Sort vs. Hash *Join* Revisited for Near-Memory Execution

Nooshin S. Mirzadeh† Onur Kocberber† Babak Falsafi† Boris Grot‡
†*EcoCloud, EPFL* ‡*University of Edinburgh*

Abstract

Data movement between memory and CPU is a well-known energy bottleneck for analytics. Near-Memory Processing (NMP) is a promising approach for eliminating this bottleneck by shifting the bulk of the computation toward memory arrays in emerging stacked DRAM chips. Recent work in this space has been limited to regular computations that can be localized to a single DRAM partition. This paper examines a Join *workload, which is fundamental to analytics and is characterized by irregular memory access patterns. We consider several join algorithms and show that while near-data execution can improve both energy-efficiency and performance, effective NMP algorithms must consider locality, access granularity, and microarchitecture of the stacked memory devices.*

1. Introduction

Large-scale analytics on massive datasets have radically transformed every aspect of our society, from medicine and scientific discovery to business and warfare. As the volume of data has skyrocketed over the last decade, growing at a pace comparable to Moore's Law [19], computing platforms for data-intensive services have struggled to keep up with the load.

Power is the key bottleneck limiting the effectiveness of computing platforms in the post-Dennard era. Semiconductor makers are no longer able to scale down supply voltages to offset the increase in switching energy that accompanies each doubling of transistors. As a result, the growing demand for data-intensive services has ushered an era of ever-larger and more power-hungry datacenters. Indeed, the power draw of leading-edge datacenters has been projected to increase 10-fold over the decade, from around 5MW per datacenter in 2005 to 50MW in 2015 [20].

The challenge, and also an opportunity, for system designers is to bridge the gap between data and technology through specialized architectures that leverage application characteristics to boost processing efficiency. Our work takes a step in this direction for in-memory analytics in the context of contemporary databases. Specifically, we examine the *Join* operation, which prior work has identified as the single largest contributor to the execution time and power consumption of an analytic pipeline [15]. The join operation combines records that match a specified predicate from a pair of database tables or columns. To reduce the number of comparisons, existing join algorithms either sort or hash the datasets. In both cases, the data movement between the memory and processor is the dominant source of energy consumption, since the actual per-element processing is trivial.

To address the data-movement bottleneck, we advocate pushing join operations toward the memory by leveraging emerging stacked DRAM designs. These designs integrate several DRAM dies on top of a logic die that controls DRAM and serves as an interface to the CPU. The logic die is an ideal substrate for simple hardware capable of executing data-intensive operations.

Recent work has examined near-memory processing (NMP) for very regular workloads that require no communication between DRAM modules [22]. In contrast, the join workload is less regular [8] and, for non-trivially sized datasets, intrinsically requires cross-DRAM-module communication. In order to be useful across a broad range of domains, NMP must be able to effectively accommodate such irregular computations. This work represents a first step in that direction by demonstrating that simply shifting the computation near the data is insufficient for attaining peak efficiency. Instead, NMP algorithms must also leverage locality, minimize the number of accesses to each DRAM row, and exploit microarchitectural characteristics of the memory substrate.

In this paper, we make the following contributions:

- We revisit sort-based and hash-based join algorithms and examine their efficiency in both NMP and CPU deployments.
- Our preliminary evaluation shows that for CPU-centric execution, hash-based algorithm performs better than the sort-based one both in terms of performance and energy consumption, corroborating the prior results [5]. However, in the NMP systems, the fine-grained random accesses in the partitioning phase penalize both the performance and energy benefits of the hash-based algorithms, making sort-based approaches more attractive for various workload scenarios.
- Overall, we show that performing a join near the memory can improve energy-efficiency by 4.5-10.7x and performance by 1.9-5.1x over the CPU execution.

2. Motivation

2.1. Database Operation: Join

Databases systems are crucial for maintaining, manipulating and analyzing large volumes of data. Contemporary databases optimized for large-scale analytics store the data as a collection of individual *columns* each containing *attributes* of a particular type. The columns are accessed via queries, written in a specialized query language (e.g., SQL), which are eventually converted into *physical operators* by the database system. *Join* is one of the fundamental operators in database systems, which iterates over a pair of columns to produce a single column based on a common key. Figure 1 shows

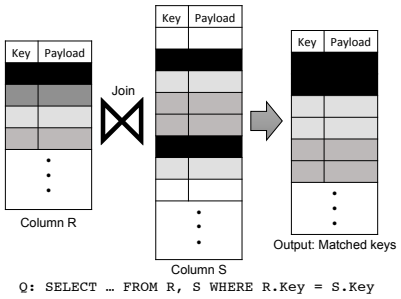

Figure 1: Join example.

an example of a join query. Join is an expensive operation whose efficient implementation plays an important role in both database queries' performance and energy consumption.

