS," in *ICDE*, 2007.
[170] H. Mao, "Hardware Acceleration for Memory to Memory Copies," Master's thesis, Univ. of California, Berkeley, 2017.
[171] F. Gao *et al.*, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.
[172] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," arXiv:1905.09822 [cs.AR], 2019.
[173] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *SIGMOD*, 1990.
[174] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proc. VLDB Endow.*, 2011.
[175] S. Zeuch and J.-C. Freytag, "QTM: Modelling Query Execution With Tasks," *Proc. VLDB Endow.*, 2014.
[176] S. L. Xi *et al.*, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.
[177] JEDEC Solid State Technology Assn., "JESD235B: High Bandwidth Memory (HBM) DRAM," 2018.
[178] E. Azarkhish *et al.*, "A Logic-Base Interconnect for Supporting Near Memory Computation in the Hybrid Memory Cube," in *WoNDP*, 2014.
[179] R. Hadidi *et al.*, "Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube," in *ISPASS*, 2018.
[180] M. Poremba *et al.*, "There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes," in *ISCA*, 2017.
[181] E. Azarkhish *et al.*, "Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube," *IEEE VLSI*, 2016.
[182] E. Azarkhish *et al.*, "High Performance AXI-4.0 Based Interconnect for Extensible Smart Memory Cubes," in *DATE*, 2015.
[183] D. L. Eager *et al.*, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation*, 1986.
[184] Google LLC, "Cloud TPU," https://cloud.google.com/tpu, 2022.
[185] N. P. Jouppi *et al.*, "Ten Lessons from Three Generations Shaped Google's TPUv4i: Industrial Product," in *ISCA*, 2021.
[186] E. Medina and E. Dagan, "Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet with Gaudi Processor," *IEEE Micro*, 2020.
[187] Amazon.com, Inc., "AWS Trainium - Amazon Web Services (AWS)," https://aws.amazon.com/machine-learning/trainium/, 2022.
[188] P. Ranganathan *et al.*, "Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild," in *ASPLOS*, 2021.
[189] Microsoft Corp., "Improved Cloud Service Performance Through ASIC Acceleration," https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/, 2022.
[190] J. H. Kim *et al.*, "Aquabolt-XL: Samsung HBM2-PIM with In-Memory Processing for ML Accelerators and Beyond," in *HCS*, 2021.
[191] I. Magaki *et al.*, "ASIC Clouds: Specializing the Datacenter," in *ISCA*, 2016.
[192] X. Yu, "DBx1000," https://github.com/yxymit/DBx1000.
[193] N. Binkert *et al.*, "The gem5 Simulator," *Comp. Arch. News*, 2011.
[194] DRAMSim2, http://www.eng.umd.edu/ blj/dramsim/.
[195] Arm Ltd., "ARM Cortex-A7," https://developer.arm.com/ip-products/processors/cortex-a/cortex-a7.

[196] J. Picorel *et al.*, "Near-Memory Address Translation," in *PACT*, 2017.
[197] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *ISCA*, 1984.
[198] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Comput.*, 1978.
[199] J. Zhang *et al.*, "MEG: A RISCV-Based System Emulation Infrastructure for Near-Data Processing Using FPGAs and High-Bandwidth Memory," *TRETS*, 2020.
[200] I. Psaroudakis *et al.*, "Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads," *ADMS 2013*, 2013.
[201] I. Psaroudakis *et al.*, "Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling," *TPCTC*, 2014.
[202] A. Raza *et al.*, "Adaptive HTAP Through Elastic Resource Scheduling," in *SIGMOD*, 2020.
[203] C. Diaconu *et al.*, "Hekaton: SQL Server's Memory-Optimized OLTP Engine," in *SIGMOD*, 2013.
[204] X. Yu *et al.*, "TicToc: Time Traveling Optimistic Concurrency Control," in *SIGMOD*, 2016.
