Deploying Hash Tables on Die-Stacked High Bandwidth Memory

Xuntao Cheng[*]
xuntao.cxt@alibaba-inc.com
Alibaba Group

Bingsheng He
hebs@comp.nus.edu.sg
National University of Singapore

Eric Lo
ericlo@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Wei Wang
wangwei@comp.nus.edu.sg
National University of Singapore

Shengliang Lu
lusl@nus.edu.sg
National University of Singapore

Xinyu Chen
xinyuc@comp.nus.edu.sg
National University of Singapore

Abstract

Die-stacked High Bandwidth Memory (HBM) is an emerging memory architecture that achieves much higher memory bandwidth with similar or lower memory access latency and smaller capacity, compared with main memories. Memory-intensive database algorithms may potentially benefit from these new features. Due to the small capacity of such die-stacked HBM, a hybrid memory architecture comprising both main memories and HBMs is promising for main-memory databases. As a starting point, we study a key data structure, hash tables, in such a hybrid memory architecture. In a large hash table distributed among multiple NUMA (non-uniform memory accesses) nodes and accessed by multiple CPU sockets, the data placement and memory access scheduling for workload balance are challenging due to the random memory accesses involved that are difficult to predict. In this work, we propose a deployment algorithm that first estimates the memory access cost and then places data in a way that exploits the hybrid memory architecture in a balanced manner. Evaluation results show that the proposed deployment is able to achieve up to three times performance improvement over the state-of-the-art NUMA-aware scheduling algorithms for hash joins in relational databases on present and simulated future hybrid memory architectures.

[*]Work done at Nanyang Technological University, Singapore. This is a preliminary version for an CIKM publication: Xuntao Cheng, Bingsheng He, Eric Lo, Wei Wang, Shengliang Lu, and Xinyu Chen. 2019. Deploying Hash Tables on Die-Stacked High Bandwidth Memory. In The 28th ACM International Conference on Information and Knowledge Management (CIKM âĂŹ19), November 3âĂŞ7, 2019, Beijing, China. ACM, New York, NY, USA, 12 pages. https: //doi.org/10.1145/3357384.3358015

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Xuntao Cheng, Bingsheng He, Eric Lo, Wei Wang, Shengliang Lu, and Xinyu Chen. 2019. Deploying Hash Tables on Die-Stacked High Bandwidth Memory. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The die-stacked HBM is an emerging memory architecture that integrates multiple DRAM layers with processing units on the same die, achieving a much higher bandwidth than that of the main memory. For example, the 2nd and 3rd generation of HBM can achieve 256 GB/s and 512 GB/s peak memory bandwidth, respectively [13, 14]. This level of high memory bandwidth is beneficial for improving the performance of memory-intensive database algorithms such as building and probing hash tables. However, the memory capacity of a HBM is limited by the die area, which are much smaller than that of the main memory [25]. On the other hand, HBMs have smaller or similar memory access latencies compared with the main memory [7, 30]. Thus, a hybrid memory architecture comprising both main memories and HBMs is promising for large databases. To exploit both types of memories in such architectures, we optimize the data placement and memory access scheduling in this paper.

We could benefit more from the added HBMs in the hybrid memory architecture if we are able to schedule more seuqnetial memory accesses to them, because random memory accesses on HBMs can be almost as expensive as those on main memories, despite HBMs' high peak memory bandwidth. On the other hand, with HBMs exposed as NUMA nodes together with main memories, unevenly placed random memory accesses among this increased number of NUMA nodes may cause workload imbalances and increase the total memory access cost significantly. To resolve these issues, we need to break the commonly accepted assumption that there is only one type of DRAM in the main memory and factor in such memory differences into system designs. Existing studies such as the round-robin scheduling policy used in the state-of-the-art NUMA-aware optimization [32] for hash tables is not sufficient due to its unawareness of such differences.

Database performance on such hybrid memory architecture is still an open area. In this paper, we start with a study on hash tables, which are among the most performance-critical data structures in main-memory databases [8, 34]. Many hardware-aware optimizations have been applied to reduce the memory access costs in building and probing hash tables, such as prefetching, various conflict resolving techniques, vectorizations, and thread-level parallelism [1, 7, 28, 32]. Despite of many previous studies on hash joins, there is little work on exploiting the hybrid memory architecture with HBMs.

