The Mondrian Data Engine

Mario Drumond Alexandros Daglis Nooshin Mirzadeh Dmitrii Ustiugov
Javier Picorel Babak Falsafi Boris Grot[1] Dionisios Pnevmatikatos[2]
EcoCloud, EPFL [1]University of Edinburgh [2]FORTH-ICS & ECE-TUC
firstname.lastname@epfl.ch,boris.grot@ed.ac.uk,pnevmati@ics.forth.gr

ABSTRACT

The increasing demand for extracting value out of ever-growing data poses an ongoing challenge to system designers, a task only made trickier by the end of Dennard scaling. As the performance density of traditional CPU-centric architectures stagnates, advancing compute capabilities necessitates novel architectural approaches. Near-memory processing (NMP) architectures are reemerging as promising candidates to improve computing efficiency through tight coupling of logic and memory. NMP architectures are especially fitting for data analytics, as they provide immense bandwidth to memory-resident data and dramatically reduce data movement, the main source of energy consumption.

Modern data analytics operators are optimized for CPU execution and hence rely on large caches and employ random memory accesses. In the context of NMP, such random accesses result in wasteful DRAM row buffer activations that account for a significant fraction of the total memory access energy. In addition, utilizing NMP's ample bandwidth with fine-grained random accesses requires complex hardware that cannot be accommodated under NMP's tight area and power constraints. Our thesis is that efficient NMP calls for an algorithm-hardware co-design that favors algorithms with sequential accesses to enable simple hardware that accesses memory in streams. We introduce an instance of such a co-designed NMP architecture for data analytics, the Mondrian Data Engine. Compared to a CPU-centric and a baseline NMP system, the Mondrian Data Engine improves the performance of basic data analytics operators by up to 49× and 5×, and efficiency by up to 28× and 5×, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Processors and memory architectures**; Single instruction, multiple data; • **Information systems** → *Main memory engines*; • **Hardware** → Die and wafer stacking;

KEYWORDS

Near-memory processing, sequential memory access, algorithm-hardware co-design

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080233

ACM Reference format:
Mario Drumond Alexandros Daglis Nooshin Mirzadeh Dmitrii Ustiugov Javier Picorel Babak Falsafi Boris Grot[1] Dionisios Pnevmatikatos[2] EcoCloud, EPFL [1]University of Edinburgh [2]FORTH-ICS & ECE-TUC . 2017. The Mondrian Data Engine. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080233

1 INTRODUCTION

Large-scale IT services are increasingly migrating from storage to memory because of intense data access demands [7, 17, 51]. Online services such as search and social connectivity require managing massive amounts of data while maintaining a low request tail latency. Similarly, analytic engines for business intelligence are increasingly memory resident to minimize query response time. The net result is that memory is taking center stage in server design as datacenter operators maximize memory capacity integrated per server to increase throughput per total cost of ownership [42, 53].

The slowdown in Dennard scaling has further pushed server designers toward efficient memory access [18, 30, 49]. A large spectrum of data services has moderate computational requirements [20, 36, 43] but large energy footprints primarily due to data movement [16]: the energy cost of fetching a word of data from off-chip DRAM is up to 6400× higher than operating on it [29]. To alleviate the cost of memory access, several memory vendors are now stacking several layers of DRAM on top of a thin layer of logic, enabling near-memory processing (NMP) [35, 46, 63]. The combination of ample internal bandwidth and a dramatic reduction in data movement makes NMP architectures an inherently better fit for in-memory data analytics than traditional CPU-centric architectures, triggering a recent NMP research wave [2, 3, 22, 23, 56, 57].

While improving over CPU-centric systems with NMP is trivial, designing an *efficient* NMP system for data analytics in terms of performance per watt is not as straightforward [19]. Efficiency implications stem from the optimization of data analytics algorithms for CPU platforms, making heavy use of random memory accesses and relying on the cache hierarchy for high performance. In contrast, the strength of NMP architectures is the immense memory bandwidth they offer through tight integration of logic and memory. While DRAM is designed for random memory accesses, such accesses are significantly costlier than sequential ones in terms of energy, stemming from DRAM row activations [65].

In addition to their energy premium, random memory accesses can also become a major performance obstacle. The logic layer of NMP devices can only host simple hardware, being severely area- and power-limited. Unfortunately, generating enough memory-level parallelism to saturate NMP's ample bandwidth using fine-grained

[2] This work was done while the author was at EPFL.

accesses requires non-trivial hardware resources. Random memory accesses result in underutilized bandwidth, leaving a significant performance headroom.

In this work, we show that achieving NMP efficiency requires an algorithm-hardware co-design to maximize the amount of sequential memory accesses and provide hardware capable of transforming these accesses into bandwidth utilization under NMP's tight area and power constraints. We take three steps in this direction: First, we identify that the algorithms for common data analytics operators that are preferred for CPU execution are not equally fitting for NMP. NMP favors algorithms that maximize sequential accesses, trading off algorithmic complexity for DRAM-friendly and predictable access patterns. Second, we observe that most data analytics operators feature a major data partitioning phase, where data is shuffled across memory partitions to improve locality in the following probe phase. Partitioning unavoidably results in random memory accesses, regardless of the choice of algorithm. However, we show that minimal hardware support that exploits a common characteristic in the data layout of analytics workloads can safely reorder memory accesses, thus complementing the algorithm's contribution in turning random accesses to sequential. Finally, we design simple hardware that can operate on data streams at the memory's peak bandwidth, while respecting the energy and power constraints of the NMP logic layer.

Building on our observations, we propose the *Mondrian Data Engine*, an NMP architecture that reduces power consumption by changing energy-hungry random memory access patterns to sequential DRAM-friendly ones and boosts performance through wide computation units performing data analytics operations on data streams at full memory bandwidth. We make the following contributions:

- Analyze common data analytics operators, demonstrating prevalence of fine-grain random memory accesses. In the context of NMP, these induce significant performance and energy overheads, preventing high utilization of the available DRAM bandwidth.
- Argue that NMP-friendly algorithms fundamentally differ from conventional CPU-centric ones, optimizing for sequential memory accesses rather than cache locality.
- Identify object permutability as a common attribute in the partitioning phase of key analytics operators, meaning that the location and access order of objects in memory can be safely rearranged to improve the efficiency of memory access patterns. We exploit permutability to coalesce random accesses that would go to disjoint locations in the original computation into a sequential stream.
- Show that NMP efficiency can be significantly improved through an algorithm-hardware co-design that trades off additional computations and memory accesses for increased memory access contiguity and uses simple hardware to operate on memory streams. Our Mondrian Data Engine improves the performance of basic data analytics operators by up to 49× and 5×, and efficiency by up to 28× and 5× as compared to a CPU-centric and a baseline NMP system, respectively.

The paper is organized as follows: §2 provides an overview of data operators and their main phases. We briefly argue for NMP architectures as a good fit for data analytics and analyze their main inefficiencies in §3, which we then address in §4. We present the Mondrian Data Engine in §5, our methodology in §6, and evaluation in §7. Finally, we discuss related work in §8 and conclude in §9.

Basic operator	Spark operator
Scan	Filter, Union, LookupKey, Map, FlatMap, MapValues
Group by	GroupByKey, Cogroup, ReduceByKey, Reduce, CountByKey, AggregateByKey
Join	Join
Sort	SortByKey

Table 1: Characterization of Spark operators.

2 IN-MEMORY DATA OPERATORS

Large-scale IT services often rely on deep software stacks for manipulating and analyzing large data volumes. To enable both iterative and interactive data querying, the data management backend of such services increasingly runs directly in memory to avoid the bandwidth and latency bottleneck of disk I/O [12, 32, 45, 62]. While at higher levels data is queried in a specialized language (e.g., SQL), the software stack eventually converts the queries into data transformations. Much as in conventional database management systems, these transformations rely on a set of physical data operators at their core to plow through data [60]. Table 1 contains a broad set of common data transformations in Spark [21] and the basic data operators needed to implement them in memory: *Scan, Group by, Join,* and *Sort.*

Contemporary analytics software is typically column-oriented [1, 12, 21, 40, 50, 71], storing datasets as collections of individual columns, each containing a key and an attribute of a particular type. A dataset is distributed across memory modules where the operators can run in parallel. The simplest form of transformation is a *Scan* operator where each dataset subset is sequentially scanned for a particular key. More complex transformations require comparing keys among one or more subsets of the dataset.

To execute the comparison efficiently (i.e., maximize parallelism and locality), modern database systems partition datasets based on key ranges [68]. For instance, parallelizing a *Join* would require first a range partitioning of a dataset based on the data items' keys to divide them among the memory modules, followed by an independent *Join* operation on each subset. A similar data item distribution is required for the *Group by* and *Sort* operators.