In practice, join algorithms fall into one of two categories: sort-based and hash-based. As the name implies, the idea of the sort join algorithm is sorting the two columns, and then merging them to find the matched keys. In contrast, the hash join starts by building a hash table on one of the columns and then *probes* the table with keys from the other column to find matches. To reduce the latency of accessing the hash table, a cache-aware radix-hash join algorithm [18] will partition both columns into slices that fit in the on-chip caches.

Both sort- and hash-based approaches have well-known overheads. The sort-based algorithm suffers from the high preparation time (i.e., sorting, whose time complexity is $O(N * logN)$). The $logN$ factor adversely affects not just the run time, but also the energy-efficiency of the algorithm. The hash-based approach is plagued by random memory accesses in both the build and probe phases, which diminish both performance and energy-efficiency.

2.2. Near-Memory Processing with Stacked DRAM

An attractive way to eliminate the high energy cost of moving the data between DRAM and CPU is to perform the processing near the memory arrays. *Processing in memory*, pursued by researchers in the late 90's, tried to do exactly that by integrating compute capabilities into DRAM chips [21, 13]. The ideas never caught on due to technology challenges (DRAM and logic benefit from very different process technologies) and the lack of a killer application or an economic driver.

Recently, advances in integration and packaging technology have led to an emergence of a new breed of memory chips that combine multiple DRAM dies into a vertical stack. Through-silicon vias (TSVs) provide a low-latency high-bandwidth interface between the DRAM dies and a logic die containing the DRAM control circuitry and CPU interfaces. As of this writing, Micron is shipping its second-generation stacked memory part called the Hybrid Memory Cube [11]. A competing product from Hynix/AMD [12] has also been announced.

By integrating high-speed logic on a dedicated die right next to DRAM, stacked memory presents an opportunity to bring processing close to the memory in a practical and affordable manner. Moreover, a strong economic incentive in the form of energy-efficiency is now in place to do so. With the two major barriers to adoption (technology and market) out of the way, there is now renewed interest in NMP [2].

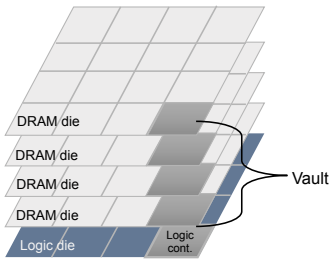

Figure 2: Hybrid Memory Cube (HMC).

Recent work examining NMP has focused on very regular processing localized to a single DRAM partition [22]. However, in order for NMP to be truly useful across a broad range of workloads, it must effectively handle less regular access patterns and data distributed across multiple partitions. From that standpoint, a Join workload makes for a good case study, as both sort- and hash-based approaches operating over a large memory space must deal with these issues.

2.3. Stacked-Memory Example: HMC

To understand the opportunities for NMP of database operations with stacked DRAM, we briefly examine the Hybrid Memory Cube as a representative design point. HMC consists of several DRAM dies and one logic die, stacked together using TSV technology. Each die is segmented into 32 autonomous partitions. A vertical stack of the partitions forms a *vault*, with each such vault controlled by a dedicated controller located on the logic die partition [11, 1]. Figure 2 shows a conceptual picture of the HMC.

Compared to conventional DRAM organized into multi-KB rows, the HMC features small rows of 256 bytes and a user-configurable fetch granularity of 32 to 128 bytes. To interface with the CPU, the HMC uses serial links running a packet-based protocol. Each 32-bit link operates independently and can be used to access any of the vaults in the HMC. The actual DRAM control logic is distributed among the vault controllers. A typical HMC access consumes 10.48pJ/bit energy which breaks down into two parts: accessing the DRAM itself (3.7pJ/bit) and the CPU-HMC interface (6.78pJ/bit) [11].