In this paper, we propose a new deployment algorithm for hash tables aiming at reducing the memory access cost of building and probing the hash table by exploiting both the HBMs and the main memory. To facilitate such deployment, we design the hash table in a way that it can be partitioned into multiple subsets and efficiently distributed to multiple NUMA nodes with different memory types. We formulate the deployment of the hash table as a partition problem [18], where each partition has its estimated memory access cost and should be placed to NUMA nodes properly for workload balancing. To estimate the memory access costs, we convert the cost estimation problem to a classic problem of cost estimation of hash joins in query optimizations. We extend the state-of-the-art sampling-based method for such estimation [6], which are originally proposed for join size estimations. With proper estimations, we further introduce hardware-conscious heuristics to solve the partition problem on the hybrid memory architecture. To cope with data misplacement at runtime due to inaccurate estimations, we design an online migration and replication method to adjust the placement on the fly in order to further reduce workload imbalance.

HBMs have been integrated with CPUs on processors like the Intel Xeon Phi processor of the Knights Landing architecture (KNL). Still, it is a hot research topic on how to integrate HBMs into the future processors [33]. In this study, we make our exploration along this direction using two complementary test beds for our evaluations: the real system implementation on Intel KNL processor which has HBMs on-chip, and simulations to study the impact of future HBMs. We adopt a state-of-the-art memory simulator named Ramulator [17] to simulate HBMs with different specifications on which we can evaluate our proposed algorithm and study the impact of potential features of the HBMs.

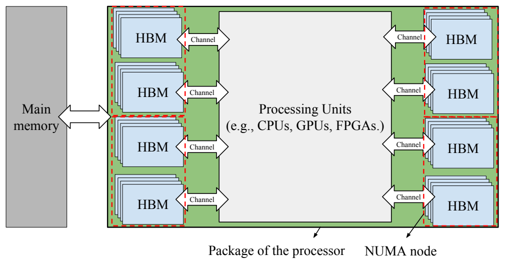


Figure 1: An emerging memory architecture with die-stacked HBMs.

Experimental results on KNL show that the proposed deployment algorithm can improve the performance of hash joins by up to three times compared with the fastest baseline [1, 7, 28]. Further simulation results show that hash join algorithms can benefit from future HBMs significantly. And, our algorithm adapts to changes in the NUMA topology and different latencies of the HBMs.

Overall, a hybrid memory architecture of HBM and main memory makes an interesting case for hash tables. Our paper has explored different HBM integration, as well as different memory architectures (including memory latencies, NUMA and memory types). With those various features, we demonstrate that cost-based adaption is essential for the performance of this hybrid memory architecture, which can shed light on the design of in-memory data structures on such a hybrid memory architecture.

2 Background and related work

2.1 Die-stacked High Bandwidth Memory

Recently, there have been many advancements in the area of die-stacked HBMs, including new processors with HBMs [11, 12], industry standards [13, 14] and academic studies [7, 25, 26, 33]. The idea is to stack one or multiple DRAM layers and then place them either by the side (i.e., 2.5D interposing) of the processor or on top of the processor (i.e., 3D stacking). This integration allows processor cores to access memories faster, compared with accessing the off-package main memory.

Figure 1 illustrates a hybrid memory architecture with both die-stacked HBMs and the main memory. Multiple DRAM layers are stacked together to form HBM modules placed next to CPUs in the same package, and can be exposed as a NUMA node with no CPU cores. The number of DRAM layers within each HBM stack is constrained by physical limitations such as heat and area constraints [25], which in turn constraints the total memory bandwidth and capacity. The second generation of the HBM (HBM2) specifies up to eight DRAM layers per stack. Both Intel and AMD have similar designs for processors with HBMs [11, 23, 33]. The bus width of the HBM on Intel KNL is 128-byte, which is 16 times as wide as the main memory. Thus, it is helpful to design the data layout and access patterns for the HBM to avoid memory accesses of small sizes to exploit the wide memory bus. In the meantime, balancing memory accesses to such increased number of NUMA nodes is important to reduce the makespan of all memory accesses. The capacity of die-stacked memory is usually in the range of several GBs, which is much smaller than main memory. A hybrid memory architecture comprising both types of memories is promising to take advantage of the best of both worlds.

2.2 Related Work on Die-stacked HBMs

Several cache designs exploiting die-stacked HBMs as a last-level cache (LLC) have been proposed [37, 38]. As LLC, die-stacked HBMs are transparent to software developers. However, few of them have been implemented in real hardware due to the complexity and cost of their implementation and verification [25]. And, the significant HBM capacity is not included in the total memory capacity if the HBMs are only utilized as caches.

Another approach to exploit HBMs is to expose all of them as NUMA nodes and allow developers to manage them explicitly. Meswani et al. studied a wide range of page replacement policies and proposed their epoch-based method to select and migrate hot pages to HBMs as software caches [25]. Uppal et al. proposed a workload-aware cache replacement policy [35]. In this study, we also exploit HBMs as NUMA nodes which allows us to strive for the highest memory bandwidth utilization and improve the workload balancing in the hybrid memory system.