Table 2 compares and contrasts the algorithmic steps in the basic data operators. All operators can be broken down into two main phases. Starting from data that is initially randomly distributed across multiple memory partitions, the goal of the first phase, *partitioning,* is to redistribute data so that memory access locality will be maximized in the upcoming operations. The partitioning phase involves data distribution to new partitions, usually based on a hash value of their key, which results in mostly random memory accesses across memory partitions. Prior work on join operators showed that partitioning accounts for roughly half of the total runtime [11, 38, 68], and may completely dominate execution time ($> 90\%$) in highly optimized implementations [10].

The second phase, *probe,* leverages the data locality created in the first phase, by probing data exclusively located in a single partition and applying the desired operation. In some cases, such as in *Join* and *Group by*, a second hash step may be applied before the actual data probe, to facilitate data finding during the upcoming computations. Even though all data accesses in the probe phase are localized in the given data partition, the access pattern is often random. As a

Operators / Phases	Partitioning		Probe	
	Histogram build	Data distribution	Hash table build	Operation
Scan	-	-	-	Scan keys
Join	Hash keys with low order bits	Copy to partitions	Hash keys & reorder	Join by key
Group by				Group by key
Sort	Hash keys with high order bits		-	Local sort

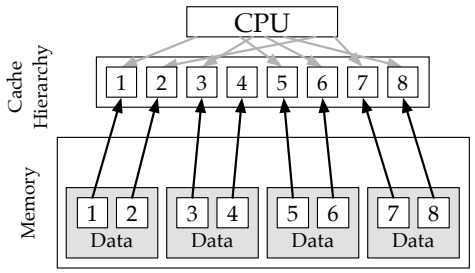
Table 2: Phases of basic data operators.

result, NMP architectures may struggle to fully utilize their available memory bandwidth. Generating high enough bandwidth demand with random accesses requires tens of outstanding requests, more than a conventional CPU typically supports. In contrast, sequential accesses allow bandwidth utilization with simple hardware, which is a critical target for the severely area- and power-constrained NMP logic. Therefore, algorithms that favor sequential accesses may be preferable to others that are optimized for locality and/or have a better algorithmic complexity.

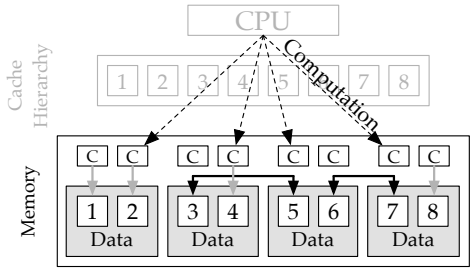
3 CHALLENGES IN DATA ANALYTICS

Performance constraints have made in-memory data analytics a common practice. While moving from disk-based to memory resident data services highlights the central role of data, traditional CPU-centric platforms for data analytics are still built around compute rather than data. CPU-centric architectures are advocates of centralization, as they require transferring all data from the memory close to the processor, the ingress point where all computations take place, as shown in Fig. 1a. Data lies across several memory partitions, but such distribution has little effect: the data transfer cost from the memory close to the CPU may be somewhat sensitive to the location of the accessed memory partition (e.g. NUMA effects), but is high in any case. CPUs rely on high locality and reuse in their cache hierarchy for efficient operation, amortizing the exorbitant cost of moving data from memory over multiple fast and cheap cache accesses. Unfortunately, this requirement makes CPUs and large-scale analytics ill-matched, as the latter exhibit either no temporal reuse or reuse at intervals beyond the reach of practical on-chip caches [20, 64].

Near-Memory Processing (NMP) is an alternative architectural approach to data analytics, being data-centric instead of compute-centric. Fig. 1b illustrates the main concept of such architectures. Instead of moving data from memory to compute, compute elements are physically placed closer to the target data, near the memory, and the CPU offloads computation to these NMP units, which then independently access memory to perform the requested task. As in the case of CPU-centric architectures, data is distributed across multiple physical memory partitions; but, unlike CPUs, the accessed memory partition affects each NMP compute element's memory access efficiency: accessing a local memory partition is significantly faster and more efficient than a remote one. Hence, simply pushing compute closer to the memory does not entirely solve the data movement bottleneck, as many operators rely on scatter/gather operations that may require accessing one or more remote partitions. Therefore, in the partitioning phase of data operators, data is shuffled across the memory partitions (black arrows in Fig. 1b) to maximize memory access locality [48, 57]. After partitioning, the NMP compute units enjoy low-latency/high-bandwidth access to their local memory partition, avoiding wasteful data movement across the memory network.



(a) CPU-centric architecture. Crossing arrows: random accesses.



(b) NMP architecture. "C" represents an NMP core.

Figure 1: Data access on two different architectures. Black arrows indicate expensive data movements.

Overall, the data-centric nature of data analytics makes NMP architectures a great fit for data analytics. However, NMP is rather an enabler than a solution. The memory access patterns (§3.1) and the NMP logic's ability to utilize the available memory bandwidth under tight area and power constraints (§3.2) are critical to an NMP system's efficiency.

3.1 Memory access patterns

DRAM-based memory has its idiosyncrasies regarding access patterns, which determine its efficiency. In general, maximizing DRAM's efficiency and effective bandwidth requires sequential accesses instead of random.

DRAM is organized as an array of rows, and a DRAM access comprises two actions: a row activation and the transfer of the requested data on the I/O bus. A row activation involves copying the entire DRAM row containing the requested data to a *row buffer*, regardless of the memory access request's size. The row buffer can then serve requests coming from the upper level of the memory hierarchy (typically cache-block-sized). For DDR3, the row activation energy is 3× higher than the energy of transferring a cache block from/to the row buffer [47]. Because of this energy breakdown, energy efficiency requires amortization of the row activation cost,

by leveraging row buffer locality (i.e., accessing all or most of the opened DRAM row's data).

While the memory access energy breakdown is different for die-stacked memory technologies, the trend remains similar [4]. Taking Micron's HMC as an example, the effective row buffer size is 32× smaller than DDR3 (256B vs. 8×1KB). It also supports transfers smaller than a typical 64B cache block (8 or 16B), as NMP compute units may directly access the memory without an intermediate cache hierarchy. Furthermore, the transfer distance in die-stacked memory is in the order of μm, as compared to mm in the case of DDR3. We use CACTI-3DD [14] to estimate the row activation energy overhead in HMC. While in the case a whole row is accessed it only accounts for 14% of the access' energy, it quickly climbs to 80% when only accessing 8B of data, a common occurrence for applications that are dominated by random memory accesses. HMC is a conservative example, as this energy gap is sensitive to the row buffer size, which is even larger in other die-stacked memory devices (2KB in HBM [35] and 4KB in Wide I/O 2 [34]).

As NMP inherently reduces the energy spent on data movement, the memory access pattern (random vs. sequential) becomes a considerable factor of the memory subsystem's overall efficiency, as row activations account for a major fraction of DRAM's dynamic energy. It is therefore important for data services running on NMP systems to replace fine-grained random accesses with bulk sequential ones.

3.2 Memory-level parallelism

NMP architectures expose immense memory bandwidth to the logic layer. However, the hardware resources that can be placed on the logic layer are tightly area- and power-constrained. Utilizing the available memory bandwidth requires the logic layer to generate enough memory-level parallelism (MLP). Doing so with fine-grained random accesses is challenging, as it requires hardware with a large instruction window and the capability of keeping many concurrent outstanding memory accesses alive at all times. Using streams of sequential accesses instead significantly relaxes the hardware requirements, by enabling utilization of the memory bandwidth with a small number of parallel outstanding memory access sequences.

To illustrate, a single HMC memory partition (vault) offers a peak memory bandwidth of 8GB/s, while the area and power budget of the vault's underlying logic layer is just $4.4mm^2$ and $312mW$, respectively [23, 33, 57]. These limitations preclude the usage of aggressive out-of-order cores as NMP units, which represent the best candidates for achieving high numbers of outstanding random memory accesses. To put numbers into perspective, an ARM Cortex-A57 core with an ROB of 128 entries would—very optimistically—be able to keep about 20 outstanding memory accesses, assuming one 8-byte memory access every 6 instructions and enough MSHRs to support that many misses. With a memory latency of 30ns, even in the ideal case where that MLP is maintained at all times, the memory bandwidth utilization would approach 5.3GB/s. While not a far cry from the peak of 8GB/s, the power of such an ARM Cortex-A57 at 1.8GHz and 20nm is rated at 1.5W, surpassing the NMP power cap by several factors. While prefetchers can help push the bandwidth utilization higher, if the application is dominated by random memory accesses, the prefetched data will rarely be useful or even detrimental to the cache's hit ratio [20]. Our experiments (§7) confirm these limitations, showing that even for operators as simple as *Scan*, the above bandwidth utilization calculation is too optimistic. Overall, conventional MIMD cores cannot sustain a high enough number of parallel fine-grained accesses to utilize the memory's full bandwidth under the tight constraints of an NMP architecture.