The 2x difference between the access and interface energy implies cheaper processing within an HMC as opposed to across multiple chips. However, the capacity of each stack is only up to 8GB. Thus, to support today's requirement, multiple stacks must be used in a system. Up to 8 HMCs can be chained together via the SerDes links [1]. Since a memory access may require traversing multiple SerDes links to reach the appropriate target HMC, and because a SerDes link traversal is more expensive than the actual DRAM access, it is essential to consider data placement and communication costs in the design of NMP algorithms.

3. Hash Join vs. Sort Join

3.1. Prior Work in Software

Over the past few decades, various algorithms have been proposed for efficient join implementation by taking advantage of the novel micro-architectural features of the processors such as multiple cores, larger on-chip caches, and additional memory bandwidth [18, 14, 5]. Among these, radix-hash and sort-merge *join* algorithms are the two most popular hardware-conscious join algorithms.

The hash-based join algorithms attempt to avoid the cache misses encountered during the join phase by partitioning both input columns into cache-sized chunks. The radix-hash join [18] proposes a lightweight, cache-aware radix-partitioning algorithm, which makes partitioning large inputs practical for main-memory systems. In contrast, parallel sort-merge join algorithms advocate for sorting both inputs to allow a join of two columns only with sequential accesses. Although joining the sorted columns is a trivial operation, because the merging (during sorting) is an inherently serial operation, the merge phase becomes the execution bottleneck. To mitigate such bottleneck, Albutiu et al. [3] proposes a partitioned massively parallel sort-merge join (P-MPSM) algorithm, which employs an additional partitioning phase to avoid the global merging of the partitioned and sorted data.

In this work, we focus on the two state-of-the-art join algorithms, namely radix-hash join [18] and parallel sort-merge join (P-MPSM) [3]. Unlike recent work, which only evaluated these algorithms in the context of CPU-centric execution [14, 5], we further evaluate them for NMP. The input columns of both algorithms, R and S, include an 8B key and 8B payload tuples with the assumption that $|R| \leq |S|$.

3.1.1. Radix-hash join The algorithm consists of three phases: partition, build and probe [18, 14]. In the partitioning phase, both columns are partitioned based on the upper B bits of the join key in three steps:

Step P1: Iterate over the entire relation and build a histogram counting the number of the tuples in each partition.

Step P2: Perform a prefix sum of the histogram to compute the starting addresses of the elements mapping to the respective indexes of the histogram.

Step P3: Reorder the tuples by scattering them to the corresponding partitions.

The partitioning phase is followed by the build phase: a hash table is created for each partition of R (the smaller relation). To locally build a hash table for each partition, the three steps described above is performed by using a hash function instead of the join key's upper bits. The result is reordered partitions of R.

The last phase is the probe phase, where we iterate over each partition of S and probe the hash table of R to find the matching keys and output the result.

In the CPU-centric execution, the number of partitions are chosen based on the L1-D cache size in order to fit a partition in it, whereas in NMP, the number of partitions is equal to the number of physical memory partitions to take advantage of the available parallelism in the probe and build phases.

3.1.2. Sort-merge join In P-MPSM [3], both columns are equally divided into C chunks (typically equal to the number of workers). The P-MPSM algorithm consists of three phases:

Phase 1 (partition R): R is partitioned into C partitions (similar to the partitioning phase in Radix-hash join).

Phase 2 (sort): Each chunk of S and R is locally sorted.

Phase 3 (merge-join): Each sorted chunk of R is merge-joined with each sorted chunk of S.

Note that because R is partitioned into the C chunks, each chunk of R is joined with only $1/C^{\text{th}}$ of each chunk of S.

In the CPU-centric execution, the number of chunks is equal to the core count, whereas in NMP it is equal to the number of physical memory partitions.

3.2. Join Algorithms Near Memory

Without loss of generality, we assume an HMC-like design (for convenience, referred to as "HMC" below) as a NMP system. However, the key ideas are applicable to other stacked memory architectures (e.g., HBM [12]).

There are two key aspects in NMP:

(1) The memory interface energy consumption is 2x more than the DRAM energy consumption in stacked-memories [11], therefore we need to exploit locality in one stack as much as possible.