2.3 Related Work on Resource Scheduling

Resource scheduling is important for database performance on modern hardware platforms when the hardware systems have more cores and memory features such as NUMA. Although there have been many previous studies on resource scheduling [9, 10, 21], to the best of our knowledge, little work has been done on how to design and implement databases on the hybrid memory systems of die-stacked HBM. The size constraint of the die-stacked HBM and different memory features of the two memory types are very different to existing resource scheduling.

Close to our study, the deployment of computation tasks and scheduling of resources have a significant impact on the performance of database algorithms on modern architectures [10, 21, 32]. Giceva et al. proposed to compact individual operators in a way that their requirements for computations and memory bandwidth are minimized, and then deploy compacted operators on the densest subset of NUMA nodes to reduce expensive remote memory accesses across QPIs [10]. Such approach does not suit the HBMs where the costs of remote memory accesses are different to that on conventional NUMA architectures and utilizing all NUMA nodes is necessary for performance improvements on the hybrid memory system comprising main memory and die-stacked HBM. Psaroudakis et al. evaluated a set of data and task placement strategies on NUMA architectures, many of which are based on partitioning columns [29]. While partitioning reduces remote memory accesses significantly, it does not necessarily improve the performance in many database algorithms because of its overhead. Our study also optimizes the NUMA effect of the hybrid memory system, which is relevant to the previous NUMA effect. Differently, both memory types have NUMA effect by itself, and our algorithms have to take care of them.

2.4 Related Work on Hash Tables

Hash tables are crucial as complex data structures in main memory databases, especially in hash joins [31]. Many software optimizations proposed for hash joins focus on optimizing memory accesses to hash tables [1, 3, 16, 28, 32]. Polychroniou et al. optimized a wide range of hash table designs such as linear probing, double hashing, and cuckoo hashing using SIMD instruction sets [28]. Barber et al. proposed the Concise Hash Table that reduces the memory footprint of hash tables significantly [4]. Barber et al. [3] develops a compact hash table design. These optimizations can be helpful in improving the performance of hash tables on the hybrid memory system of die-stacked HBM. Makreshanski et al. [24] studied the performance of shared hash joins from many queries (named MQJoin) on KNL. Their study demonstrated very promising results when the join fits into HBM, which is consistent to our experiments. In contrast, they have not studied the data management between DRAM and HBM, which is the focus of this paper. This paper focuses on deploying hash tables in a single machine with a hybrid memory system. Large-scale hash joins have been explored in distributed settings [5, 22], which are interesting future studies for this paper.

3 Overview

3.1 Problem Statement

We consider the performance of both building and probing hash tables, given input relations. Through data placements and scheduling memory accesses in the hybrid memory architecture, we improve the performance of build and probe by minimizing the memory access costs involved. This problem has two pillars. One is to utilize all HBM stacks and the main memory (exposed as NUMA nodes) to maximize the benefits of both types of memories. The other is to reduce workload imbalance among NUMA nodes.

There are several issues making this problem challenging to solve. Firstly, both build and probe of hash tables involve random memory accesses that are difficult to predict. Without accurate estimations of such memory accesses, especially their memory accesses costs, it is difficult to balance the distribution of memory accesses to avoid workload imbalance among different NUMA nodes, which in turn leads to performance degradation.

Secondly, existing NUMA-aware optimizations assume memories of the same socket (including multiple cores of a single CPU) has the same latency and bandwidth [10, 29]. Thus, they tend to interleave data across multiple NUMA nodes, expecting that the workload is balanced as long as each NUMA node has an equal share of all types of memory accesses. However, with die-stacked HBMs, this assumption is no longer true, so that we cannot rely on balancing data sizes for workload balance.

3.2 Overview of the solution

Our solution comprises two major components on deploying the hash table in the hybrid memory architecture. Firstly, we **estimate the memory access cost** for either build or probe of the hash table. To address the random memory access estimation, we formulate the cost estimation to a classic problem of estimating the cost of hash joins. We adopt the state-of-the-art sampling-based estimation method from Chen et al. [6], and extend it to estimate the memory access cost.

Secondly, we take advantage of the estimated costs to **deploy the hash table** across all the NUMA nodes for the build. For the probe, we introduce **online migration and replication** to adapt the placement of the hash table to the runtime. These online techniques are necessary if inaccurate estimations have led to an unbalanced distribution of the workload that undermines the performance significantly.