4 EFFICIENT MEMORY ACCESS

Addressing the two efficiency challenges in near-memory analytics presented in §3 requires the synergy of algorithms and hardware. We now discuss the required modifications in data analytics software (§4.1) and the NMP hardware characteristics necessary to utilize the available bandwidth under tight power and area constraints (§4.2).

4.1 Sequential accesses and row buffer locality

Memory access patterns are a direct consequence of the data structures and algorithms used at the application level. Modern software is optimized for execution on CPU-centric architectures, favoring cache-aware data accesses to minimize expensive memory access and maximize locality. These algorithms should be revisited in the face of near-memory processing, where efficiency is derived not by caching, but rather by proximity to memory. For NMP architectures, efficient algorithms are those that access memory sequentially, thus allowing simple hardware to exploit rich internal DRAM bandwidth.

4.1.1 Choice of algorithm. Going back to the basic operators (Table 2), each operation's probe phase can be completed by different algorithms. To illustrate, *Join* algorithms generally fall into two categories: sort- and hash-based. The former target efficiency through sequential memory accesses, albeit at a higher algorithmic complexity; the latter optimize for fast lookups and high cache locality. While the gradual increase of SIMD width in mainstream CPUs may eventually tip the balance in favor of sort-based join algorithms [38], hash-based joins are still preferred for CPU execution [8].

In the case of NMP architectures, two characteristics motivate revisiting the algorithms known to perform best for CPUs: significantly cheaper and faster memory access and a constrained logic capacity, which limits the number of in-flight random memory accesses. These features motivate the use of algorithms with simpler—sequential—access patterns, even at the cost of higher algorithmic complexity (i.e., more passes on the data). In the case of the *Join* operator, sort-merge join is an alternative to hash join: while it has a higher algorithmic complexity than hash join (*O(nlogn)* vs. *O(n)*), requiring sorting both relations prior to merge-joining them, it allows execution of the operator using solely sequential memory accesses. The same tradeoff applies to other data operators (e.g., *Group by*), as well: replacing hash-based approaches with—algorithmically more expensive—sort-based approaches, can significantly improve the efficiency of the operator's probe phase.

4.1.2 Data permutability. The access patterns of the operators' probe phase are largely determined by the used algorithm. However, that is not the case for the partitioning phase, which accounts for a major fraction of the operators' total runtime [10, 11, 38, 68]. During partitioning, each computation unit sequentially goes through its local memory partition and determines each data element's destination partition. As multiple partitions execute this data partitioning in parallel, writes arrive at every destination partition interleaved,

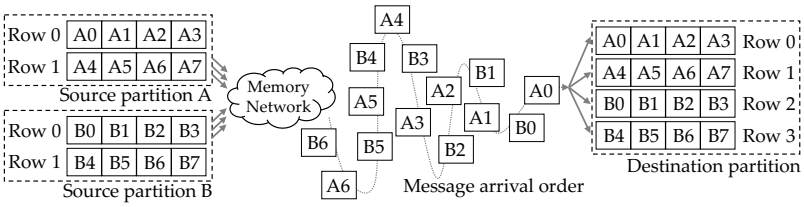


Figure 2: Partitioning phase: data shuffling across memory partitions.

resulting in random memory accesses. This random access pattern is inherent to such data shuffling and is an unavoidable result of memory message interleaving in the memory network, hence it cannot be controlled at the software/algorithmic level, as is the case for the data operators' probe phase.

Fig. 2 exemplifies a memory access pattern resulting from partitioning. For simplicity, we only show two source partitions, A and B, where all of their data has to end up in the same destination partition. As both source partitions send data, the messages get interleaved in the memory network, resulting in random writes at the destination: no two consecutive writes hit in the same DRAM row. In reality, multiple DRAM rows can be kept open at the same time, so even interleaved accesses can hit in an open row buffer. However, the probability of an access finding an open row quickly drops with the system size, as more sources shuffle data concurrently. Finally, even though memory controllers have to some extent the ability to reorder incoming requests and prioritize accesses to open rows, the distance of accesses to different locations within a row is typically too long for this scheduling window [65, 66]; maximizing row buffer locality requires more advanced techniques.

We observe that we can turn *all* random accesses during the partitioning phase of data analytics operators into sequential ones by exploiting a data structure property prevalent in data analytics with minimal hardware support in the memory controller. Our key insight is that the exact location of data objects within each memory partition is not important, because data operators apply the same computation on every data object in a memory region, much like an unordered data bucket. Memory partitions in the partitioning phase of data operators are commonly treated like hash table buckets. The order of items in a hash table bucket is not important, as it is always treated as a heap of data: finding something in a bucket requires going through all of its contents. Therefore, data objects in a hash table bucket—similarly, in a memory partition—are *permutable:* any order in the bucket is acceptable, and does not affect the operator's correctness. In Fig. 2's example, writing data objects sequentially into memory as they arrive would not affect correctness, as long as the software will only use that memory region as a sequence of unordered data objects during the upcoming probe phase. Such an approach avoids redundant row buffer activations by guaranteeing that every row buffer is activated *exactly once.*

The data object permutability insight is broadly applicable to a range of key data analytics operators. It holds for the partitioning phase of the basic *Join, Group by,* and *Sort* operators (Table 2) and also extends to more complex operators that build upon these three (e.g., see Table 1 for Spark operators). It also applies to the data partitioning and shuffling phase of MapReduce and any BSP-based graph processing algorithm, or, more generally, to any operation that involves coarse-grained data partitioning in buckets.

4.2 Streaming and SIMD

Revisiting the software-implemented algorithms to expose sequential accesses to hardware brings us half way to our goal of efficient bandwidth utilization; the remaining half requires designing hardware capable of leveraging these sequential accesses to execute data operators at full memory bandwidth. While the exact hardware parameters such as the available memory bandwidth or power and area limitations per compute unit differ across NMP architectures, the main limitations and challenges remain the same. In all cases, a compute unit for an NMP architecture has access to memory bandwidth several times higher than a CPU core, with much stricter area and power constraints. At the same time, NMP compute units should be programmable to be capable of executing a broad range of data analytics operators.

By relying on software to access memory sequentially, NMP units can be programmable processing units capable of fully utilizing their corresponding memory bandwidth by streaming data from memory, thus requiring a small amount of hardware state to sustain the necessary memory-level parallelism. However, computation has to keep up with the data streams in order to sustain peak memory bandwidth. As the vast majority of data analytics operators are data parallel, we observe that a modest degree of SIMD capabilities is in most cases sufficient to achieve that. SIMD extensions are already commonplace, even in low-power processors that are usable as NMP compute units (e.g., ARM Cortex-35 [6]).

5 THE MONDRIAN DATA ENGINE

We now present the Mondrian Data Engine, a novel NMP architecture for data analytics built on top of our algorithm-hardware co-design insights that efficiently utilizes the ample memory bandwidth that is available due to memory proximity.

5.1 Architecture Overview

The Mondrian Data Engine architecture consists of a network of NMP-capable devices. Fig. 3a shows the floorplan of such a network. Each device has a number of memory partitions, and each memory partition features a tightly coupled compute unit capable of operating on data independently. All the partitions are interconnected via both on-chip interconnects and an inter-device network, allowing all compute units to access any memory location in the NMP network. The NMP network is also attached to a CPU, which has a supervisory role: it does not actively participate in the evaluation

of the data operators, but initializes all components, launches and orchestrates the execution, and collects the final results. Without loss of generality, we assume a flat physical address space spanning across conventional planar DRAM and the NMP-capable devices. The address space covered by the NMP-capable devices is accessible by the CPU, and can be managed as a unified address space, similar to GPUs [31] and recent work on virtual memory support for NMP systems [55]. NMP compute units can only access the aggregate address space of the NMP-capable devices via physical addresses.