(2) We need to minimize the number of fine-grain (e.g., single word) accesses to DRAM. The DRAM access has a wide interface in comparison to a cache access, and the access is destructive (i.e., even when a single word of a DRAM row is accessed, the whole row must be precharged in the row buffer, and then written back to DRAM). As a result, accessing one tuple key (typically 16 bytes) from DRAM consumes significant energy as the entire row is accessed. In contrast, "bulk" accesses that operate on the entire row can amortize the DRAM access energy over operations on multiple words [25].

3.2.1. Locality The number of partitions (in Radix-hash join) and the number of chunks (in P-MPSM) are equal to the total number of vaults (v) in HMCs to exploit the available parallelism. However, due to the nature of the algorithms, data movement is unavoidable. In Radix-hash join, the data movement occurs during the partitioning phase and in P-MPSM, data is moved during the partitioning and merge-join phases (phases 1 and 3).

In the partitioning phase of both algorithms, the tuples that belong to the other partitions will be sent to the other vaults, which can be either on the same chip or on the other chips. The number of chips is denoted by h. Given the uniform data distribution, $\frac{(h-1)}{h}$ of the tuples will move between the chips. The difference between the two algorithms is the amount of data that needs to be partitioned. In Radix-hash join, both columns S and R are partitioned, while in P-MPSM only R is

partitioned. If we assume that $|S| = c|R|$, Radix-hash join's data movement is $(c+1)$ times more than P-MPSM's data movement in the partitioning phase.

Radix-hash join does not suffer from data movement in its build and probe phases, because both phases are performed within a vault. However, P-MPSM, in merge-join phase, needs to move each R partition to the all S chunks. To do so, P-MPSM requires moving each partition of R, $v \times (h-1)$ times between the chips, which could result in a large amount of data movement. To reduce such data movement, we can move each R partition only once per HMC and have each receiving HMC broadcast the incoming partitions to its vaults. Thus, we need to move each partition of R just $(h-1)$ times. In total, we have v partitions which include $\frac{|R|}{v}$ amount of data and we need to move each partition $(h-1)$ times. Thus, P-MPSM's merge-join phase needs to move $(h-1) \times |R|$ amount of data.

The total data movement of Radix-hash join is equal to the data movement in partition phase which is $\frac{(h-1)}{h} \times (|R| + |S|) = \frac{(h-1)}{h} \times (c+1) \times |R|$. In contrast, the total data movement of P-MPSM is equal to the sum of data movement in partition and merge-join phases which is $\frac{(h-1)}{h} \times |R| + (h-1) \times |R| = \frac{(h^2-1)}{h} \times |R|$. Therefore, the ratio of data movement in Radix-hash join to P-MPSM is $\frac{(c+1)}{(h+1)}$. This ratio shows that by increasing c (i.e., the ratio of $|S|$ to $|R|$), Radix-hash join algorithm has more data movement than P-MPSM, which is not the case for the CPU-centric execution.

3.2.2. Accessing Memory The second key point in the NMP execution is to reduce word-granularity accesses to DRAM. To do so, we must avoid random accesses as much as possible. Considering the three phases of Radix-hash join, we can see that reordering data in both partition and build phases always requires random accesses. In the probe phase, reading S tuples is sequential, however, reading the corresponding R tuples requires random accesses to memory. As a result, Radix-hash join algorithm writes R and S columns randomly in partition phase and reads R column randomly in probe phase. On the other hand, all phases in P-MPSM algorithm access the data sequentially except for phase 1 (i.e., partition R), which randomly writes only R column.

Because a sequential access pattern amortizes the DRAM activation energy over all words in the DRAM row, it is preferred to a random access pattern that opens each row to operate on just one tuple. As a result, comparing the two algorithms based on their memory access patterns shows that for near-memory execution, P-MPSM is superior. The difference is particularly significant as the ratio of $|S|$ to $|R|$ increases.

3.3. Hardware Requirements

In order to execute the join algorithms near memory, several hardware components are required. First, every HMC vault needs to employ a logic unit, which is capable of sorting, hashing, comparing, and generating data load/stores. The load/store interface should be wide enough to be able to operate on an entire DRAM row at once (e.g., wide SIMD logic). The unit can either be programmable (e.g., a microcontroller) or fixed-function. Second, each HMC needs to have an inter-vault network-on-chip (NOC) with broadcast or multicast capability (see Section 3.2.1). Finally, a TLB may be used if virtual addressing is extended to cover near-memory execution. However, we observe that database management systems typically preallocate and pin large chunks of contiguous memory, which may imply that physical or segment-based addressing [6] is sufficient.