[205] Y. Xia *et al.*, "Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems," *Proc. VLDB Endow.*, 2020.
[206] X. Yu *et al.*, "Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System," *Proc. VLDB Endow.*, 2018.
[207] K. Hsieh *et al.*, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[208] G. Graefe and W. J. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," in *ICDE*, 1993.
[209] P. A. Boncz *et al.*, "MonetDB/X100: Hyper-Pipelining Query Execution," in *CIDR*, 2005.
[210] T. Neumann and V. Leis, "Compiling Database Queries into Machine Code," *IEEE Data Eng. Bull.*, 2014.
[211] J. Giceva, "Lecture Notes for Data Processing On Modern Hardware – Lecture 3: Cache Awareness for Query Execution Models," https://db.in.tum.de/teaching/ss20/dataprocessingonmodernhardware/MH_3.pdf, 2020.
[212] Transaction Processing Council, "TPC-C Benchmark," 2010. [Online]. Available: http://www.tpc.org/tpcc/spec/tpcc_current.pdf
[213] Transaction Processing Council, "TPC-H Benchmark," 2021. [Online]. Available: http://www.tpc.org/tpch
[214] P. Boncz *et al.*, "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark," in *TPCTC*, 2013.
[215] A. Glew, "MLP Yes! ILP No!" in *ASPLOS WACI*, 1998.
[216] O. Mutlu *et al.*, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.
[217] M. K. Qureshi *et al.*, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.
[218] O. Mutlu *et al.*, "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," *IEEE Micro*, 2006.
[219] O. Mutlu *et al.*, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.
[220] Y. Chou *et al.*, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *ISCA*, 2004.
[221] J. Tuck *et al.*, "Scalable Cache Miss Handling for High Memory-Level Parallelism," in *MICRO*, 2006.
[222] C. J. Lee *et al.*, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
[223] G. Kim *et al.*, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *PACT*, 2013.
[224] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSIT*, 2012.
[225] N. Muralimanohar *et al.*, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.
[226] Mentor Graphics Corp., "Catapult High-Level Synthesis," https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.
[227] H. Wong, "A Comparison of Intel's 32nm and 22nm Core i5 CPUs: Power, Voltage, Temperature, and Frequency," 2012. [Online]. Available: http://blog.stuffedcow.net/2012/10/intel32nm-22nm-core-i5-comparison/
[228] U. Sirin *et al.*, "Performance Characterization of HTAP Workloads," in *ICDE*, 2021.
[229] Oracle Corp., "Oracle GoldenGate 12c: Real-Time Access to Real-Time Information," 2015.
[230] O. Kocberber *et al.*, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," in *MICRO*, 2013.
[231] L. Wu *et al.*, "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," in *ISCA*, 2013.
[232] N. Mirzadeh *et al.*, "Sort vs. Hash Join Revisited for Near-Memory Execution," in *ASBD*, 2007.
[233] C. Xie *et al.*, "Processing-in-Memory Enabled Graphics Processors for 3D Rendering," in *HPCA*, 2017.
[234] R. Hadidi *et al.*, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-in-Memory," *ACM TACO*, 2017.
[235] P.-A. Tsai *et al.*, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *MICRO*, 2018.
[236] P. Liu *et al.*, "3D-Stacked Many-Core Architecture for Biological Sequence Analysis Problems," *IJPP*, 2017.
[237] P. C. Santos *et al.*, "Processing in 3D Memories to Speed Up Operations on Complex Data Structures," in *DATE*, 2018.
[238] R. Balasubramonian *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, 2014.
[239] B. Akın *et al.*, "HAMLeT: Hardware Accelerated Memory Layout Transform within 3D-Stacked DRAM," in *HPEC*, 2014.
[240] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *ISCA*, 2008.
[241] D. G. Tomé *et al.*, "HIPE: HMC Instruction Predication Extension Applied on Database Processing," in *DATE*, 2018.