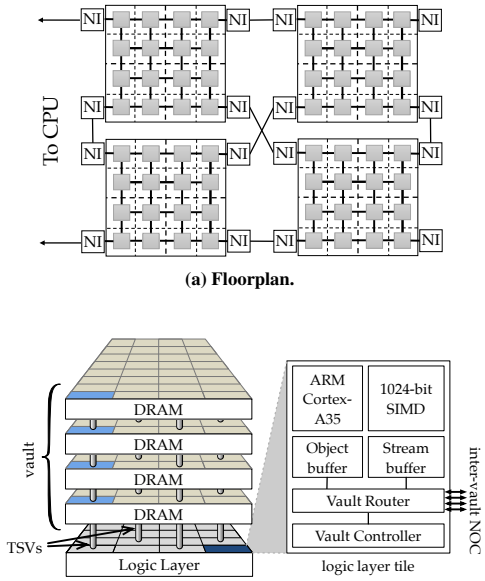
We design the Mondrian Data Engine's compute units to execute common data analytics operations at their local memory partition's peak effective bandwidth. We also apply minimal modifications to the memory controllers to exploit the data permutability opportunity inherent in data analytics. In the rest of this section, we discuss the Mondrian Data Engine's implementation and programming model.

5.2 Compute units

We use Micron's HMC [46] as our basic building block, which features a 3D stack of DRAM dies on top of a logic die, interconnected using TSV technology. Each die is segmented into 32 autonomous partitions. As shown in Fig. 3b, a vertical stack of the partitions forms a *vault*, with each such vault controlled by a dedicated controller residing on the logic die. Compared to conventional DRAM, which is organized into multi-KB rows, the HMC features smaller rows of 256B and a user-configurable access granularity of 8 to 256 bytes. To interface with the CPU, the HMC uses serial SerDes links running a packet-based protocol. Each link operates independently and can route memory requests crossing HMC boundaries. The actual DRAM control logic is distributed among the vault controllers.

Every HMC vault offers an effective peak memory bandwidth of 8GB/s with an underlying logic layer of $4.4mm^2$ and a peak power budget of $312mW$ per vault. As our first-order analysis in §3.2 already showed, using conventional MIMD cores to saturate the bandwidth under that area and power envelope is very challenging. In contrast, the 8GB/s bandwidth target can be easily met by streaming, as long as the software exposes sequential accesses to the hardware. We thus provision the logic layer with eight 384B (1.5× the row buffer size) stream buffers, sized to mask the DRAM access latency and avoid memory-access-related stalls. The stream buffers are programmable and are used to keep a constant stream of incoming data in the form of binding prefetches to feed the compute units. The next challenge lies in processing the incoming memory streams at a rate matching the memory bandwidth.

Going back to the basic data analytics operators, we identify that the most challenging operation on data streams is sorting. Since most algorithms that favor sequential accesses over random eventually require sorting data prior to the target operator's probe phase, providing hardware that can operate on data tuples at memory bandwidth is critical to performance. We base the following analysis on 8B/8B key/value tuples, as commonly done in data analytics research [10, 11]. This assumption does not limit our hardware's capabilities; in fact, the smaller the tuples, the more challenging it is for compute units to utilize the available bandwidth. We estimate that we can sort incoming streams of 16B tuples at the available 8GB/s per-vault memory bandwidth with an 8-tuple-wide SIMD unit; hence we require an 128B (1024-bit) SIMD unit. We identify mergesort



(a) Floorplan.

(b) HMC unit and logic tile.

Figure 3: Mondrian Data Engine architecture.

as the fittest near-memory sort algorithm, as it spends most of the time merging ordered streams of tuples, thus maximizing sequential memory accesses. Mergesort sorts n tuples by performing $logn$ sequential passes on the dataset (O(nlogn) algorithmic complexity). In order to match the available bandwidth, and assuming a frequency of 1GHz, we must process a tuple every 4 cycles; hence, with an 8-tuple-wide SIMD unit, we must process a group of 8 tuples every 32 cycles. We can trivially merge 8 streams of tuples into 4 sorted streams in a data parallel manner under this time budget. We further optimize mergesort with an initial bitonic sort pass, using the SIMD algorithm used in [8], where we sort small groups of tuples that are later merged (intra-stream sorting). This optimization reduces the required number of passes on the dataset by four, which in our system setup (512MB vault filled with 16B tuples) corresponds to a $\sim$ 20% reduction in the total number of passes.

We choose the ARM Cortex-A35, a dual-issue, in-order core, as our baseline NMP compute unit, dissipating 90mW at 1GHz in 28nm technology. The Cortex-A35 offers customizable features; we estimate that a single core with 8KB caches and a 128bit-wide SIMD unit has an area footprint of $0.65mm^2$, with less than $0.25mm^2$ attributed to the SIMD unit [27]. Given that data analytics do not require the unnecessary complexity of floating-point SIMD, we can extend the SIMD unit to 1024 bits for just 2× higher power than the original 128-bit Neon, as illustrated by Qualcomm's Hexagon 680 DSP [25]. This dramatic reduction in power is corroborated by DianNao [15], which shows that fixed point ALUs are about 8× more efficient that floating point ALUs. Assuming similar area and power scaling while increasing the SIMD width and replacing floating point with fixed point precision, we estimate the modified ARM Cortex-A35 to be $1.15mm^2$ and dissipate at most $180mW$, staying well within our per-vault area and power budget.

```
//allocate permutable regions per vault
for (vault_id = 0; vault_id < vault_num; vault_id++)
  perm_array = malloc_permutable(size, object_size,
                                 vault_list[vault_id]);
//enable memory permutability
shuffle_begin(perm_array);
....
shuffle_end(); //disable memory permutability
```

(a) Toggling permutability during partitioning phase.

```
//Configure prefetcher to prefetch up to 8 data streams
//Streams are allocated as follows:
//Stream 0: [start_addr, start_addr + stream_size)
//Stream 1: [start_addr + stream_size,
//                    start_addr + 2*stream_size)
// ....
prefetch_in_str_buf(start_addr, stream_size, num_streams);

while (!all_stream_buffer_done()) {
  tuples = read_stream_heads(tuple_size);

  //Perform some computation with the tuples [...]

  //Advance stream heads - notify prefetcher
  pop_input_stream(tuple_size);

  store(modified_tuples, output_addresses);
}
```

(b) Stream buffer management.

Figure 4: Mondrian Data Engine sample pseudocode.

5.3 Exploiting data permutability

We leverage the permutability property of data during the partitioning phase of data analytics to convert random memory accesses to sequential ones. Initially, when the CPU sets up an NMP execution, a per-vault base physical address and size of the destination buffer for the partitioning phase is sent to each vault controller by writing to a set of special memory-mapped registers. Every received write request marked as permutable is sequentially written in the destination buffer, thus maximizing row buffer locality. As the exact per-vault destination buffer size is only known after the data operator's histogram build step of the partitioning phase, the CPU only provides a best-effort overprovisioned estimation.

A subtle but critical aspect of the implementation is that the permutability feature holds on a *per-object* rather than a *per-memory-message* basis: if a single object is broken into multiple memory requests, these requests cannot be handled independently and treated as permutable. A solution to this implication is to always send write requests that contain the whole object. A vault controller only makes inter-request and never intra-request memory location permutations.

In order to prevent data objects from straddling more than a single memory message, we introduce special *object buffers*. During the initialization of the permutable memory regions, the software exposes the used object sizes (and hence the granularity of permutability) to the hardware. During the partitioning phase, the hardware drains the object buffer to the vault router only when its contents match the size of the specified object size, thus injecting an object-sized write request in the network. In our implementation, we use a single 256B object buffer per compute unit. The buffer size is determined by the row buffer size and is compliant with the HMC protocol, which allows messages up to 256B long to be exchanged between vault controllers. The object buffer size does preclude using data objects larger than 256B. However, beyond that size, the objects are big enough to exploit row buffer locality without being permuted by the destination vault controller.

5.4 Programming the Mondrian Data Engine

We provide two special functions to mark the code region of the partitioning phase, which generates permutable data object stores (Fig. 4a): *shuffle_begin* and *shuffle_end*. *Shuffle_begin* blocks execution until all participating vault controllers have been set up for the upcoming batching phase and involves two steps. First, every NMP unit computes the amount of data it is going to write to each remote vault, and then writes it to a predefined location of the corresponding remote vault. This information is collected during the *histogram build* step of the operators' partitioning phase (see Table 2), which is necessary in the baseline CPU system as well. After this information is exchanged, every NMP unit sums all these received values to compute the total size of inbound data. At this point, an NMP unit may identify that the incoming data will overflow the local destination buffer, which was allocated in advance by the CPU. In such a case, an exception may be raised for the CPU to handle, as the histogram build of the partitioning phase should be retried with a second round of partitioning in order to balance the resulting partitions' sizes. We focus on uniform data distributions where this behavior never arises and defer support for skewed datasets to future work.