4. Methodology

Evaluated Systems: We compare the efficiency of Radix-hash join and P-MPSM on two platforms. *CPU-centric* represents a typical setup of a CPU and a memory. However, instead of commodity DDR3 DRAM, we consider four HMCs as a main memory for future-generation servers. In this design, the join algorithms execute on the CPU. The other platform is *NMP* in which join algorithms execute on the logic layer of the HMCs. It is important to note that although both Radix-hash join and P-MPSM can handle skewed data, in this work, for simplicity we assume uniform data distribution.

Table 1 summarizes the key parameters of the modeled systems. On the CPU side, we model a core microarchitecture that is expected to maximize overall performance and energy-efficiency for the target workload domain. Specifically, we choose a modest-complexity out-of-order core and augment it with a wide vector unit that is able to exploit the rich data-level parallelism that exists in analytic kernels [24]. Our specific design point combines a low-power 3-way OoO core modeled after the Qualcomm Krait 400 [10] with a high-performance 512-bit vector engine similar to that in the Intel Xeon Phi.

For the join logic, we model a simple micro-controller which can support 256B SIMD. We also assume that it is able to support bitonic merge sort for 16 elements. To support modeling data movement within a chip, we consider a 2D mesh NoC in $20mm \times 20mm$ chip. We also model ring topology for connecting the HMCs to each other and the CPU.

Performance evaluation: Our evaluation is based on a first-order analytical model. For the CPU-centric design, we use a similar analytical model used in the prior studies [14, 18]. Additionally, for the NMP design, we augment our model with the following two parameters: (1) latency of moving data from device to another in the partition phase, and (2) latency of random/sequential read/write operations within a physical memory partition.

Energy consumption: Table 1 summarizes the power and energy estimates for various components. We estimate the energy of the scalar core and L1 complex by using publically available data for the Krait 400 [10]. To account for the energy overhead of the vector unit, we use recent studies showing that SIMD/AVX engines on both a simple ARM core and a high-end Intel core add approximately 10% to core's dynamic

	Arch/uArch Features	Energy/Power
CMP	22nm 16 cores	
Core	OoO, 3-wide, 60-entry ROB 512-bit SIMD, 2.5GHz 64KB L1-I/D, 64B blocks	Power: 900mW
LLC	4MB, 16-way 8-cycle hit latency	Read/Write: .63nJ/.70nJ
HMC	4 cubes, ring 50nm DRAM, 8GB 8 DRAM dies, 32 vaults 10GB/s BW per vault 4 links per each cube BW per link: 60GB/s per dir.	Access: 3.7pj/bit I/O+logic: 6.78pj/bit
Join logic	22nm logic die 256B SIMD 2D mesh NoC	logic: 0.042pJ/bit NOC: 0.04pJ/pit/mm

Table 1: Evaluation parameters.

power for vector widths of 128 and 256 bits [7, 23]. Based on these results, we increase the scalar core's power by 20% to account for the modeled 512-bit vector engine.

Beyond the core, we estimate LLC power through CACTI 6.5 [16]. For the HMC, we use the energy data reported by Micron [11]. For the NOC, we use the previously published data [9]. Finally, we estimate the energy for the join design by using the numbers in [4] for a fixed point logic and scale it down to 22nm.

5. Evaluation

Figure 3 quantifies the normalized performance and energy benefits of the sort and hash-based join algorithms varying the relation (S) size. On the CPU-based platform, we observe that Radix-hash join constantly outperforms the P-MPSM by achieving a 52% better performance and 54% reduction in energy consumption on average. Although both algorithms have preparation overheads (i.e., sorting and partitioning), the sort-based algorithm suffers from a logarithmic preparation overhead dominated by the sort step, while the random accesses in Radix-hash join's partitioning phase are filtered by the L1-D cache as the algorithm can be tuned to take advantage of the L1-D cache parameters [5].