All the stores within *shuffle_begin* and *shuffle_end* falling in the permutable memory region are treated by the receiving memory controllers as permutable. The Mondrian Data Engine does not bound the number of *permutable_store*s in flight during a partitioning phase, and does not enforce any ordering among them. However, a synchronization mechanism to identify the completion of all in-flight stores before execution continues to its next phase (probe) is necessary. The programmer marks this synchronization point in the code with a *shuffle_end*. When an NMP compute unit reaches that point, it suspends its execution until signaled to continue. We leverage message-signaled interrupts (MSI) as our notification mechanism. Every NMP accelerator has an interrupt vector with a width matching the total number of vaults, each bit representing a vault. When a vault controller finishes writing all expected data (as specified by *shuffle_begin*), it sends an MSI to all NMP units. When an NMP unit's interrupt controller finds all bits of the interrupt vector set, it wakes up its compute unit to resume execution. The same notification mechanism is used to advance into the partitioning phase after the completion of a *shuffle_begin*. While this all-to-all communication protocol can be expensive, it is only required at the beginning and end of a long partitioning phase, hence we did not attempt to optimize it.

Fig. 4b's pseudocode illustrates the Mondrian Data Engine's stream buffer management. We provide a programming interface that allows tying a data stream to a buffer and popping the head tuple of the stream to operate on. While the application consumes the stream buffers' head entries, stream buffers continue getting filled from memory, until the user-specified end of the stream is reached.

6 METHODOLOGY

System organization. We evaluate a collection of basic data analytics operators on three different architectures: CPU, NMP, and

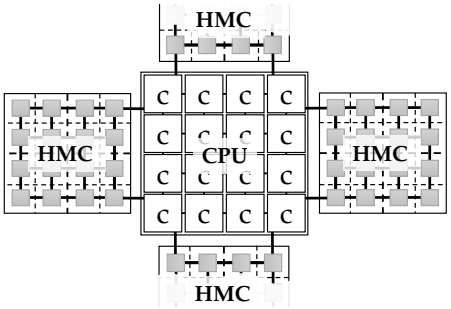


Figure 5: Modeled CPU-centric architecture.

Mondrian Data Engine. As platform setups for data analytics are memory driven, we use memory capacity as the normalization metric across the evaluated systems. Going after a 32GB system, we deploy four 8GB HMCs. Each modeled HMC features 16 512MB vaults (instead of 32 256MB vaults of the real device, because of simulation limitations), for a total capacity of 32GB in 64 vaults. Taking into account that a typical CPU setup for data analytics allocates 2-8GB of memory per core (Cloudera) [64], we populate our 32GB CPU system with 16 cores. We assume a tiled chip, connected to four passive HMCs in a star topology (Fig. 5). In the NMP systems, the HMCs are active and fully connected, as illustrated in Fig. 3a.

For the baseline NMP system we assume compute units that resemble the CPU-centric architecture's MIMD cores. We therefore use the best OoO core we can fit under the per-vault power budget: a 3-wide 1GHz core with a 48-entry ROB, similar to a Qualcomm Krait400 [26]. Both the CPU and the NMP baseline systems feature a next-line prefetcher, capable of issuing prefetches for up to three next cache lines.

Simulation. We use Flexus [67], a full-system cycle-accurate simulator, combined with DRAMSim2 [58] to evaluate and compare the three aforementioned systems. Table 3 summarizes the used parameters.

Evaluated operators. We evaluate four basic operators, namely *Scan, Join, Group by,* and *Sort*, which are representative of a broad range of data operators (e.g., see relation to Spark operators in Table 1). For the CPU, our operator implementations are adaptations of the *Join* algorithm used in [10], based on the radix hash join algorithm described by Kim et al. [38] (source code available [9]). In contrast, the Mondrian Data Engine performs sort-merge join to maximize row buffer locality. In all cases, we use 16-byte tuples comprising an 8-byte integer key and an 8-byte integer payload, representing an in-memory columnar database. We assume a uniform distribution of keys in the input relations.

We describe the *Join* operator ($R \bowtie S$) in more detail, being the most complicated of the four basic data operators. We assume that keys in the R and S relations follow a foreign key relationship: every tuple in S is guaranteed to find exactly one join match in R. The algorithm consists of two main phases, as illustrated in Table 2: *partitioning*, where data is shuffled across partitions, and *probe*, where the actual join is performed. The Mondrian Data Engine leverages data permutability to improve the efficiency of the partitioning phase, which involves two steps for both the small relation R and the large relation S: (i) building a histogram, and (ii) distributing data to the partitions. In the histogram step, all keys in the source memory partitions are hashed into N buckets, to determine their location in their destination partition, where N is the number of destination partitions. The hash function uses a number of the key's bits to determine each tuple's destination partition. In the code's CPU version, we use the keys' 16 low order bits, optimizing for our modeled system's private cache size. For the NMP systems, we use six bits, matching the total number of NMP vaults. In the second step of the partitioning phase, tuples are copied to their corresponding locations in their destination partitions, as determined in the histogram step.

In the *Join* operator's probe phase, matching tuples of the R and S relations are joined. This operation is fully localized within each partition. In the case of CPU, the phase starts with building a hash table and computing a prefix sum (similar to the first step of the partitioning phase) for the elements of the smaller relation R. The purpose of this second hashing is to group together keys of the R relation that map to the same hash index, and store them in a contiguous address range (further referred to as an *index range*). Finally, for each tuple in S, the index range of R that corresponds to the S tuple's key hash is probed, and the matching R tuple in the index range is joined with the probing S tuple. In the case of the Mondrian Data Engine, the probe phase is implemented as a sort-merge join. Thus, all data in the local vault is sorted and the two relations are joined doing a final pass.

For the *Group by* operator, we altered the last step of the join's algorithm to perform six aggregation functions *(avg, count, min, max, sum,* and *sum squared)*, which are applied to all the tuple groups. Our modeled *Group by* query results in an average group size of four tuples. In the partitioning phase of the sort

CPU baseline	
Cores	ARM Cortex-A57; 64-bit, 2GHz, OoO, RMO 3-wide dispatch/retirement, 128-entry ROB
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i 64-byte blocks, 2 ports, 32 MSHRs 2-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 4MB, non-inclusive 16-way, 1 bank/tile, 4-cycle hit latency
Coherence	Directory-based MESI
NMP baseline	
Cores	Qualcomm Krait400; 64-bit, 1GHz, OoO RMO, 3-wide dispatch/retire, 48-entry ROB
L1 Caches	Same as CPU-centric
Mondrian Data Engine	
Cores	ARM Cortex-35; 64-bit, 1GHz, in-order, RMO dual-issue. 1024-bit fixed-point SIMD unit
Common	
DRAM Organization	32GB: 8 layers × 16 vaults × 4 stacks
DRAM Timing	t_{CK} = 1.6 ns, t_{RAS} = 22.4 ns, t_{RCD} = 11.2 ns t_{CAS} = 11.2 ns, t_{WR} = 14.4 ns, t_{RP} = 11.2 ns
NOC	2D mesh, 16B links, 3 cycles/hop
Inter-HMC Network	Fully connected for NMP, Star for CPU SerDes links @ 10GHz: 160Gb/s per direction

Table 3: System parameters for simulation on Flexus.

operator, we use *high order bits*, as opposed to low order bits used by the join, to obtain the partitions that contain the keys strictly smaller or larger than the keys in any other partition. Then, in the probe phase, all tuples within each partition are sorted using quicksort, in the case of CPU, and mergesort, in the case of Mondrian Data Engine. The last and simplest operator, scan, does not have a data partitioning phase; each input data partition is scanned in parallel, and each tuple is compared to the searched value.

Evaluated configurations. We evaluate a number of different system configurations to break down the benefits of the algorithm-hardware co-design the Mondrian Data Engine is based on. In the partitioning phase, we separate the benefits of permutability and deploying SIMD units by evaluating two configurations in addition to the NMP baseline (*NMP*) and Mondrian: *NMP-perm* and *Mondrian-noperm*. The former uses the NMP baseline's computation units but also leverages permutability, the latter uses Mondrian's SIMD units without leveraging permutability. In the probe phase, we aim to illustrate that deploying an algorithm that favors sequential accesses over random is not beneficial without the proper hardware support to translate this favorable memory access pattern into high effective bandwidth utilization and, conversely, performance. We therefore deploy two versions of the *Join* and *Group-by* operators on the NMP baseline: the hash-based version (random memory accesses, also used by the CPU) and the sort-based version (sequential memory accesses, also used by Mondrian). We refer to these two versions as *NMP-rand* and *NMP-seq*, respectively.