Near-memory processing (NMP) of both algorithms further improves the performance by taking advantage of parallelism across many vaults and HMC's high internal bandwidth. The offloaded NMP sort- and hash-based designs execute the join operations 2X and 3.5X faster, respectively, than the CPU-based algorithms by spreading the computation across the 128 partitions of the HMC. If the same degree of parallelism existed in the CPU baseline, NMP offloading would still deliver a 1.7x speedup by exploiting HMC's high internal bandwidth.

We observe that P-MPSM is sensitive to the $|R|$ to $|S|$ ratio and performs better than Radix-hash join when $|R|$ to $|S|$ ratio is 1:4 or greater. As explained in Section 4, P-MPSM has the

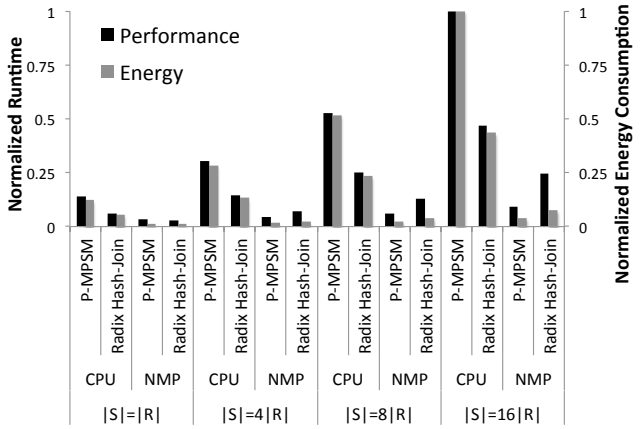

Figure 3: Performance and energy consumption of hash- and sort-based join algorithms (normalized to P-MPSM $|S| = 16|R|$ on CPU).

sorting overhead of both columns but requires partitioning only R, while Radix-Hash Join algorithm requires partitioning both R and S. Therefore, as the size of S grows, the performance penalty of remote and random writes in the partitioning of S leads to poor performance in Radix-hash join for larger probe relations. This behavior is not observed in CPU-based execution as the close coupling of the CPU and L1-D and their fast, narrow interface are well-matched to fine-grain accesses.

Figure 3 also compares the energy efficiency of NMP and CPU-based approaches. We observe that NMP reduces the energy consumption of the join operation by 81-95% over the CPU. Nearly two-thirds of the energy savings come from the use of wide specialized logic in the NMP design. The rest of the efficiency benefit is attributed to the reduction in chip-to-chip data movement.

We further explore the energy-efficiency of the NMP execution in Figure 4, which is normalized to P-MPSM |S| = 16|R| on CPU. It breaks down the energy consumption into three categories: 1) computation, 2) data movement, and 3) DRAM access energy. An important conclusion of Figure 4 is that Radix-hash join, which achieves significantly lower energy consumption than P-MPSM on the CPU, is just marginally better than P-MPSM in the $|R| = |S|$ NMP case. Moreover, Radix-hash join's energy consumption relative to P-MPSM increases consistently with the size of $|S|$. In the $|R| = 16|S|$ case, P-MPSM achieves 47% lower energy consumption than Radix-hash join.

As we analyze the two algorithms, we conclude that the locality and access granularity play an important role in the energy efficiency of NMP-based systems. P-MPSM's sorting phase, which is the energy hog on the CPU-based execution, almost disappears as P-MPSM algorithm sorts data locally in each HMC vault, where it effectively leverages DRAM row locality. At the same time, both probe and partitioning phase in Radix-hash join require data accesses at word granularity that burn significant energy in full DRAM row activations.

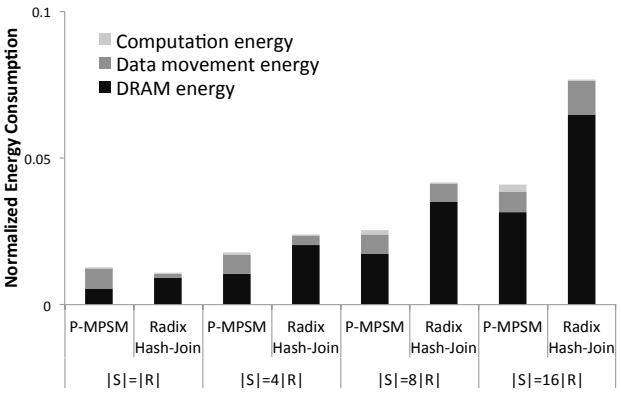

Figure 4: Energy breakdown of hash- and sort-based join algorithms in NMP (normalized to P-MPSM $|S| = 16|R|$ on CPU).