Performance model. As the executed code differs across the evaluated systems, we base our performance comparisons on the runtime of each operator on the same datasets. We use the SMARTS sampling methodology [70] to reduce the turnaround time of cycle-accurate simulation, and measure the achieved IPC in each case. We then use fast functional-only simulation to measure the instruction count of each algorithm phase. Finally, we multiply the IPC with the number of instructions to estimate the runtime of each operator on each of the architectures.

Energy model. We develop a custom energy modeling framework to take the energy consumption of all major system components into account, combined with event counts from Flexus (cache accesses, memory row activations, etc.) to estimate the total expended energy for each experiment. Table 4 summarizes these power and energy estimations. We assume a 28nm technology and estimate LLC power using CACTI 6.5 [41]. We estimate core power based on the core's peak power and its utilization statistics. For the HMC, we scale the energy data reported by Micron [33] and separate the row activation from the access energy by modeling HMC on CACTI-3DD [14]. The HMC features the smallest row buffer size among commercial stacked DRAM devices, resulting in a conservative activation/access row buffer energy ratio. The energy benefits of reducing row buffer activations would be even larger for devices with larger row buffers, such as Wide I/O 2 [34] and HBM [35]. We also estimate HMC background power and energy per operation based on Micron models as implemented in DRAMSim2 [58] and Micron data sheets [47]. Finally, we use previously published data for the NOC [24, 65] and SerDes energy [22, 44].

System Component	Power / Energy
CPU Core	Power: 2.1W
NMP Baseline Core	Power: 312mW
Mondrian Core	Power: 180mW
LLC	Access Energy: 0.09nJ Leakage Power: 110mW
NOC	Energy: 0.04$pJ/bit/mm$ Leakage Power: 30mW
HMC (per 8GB cube)	Background Power: 980mW Activation Energy: 0.65nj Access Energy: 2pj/bit
SerDes	Idle: 1pj/bit, Busy: 3pj/bit

Table 4: Power and energy of system components.

7 EVALUATION

We now proceed to evaluate the four systems described in the methodology section: CPU-centric, NMP-perm, NMP-rand, NMP-seq, Mondrian-noperm, and Mondrian. We use the four common data operators that are representative of data analytics services (*Scan, Sort, Group by, Join*) to compare all of the above system configurations in terms of performance (§7.1), energy and efficiency (§7.2).

7.1 Performance

We break down the data operators into their two main phases, namely *partitioning* and *probe*. We first show the results for the two phases separately and then combine them to deduce the overall performance of each evaluated system. All performance results appear as speedups over the CPU baseline.

Table 5 shows the speedup for the partitioning phase on NMP, NMP-perm, Mondrian-noperm and Mondrian. The partitioning phase for all operators is almost identical, so we only show the results for the *Join* operator. The conventional partition algorithm executed by CPU, NMP and Mondrian-noperm builds and maintains a histogram of tuple keys to calculate the exact destination address for each tuple. The histogram manipulation code suffers from heavy data dependencies and results in low bandwidth utilization and, ultimately, poor performance. NMP outperforms the CPU by 58× due to its high parallelism and proximity to memory, but falls short of its full potential with an average per-core IPC of 0.98. Overall, NMP utilizes only 1.0GB/s of memory bandwidth per vault.

Mondrian-noperm leverages its SIMD units to outperform NMP by 2.4×, but is still compute-bound. Due to the same data dependency issues observed in NMP, Mondrian-noperm cannot use SIMD instructions throughout the partition loop and manages to utilize 2.4GB/s of memory bandwidth on average.

NMP-perm and Mondrian exploit data permutability during partitioning. NMP-perm achieves a speedup of 1.7× over NMP mainly due to the execution of simpler code. Permutability eschews the need for destination address calculation and greatly reduces dependencies in the code. However, even after leveraging permutability, NMP-perm utilizes only 1.6GB/s memory bandwidth per vault.

Finally, Mondrian leverages both SIMD and permutability. It uses SIMD instructions across the entire partition loop, enabling data-parallel processing of 8 tuples, and shifts the performance bottleneck

System	Speedup over CPU
NMP	58×
NMP-perm	98×
Mondrian-noperm	142×
Mondrian	273×

Table 5: Partition speedup vs. CPU.

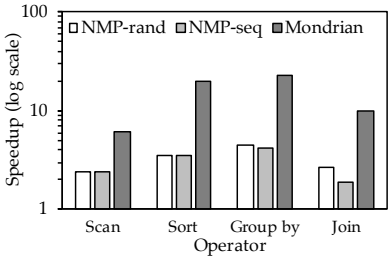


Figure 6: Probe speedup vs. CPU.

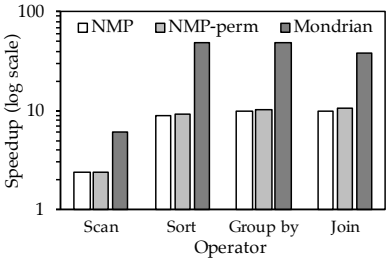


Figure 7: Overall speedup vs. CPU.

to the SerDes links' bandwidth. The outcome is a bandwidth utilization of 4.5GB/s per vault and speedups of 1.9×, 2.8× and 273× over Mondrian-noperm, NMP-perm, and CPU, respectively.

Fig. 6 shows the probe phase performance results. In the case of *Scan*, NMP-rand and NMP-seq are identical, as they execute the same code, and achieve a performance improvement of 2.4× over the CPU due to memory proximity. However, NMP cannot fully utilize the memory bandwidth, reaching a per-vault bandwidth utilization of only 2.5GB/s, due to a combination of a narrow pipeline and code with heavy data dependencies. Although the overall CPU performance is lower than NMP, the per-CPU-core bandwidth utilization is 4.3GB/s, thanks to the cores' larger instruction window and higher frequency. Mondrian overcomes NMP's limitations by using wide SIMD operations that allow entire tuples to be processed by a single SIMD instruction, improving performance by 2.6× as compared to the NMP baselines. Mondrian utilizes 6.7GB/s of per-vault memory bandwidth, nearing the theoretical peak of 8GB/s.

Sort shows a similar trend as *Scan*, with two key changes. The performance gap between NMP and CPU grows due to an increased number of memory accesses. The gap between NMP and Mondrian grows even further because of *Sort's* higher computational requirements, which are better accommodated by Mondrian's wide SIMD. Both NMP-seq and NMP-rand execute mergesort.

For *Group by* and *Join*, we observe that NMP-rand outperforms NMP-seq, even though NMP-seq's IPC of 0.95 is significantly higher than NMP-rand's IPC of 0.24. While NMP-seq is faster due to its sequential memory access patterns, it is not fast enough to compensate for the added *logn* factor to algorithmic complexity. Mondrian's wide SIMD units absorb this algorithmic complexity bump, achieving speedups of 22× over the CPU and 5× over the best NMP.

Fig. 7 trivially combines the operators' partitioning and probe phases to represent the complete execution. For NMP and NMP-perm, we combine their corresponding partition phase with the best performing probe algorithm, NMP-rand. Mondrian's overall speedup over the CPU and the best NMP baseline, composed of NMP-perm+NMP-rand, peaks at 49× and 5×, respectively.

7.2 Energy and Efficiency

Fig. 8 shows the energy breakdown for CPU, NMP, NMP-perm, and Mondrian across their main components. In the CPU case, DRAM bandwidth is severely underutilized and core energy dominates. However, we simulate a relatively small system due to practical constraints, and we expect commercial systems to have more DRAM

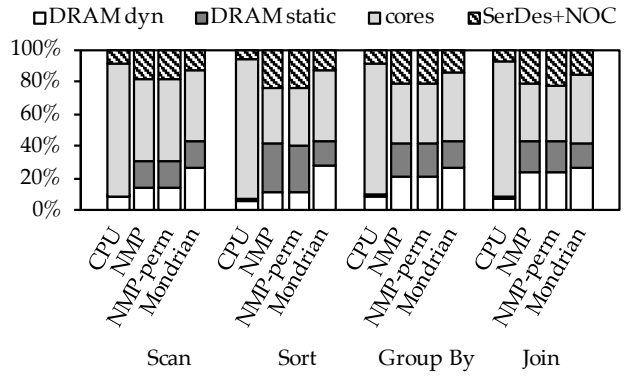


Figure 8: Energy breakdown.

per core (i.e., 8GB+ per core as opposed to the evaluated 2GB), making the DRAM energy component more significant. In NMP and Mondrian, lower core power promotes other components as the main sources of system energy. In the NMP architectures, execution is dominated by the probe phase; therefore, the impact of the partitioning mechanism used is small and the energy profiles of NMP and NMP-perm are near-identical. Mondrian's energy breakdown reflects its aggressive DRAM bandwidth utilization, where static-dominated components of the energy utilization, namely *SerDes* and *DRAM static*, are relatively smaller as compared to NMP.

Finally, Fig. 9 illustrates the overall efficiency improvement—in terms of performance per watt—for each of the three systems over the CPU baseline. While efficiency follows the performance trends, the gains are smaller than the performance improvements, reflecting Mondrian's high utilization of system resources. Mondrian cores draw higher dynamic power for higher performance and DRAM bandwidth utilization. Overall, Mondrian improves efficiency by 28× over the CPU and by 5× over the best NMP baseline.

8 RELATED WORK

Near Memory Processing. Several attempts to go down the promising NMP path have been made in the past. However, ambitious projects such as FlexRAM [37], IRAM [54], DIVA [28], and Active Pages [52] that introduced novel NMP architectures faced technology as an ultimate showstopper. Integration of logic and memory on the same die proved particularly challenging, damaging yields, while thermal concerns severely limited the logic's performance.

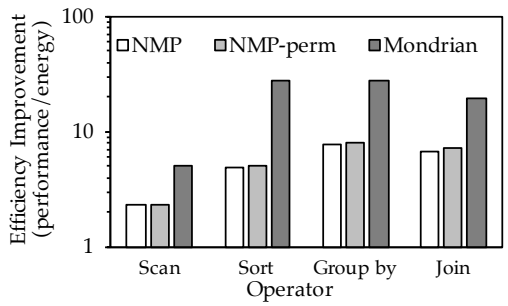


Figure 9: Efficiency improvement vs. CPU (log scale).

Hence, the NMP idea was limited to academic research, and eventually faded away. The interest in NMP has recently been rekindled by advancements in die-stacking technology, which enables the tight integration of logic and memory while still allowing separate chip manufacturing. Commercial products such as AMD-Hynix's HBM [5, 35], Micron's HMC [46], and Tezzaron's DiRAM [63] are already available, and even though these existing products are currently used as memory bandwidth boosters, ongoing efforts are focused on leveraging the logic layer that lies under the memory dies for NMP.

Several recent proposals seek to make NMP practical and maximize its efficiency. Tesseract [2] is a specialized NMP architecture for graph processing that employs graph-processing-specific prefetching schemes to utilize the ample available memory bandwidth, but does not optimize for memory access efficiency (i.e., row buffer locality). Ahn et al. [3] focus on facilitating programming of NMP architectures through ISA extensions. Gao et al. [22] propose an NMP architecture with an optimized communication scheme across NMP compute units. It still relies on caches for performance and proposes software-assisted coherence to alleviate coherence overheads. In contrast, we advocate against the traditional approach of relying on cache-optimized algorithms. Instead, we propose adapting the algorithms and designing the NMP hardware to extract performance from immense memory bandwidth through streaming.

Pioneer work on near-memory data analytics has proposed an HMC-based architecture for MapReduce-style applications [56, 57], which share a lot of similarities with the data operators we investigated. However, their approach is limited to mapping the CPU algorithm to a set of simple in-order cores, used as NMP compute units on the HMC's logic layer. As our algorithmic-hardware co-design observations illustrate, such an approach leaves a significant efficiency improvement headroom.

HRL [23] introduces a novel FPGA/CGRA hybrid as a compute substrate to maximize efficiency within NMP's tight area & power budget. We show that reconfigurable logic is not necessary for efficient bandwidth utilization in NMP; restructuring data analytics algorithms to be streaming-friendly and amenable to wide, data-parallel processing (SIMD) is enough to meet the goal of utilizing the available bandwidth within the NMP constraints, while maintaining a familiar programming model. We show that NMP favors algorithms that trade off algorithmic complexity for sequential accesses, and can achieve high efficiency by maximizing memory bandwidth utilization through streaming and wide computation units.

Dynamic address remapping. Impulse [13] introduces a software-managed address translation layer in the memory controller, to restructure data and improve memory bandwidth and effective cache capacity. However, it does not change the internal DRAM data placement and access pattern, hence it does not directly improve DRAM access efficiency. Akin et al. [4] also identify that data relocation in 3D memory architectures results in suboptimal memory accesses, and introduces the reshape accelerator that dynamically and transparently remaps physical addresses to DRAM addresses to increase row buffer locality. In contrast, the Mondrian Data Engine changes the physical addresses of data objects, by relying on the end-to-end observation that in data analytics, operations often occur on buckets of data rather than individual objects, hence a memory bucket's internal organization can be relaxed by the hardware using hints coming from software, for efficiency benefits.

Partitioning acceleration. HARP [68] identifies partitioning as a major runtime fraction of data analytics operators and introduces a specialized partitioning accelerator to alleviate that overhead. HARP is synergistic with our optimization for partitioning, which exploits memory permutability in data analytics to maximize row buffer locality. HARP always preserves record order by design, conservatively assuming that all tuple outputs are in an "interesting order" [59]. Interesting orders are explicitly marked by the query planner, and the Mondrian Date Engine's design allows enabling the permutability optimization on demand.

Data analytics acceleration. With data analytics taking center stage in modern computing, there have been proposals for analytics accelerators from both the industry and academia, such as Oracle's DAX [61], Widx [39] and Q100 [69]. Unlike these accelerators, Mondrian's compute units are specifically designed to fit under NMP's tight area and power constraints. On-chip accelerators, like Q100, employ specialized pipelined units to maximize on-chip data reuse and minimize expensive off-chip memory accesses. In contrast, Mondrian leverages its proximity to memory to achieve high performance with simple general-purpose compute units.

9 CONCLUSION

The end of Dennard scaling has made the design of new architectures a one-way road for increasing the performance and energy efficiency of compute systems. At the same time, modern age data deluge is further pushing this urge, as information extraction requires plowing through unprecedented amounts of data. NMP architectures, which have recently become technologically viable, provide a great fit for energy-efficient data-centric computation, as they feature low-power logic that can utilize tightly coupled high-bandwidth memory. While NMP systems are architecturally different from CPU-centric ones, they both execute similarly structured programs: ones that involve lots of fine-grained random memory accesses. However, such accesses are inefficient in NMP systems, as random accesses result in an excess of expensive row buffer activations.

In this work, we showed that efficient NMP requires radically different algorithms from the ones that are typically preferred for

CPUs. Instead of relying on cache locality and data reuse, NMP's ample memory bandwidth favors memory streaming through sequential accesses. Adapting modern data analytics to the strengths of NMP architectures requires algorithmic modifications to maximize sequential memory accesses, and hardware support to operate on data streams at memory bandwidth. We materialized our key observations by designing the Mondrian Data Engine, a novel NMP architecture optimized for efficient data analytics. Compared to a CPU-centric and a baseline NMP system, the Mondrian Data Engine improves the performance of basic data analytics operators by up to 49× and 5×, and efficiency by up to 28× and 5×, respectively.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers and Arash Pourhabibi for their precious comments and feedback. This work has been partially funded by a Microsoft Research PhD scholarship and the following projects: Nano-Tera *YINS*, CHIST-ERA *DIVIDEND*, and Horizon 2020's *dRedBox* and *CE-EuroLab-4-HPC*.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*. 105–117. https://doi.org/10.1145/2749469.2750386

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*. 336–348. https://doi.org/10.1145/2749469.2750385

[4] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*. 131–143. https://doi.org/10.1145/2749469.2750397

[5] AMD. 2016. High Bandwidth Memory, Reinventing Memory Technology. (2016). Retrieved April 26, 2017 from http://www.amd.com/en-us/innovations/software-technologies/hbm.

[6] ARM. 2017. Cortex-A35 Processor. (2017). Retrieved April 26, 2017 from https://www.arm.com/products/processors/cortex-a/cortex-a35-processor.php.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2012)*. 53–64. https://doi.org/10.1145/2254756.2254766

[8] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment* 7, 1 (Sept. 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[9] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Ozsu. 2013. Multicore hash joins source code. (2013). Retrieved April 26, 2017 from https://www.systems.ethz.ch/node/334/.

[10] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th International Conference on Data Engineering, (ICDE 2013)*. 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2011)*. 37–48. https://doi.org/10.1145/1989323.1989328

[12] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Preceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. 225–237. http://www.cidrdb.org/cidr2005/papers/P19.pdf

[13] John B. Carter, Wilson C. Hsieh, Leigh Stoller, Mark R. Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael A. Parker, Lambert Schaelicke, and Terry Tateyama. 1999. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA 1999)*. 70–79. https://doi.org/10.1109/HPCA.1999.744334

[14] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE 2012)*. 33–38. https://doi.org/10.1109/DATE.2012.6176428

[15] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*. 269–284. https://doi.org/10.1145/2541940.2541967

[16] Bill Dally. 2015. Keynote: Challenges for Future Computing Systems. (2015). Retrieved April 26, 2017 from https://www.cs.colostate.edu/~cs575dl/Sp2015/Lectures/Dally2015.pdf.