Overall, we observe that NMP can provide high performance and energy benefits, but the choice and tuning of the algorithm greatly impacts the overall performance and efficiency.

6. Related Work

Virtually all prior research on near-memory processing focused on bringing programmable cores close to the DRAM arrays. Examples include work in 90s and early 2000 (e.g., IRAM [21] and FlexRAM [13]), as well as recent work from AMD [26]. While fully programmable designs are clearly the most flexible, they do incur an array of complexities [17]. These include energy-efficiency and thermal considerations; orchestration of exception handling (e.g., in response to a page fault); and address translation challenges or, for designs aiming to extend virtual memory to cover the near-memory processing logic, extensions to the cache coherence protocol.

Recent work from Loh et al. made a general case for fixed-function near-memory accelerators [17]. The authors described a number of potential functions that could be viable targets for such acceleration and introduced a taxonomy to help reason about the design space. Our study represents a concrete case for near-memory accelerators. Our key contributions include identifying a specific high-value functionality for acceleration, characterizing the workload behavior, as well as identifying and addressing two key efficiency challenges – namely, data access granularity and locality.

7. Conclusion

This work revisits the "Hash vs. Sort" question in the context of Near-Memory Processing (NMP). Our results show that NMP systems have a great potential to overcome the performance and energy bottlenecks of today's CPU-centric systems. However, achieving the full potential of NMP requires a careful analysis of the algorithms. We find that data access granularity and locality play an essential role in NPM efficiency and performance. To increase the utility and effectiveness of such near-memory processing, future work should study data skew, and focus on composing multiple operations with minimal CPU involvement.

Acknowledgments

This work has been partially funded by the DeSyRe project of the Seventh Framework Programme of the European Commission, the Google Faculty Research Award, the Google Europe Doctoral Fellowship, and the Microsoft Research PhD Scholarship.

References

[1] "The HMC Specification 2.0," http://www.hybridmemorycube.org/.
[2] *First Workshop on Near-Data Processing (WoNDP)*, Dec. 8, 2013.
[3] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *VLDB*, vol. 5, no. 10, 2012.
[4] D. A. Bader, "Opportunities beyond single-core microprocessors."
[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *VLDB*, vol. 7, no. 1, 2013.
[6] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013.
[7] J. Cebrián, http://research.idi.ntnu.no/multicore/_media/jmcg_pp4ee.pdf.
[8] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi, "Accelerating database operators using a network processor," in *Proceedings of the 1st international workshop on Data management on new hardware*, 2005.
[9] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, 2011.
[10] L. Gwennap, "Qualcomm Krait 400 Hits 2.3GHZ," in *Microprocessor report*, January 2013.
[11] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSIT*, 2012.
[12] JESD235, "High Bandwidth Memory (HBM) DRAM," October 2013.
[13] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 1999.
[14] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *VLDB*, vol. 2, no. 2, 2009.
[15] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," in *MICRO*, 2013.
[16] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-Level Modeling for SRAM-Based Structures with Advanced Leakage Reduction Techniques," in *ICCAD*, 2011.
[17] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. Ping Zhang, and M. Ignatowski, "A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM," in *WoNDP*, 2013.
[18] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 4, 2002.
[19] A. McAfee and E. Brynjolfsson, "Big Data: The Management Revolution," in *Harvard Business Review*, October 2012.
[20] M. P. Mills, "Big Data, Big Networks, Big Infrastructure, and Big Power. An Overview of the Electricity Used by the Global Digital Ecosystem," August 2013.
[21] D. PattersonT, N. C. Anderson, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick, "A Case for Intelligent DRAM: IRAM," *IEEE Micro*, 1997.
[22] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in *ISPASS*, 2014.
[23] C. Shore, "Developing Power-Efficient Software Systems on ARM Platforms," http://www.iqmagazineonline.com/current/pdf/Pg48-53.pdf.
[24] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. Compilation in Query Execution," in *DaMoN*, 2011.
[25] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "Bump: Bulk memory access prediction and streaming," in *MICRO*, 2014.
[26] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski, "A New Perspective on Processing-in-Memory Architecture Design," in *MSPC*, 2013.