[17] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. https://doi.org/10.1145/2408776.2408794

[18] Hewlett-Packard Enterprise. 2015. The Machine: A new kind of computer. (2015). Retrieved April 26, 2017 from http://www.labs.hpe.com/research/themachine/.

[19] Babak Falsafi, Mircea Stan, Kevin Skadron, Nuwan Jayasena, Yunji Chen, Jinhua Tao, Ravi Nair, Jaime H. Moreno, Naveen Muralimanohar, Karthikeyan Sankaralingam, and Cristian Estan. 2016. Near-Memory Data Services. *IEEE Micro* 36, 1 (2016), 6–13. https://doi.org/10.1109/MM.2016.9

[20] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. 37–48. https://doi.org/10.1145/2150976.2150982

[21] Apache Software Foundation. 2017. Apache Spark. (2017). Retrieved April 26, 2017 from http://spark.apache.org/.

[22] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT 2015)*. 113–124. https://doi.org/10.1109/PACT.2015.22

[23] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proceedings of the 2016 International Symposium on High Performance Computer Architecture (HPCA 2016)*. 126–137. https://doi.org/10.1109/HPCA.2016.7446059

[24] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. 2011. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA 2011)*. 401–412. https://doi.org/10.1145/2000064.2000112

[25] Linley Group. 2015. Hexagon 680 Adds Vector Extensions. *Mobile Chip Report* (September 2015).

[26] Linley Gwennap. 2013. Qualcomm Krait 400 hits 2.3 GHz. *Microprocessor report* 27, 1 (January 2013), 1–6.

[27] Linley Gwennap. 2015. Cortex-A35 Extends Low End. *Microprocessor Report* 29, 11 (November 2015), 1–10.

[28] Mary W. Hall, Peter M. Kogge, Jefferey G. Koller, Pedro C. Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John J. Granacki, Jay B. Brockman, Apoorv Srivastava, William C. Athas, Vincent W. Freeh, Jaewook Shin, and Joonseok Park. 1999. Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture. In *Proceedings of the ACM/IEEE Conference on Supercomputing, (SC 1999)*. 57. https://doi.org/10.1145/331532.331589

[29] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA 2016)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[30] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward Dark Silicon in Servers. *IEEE Micro* 31, 4 (2011), 6–15. https://doi.org/10.1109/MM.2011.77

[31] Mark Harris. 2013. Unified memory in CUDA 6. (2013). Retrieved April 26, 2017 from http://on-demand.gputechconf.com/supercomputing/2013/presentation/SC3120-Unified-Memory-CUDA-6.0.pdf

[32] IBM. 2017. IBM DB2. (2017). Retrieved April 26, 2017 from http://www.ibm.com/analytics/us/en/technology/db2/.

[33] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*. IEEE, 87–88. https://doi.org/10.1109/VLSIT.2012.6242474

[34] JEDEC. 2013. Wide I/O 2 Standard. (2013). Retrieved April 26, 2017 from http://www.jedec.org/standards-documents/results/jesd229-2.

[35] JEDEC. 2015. High Bandwidth Memory (HBM) DRAM. (2015). Retrieved April 26, 2017 from https://www.jedec.org/standards-documents/docs/jesd235a.

[36] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*. 158–169. https:

//doi.org/10.1145/2749469.2750392

[37] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Vi Lam, Josep Torrellas, and Pratap Pattnaik. 1999. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the IEEE International Conference On Computer Design, VLSI in Computers and Processors, (ICCD 1999)*. 192–201. https://doi.org/10.1109/ICCD.1999.808425

[38] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (Aug. 2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[39] Yusuf Onur Koçberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T. Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual International Symposium on Microarchitecture (MICRO 2013)*. 468–479. https://doi.org/10.1145/2540708.2540748

[40] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[41] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the 2011 International Conference on Computer-Aided Design (ICCAD 2011)*. 694–701. https://doi.org/10.1109/ICCAD.2011.6105405

[42] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA 2009)*. 267–278. https://doi.org/10.1145/1555754.1555789

[43] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[44] Mozhgan Mansuri, James E. Jaussi, Joseph T. Kennedy, Tzu-Chien Hsueh, Sudip Shekhar, Ganesh Balamurugan, Frank O'Mahony, Clark Roberts, Randy Mooney, and Bryan Casper. 2013. A Scalable 0.128-1 Tb/s, 0.8-2.6 pJ/bit, 64-Lane Parallel I/O in 32-nm CMOS. *J. Solid-State Circuits* 48, 12 (2013), 3229–3242. https://doi.org/10.1109/JSSC.2013.2279052

[45] MEMSQL. 2017. MEMSQL: The Fastest In-Memory Database. (2017). Retrieved April 26, 2017 from http://www.memsql.com/.

[46] Micron. 2014. Hybrid Memory Cube Second Generation. (2014). Retrieved April 26, 2017 from http://investors.micron.com/releasedetail.cfm?ReleaseID=828028.

[47] Micron. 2017. DDR3 SDRAM System-Power Calculator. (2017). Retrieved April 26, 2017 from https://www.micron.com/support/tools-and-utilities/power-calc.

[48] Nooshin Mirzadeh, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. 2015. Sort vs. hash join revisited for near-memory execution. In *Proceedings of the 5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*. http://acs.ict.ac.cn/asbd2015/papers/ASBD_2015_submission_3.pdf

[49] Cavium Networks. 2014. Cavium Announces Availability of ThunderX: Industry's First 48 Core Family of ARMv8 Workload Optimized Processors for Next Generation Data Center & Cloud Infrastructure. (2014). Retrieved April 26, 2017 from http://www.cavium.com/newsevents-Cavium-Announces-Availability-of-ThunderX.html.

[50] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. 677–689. https://doi.org/10.1145/2723372.2749436

[51] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*. USENIX, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[52] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA 1998)*. 192–203. https://doi.org/10.1109/ISCA.1998.694774

[53] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The case for RAMCloud. *Commun. ACM* 54, 7 (2011), 121–130. https://doi.org/10.1145/1965724.1965751

[54] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (Mar 1997), 34–44. https://doi.org/10.1109/40.592312

[55] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. 2016. Near-Memory Address Translation. *CoRR* abs/1612.00445 (2016). http://arxiv.org/abs/1612.00445

[56] Seth H. Pugsley, Jeffrey Jestes, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *IEEE Micro* 34, 4 (2014), 44–52. https://doi.org/10.1109/MM.2014.54

[57] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proceedings of the 2014 International Symposium on Performance Analysis of Systems and Software (ISPASS 2014)*. 190–200. https://doi.org/10.1109/ISPASS.2014.6844483

[58] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* 10, 1 (2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[59] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD 1979)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[60] Minglong Shao, Anastassia Ailamaki, and Babak Falsafi. 2005. DBmbench: fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative Research*. 254–267. https://doi.org/10.1145/1105634.1105653

[61] R. Sivaramakrishnan and S. Jairath. 2014. Next generation SPARC processor cache hierarchy. In *IEEE Hot Chips 26 Symposium (HCS), 2014*. 1–28. https://doi.org/10.1109/HOTCHIPS.2014.7478828

[62] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27. http://sites.computer.org/debull/A13june/VoltDB1.pdf

[63] Tezzaron. 2017. DiRAM4 3D Memory. (2017). Retrieved April 26, 2017 from http://www.tezzaron.com/products/diram4-3d-memory/.

[64] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. 2017. Fat Caches for Scale-Out Servers. *IEEE Micro* 37, 2 (2017), 90–103.

[65] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. 2014. BuMP: Bulk Memory Access Prediction and Streaming. In *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO 2014)*. 545–557. https://doi.org/10.1109/MICRO.2014.44

[66] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal streams in commercial server applications. In *4th International Symposium on Workload Characterization (IISWC 2008)*. 99–108. https://doi.org/10.1109/IISWC.2008.4636095

[67] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31. https://doi.org/10.1109/MM.2006.79

[68] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*. 249–260. https://doi.org/10.1145/2485922.2485944

[69] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: the architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*. 255–268. https://doi.org/10.1145/2541940.2541961

[70] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA 2003)*. 84–95. https://doi.org/10.1109/ISCA.2003.1206991

[71] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *Proceedings of the 28th International Conference on Data Engineering (ICDE 2012)*. 1349–1350. https://doi.org/10.1109/ICDE.2012.148