4 Design of Hash Table

4.1 Design Requirements

With the different characteristics of the HBMs and the main memory in mind, we have the following design requirements for the efficiency of hash tables.

- **Avoiding memory fragmentation.** Because the HBMs have much smaller memory capacity than the main memory, we should avoid wasting available memory spaces of the HBMs. Due to the potential memory allocations in the build operation of the hash table, memory fragmentation is a major problem to be avoided. On the other hand, excessive memory fragmentation also lead memory accesses of small sizes, which may fail to exploit the high memory bandwidth of HBM.
- **Utilizing the high sequential memory bandwidth.** The design of the hash table needs to be able to utilize the most important feature of the HBMs, its high sequential memory bandwidth.
- **NUMA-aware.** Because the HBMs introduce multiple NUMA nodes, the hash table needs to be scaled out and allocated to multiple NUMA nodes.

We evaluate existing hash table designs (such as [4, 20, 36]) to see whether they can satisfy these requirements. Table 1 summarizes five common hash table designs, including the state-of-the-art ones optimized for memory usages [4]. We have the following observations. Overall, we find that the previous studies fails to satisfy at least one design requirements for an efficient hash table on the hybrid memory architecture.

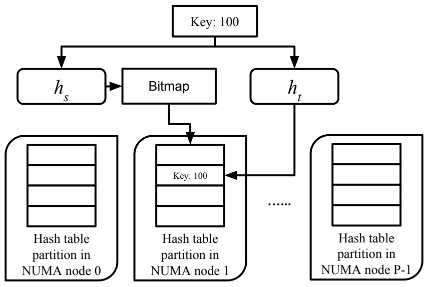
- Memory fragmentation is a major problem of separate hashing and its variants because all buckets are allocated in separate locations as nodes in a linked list.
- Although coalesced hashing can improve the memory efficiency, records of the same hash bucket may be scattered in the memory, making it not an array of buckets and its usage of the memory bandwidth poor.
- Cuckoo hashing can help in to reduce the impacts of memory fragmentation, but it uses multiple hash functions, which scatter records of the same hash bucket in the array.
- Concise hash table is the state-of-the-art memory-efficient design that satisfies all requirements, except that it is not designed to be aware of NUMA [4].

Table 1: Summary of hash table designs

Hash table design	**Avoid memory fragmentation**	**Bandwidth utilization**	**NUMA-aware**
Separate chaining	✗	✗	✓
Coalesced hashing [36]	✓	✗	✓
Open addressing	✓	✓	✗
Cuckoo hashing [20]	✓	✓	✗
Concise hash table [4]	✓	✓	✗
Our design	✓	✓	✓

4.2 Our proposed data structure

To overcome limitations in existing designs, we propose our own design in this study. Inspired by the previous study on compact hash table [3], we enhance the hash table design with linear probing (a form of open addressing) [27] to support the NUMA-aware placement. Our design can satisfy all the design requirements. In linear probing, the high bandwidth of the HBM can be utilized while probing a large hash table. Linear probing can be implemented in arrays



Key: 100
h_s
Bitmap
h_t
Key: 100
......
Hash table partition in NUMA node 0
Hash table partition in NUMA node 1
Hash table partition in NUMA node P-1

Figure 2: The proposed hash table design. h_s and h_t are two hash functions used to hash a record to its corresponding partition of the hash table and slots in the hash table, respectively.

and resolves collisions, by placing the new key into the closest following empty cell. Thus, it minimizes the memory fragmentation.

In a NUMA architecture consisting of P nodes, we choose to build P separate hash tables, with one table assigned to one node. We introduce a hash function (denoted as h_s hereafter) with a fan out of P to hash keys into its corresponding hash table in the NUMA architecture.

Figure 2 shows the overview of the design. We imagine a global hash table using linear probing where a hash function h_s is applied to hash keys to hash buckets in a partition. We split the global hash table into P partitions logically, with each partition assigned to a NUMA node. We use a bitmap to record which NUMA node the record is hashed to. Each partition has $\lceil \log_2 P \rceil$ bits in this bitmap. We use another function h_t to determine the offset of a record in a partition on its corresponding NUMA node. We apply a sampling-based method (Section 5) to estimate the cardinality of each hash table, allowing us to develop h_t in a way that reduces hash collisions during the build. In case we want to migrate a bucket to a different NUMA node, we can update its associated value in the bitmap. In Figure 2, we show an example where a record with a key value of '100' is first hashed to the partition of the hash table in NUMA node 1, and then hashed to its matches within that partition.

5 Cost Estimation

In this section, we first model memory access costs for build and probe. We then extend a sampling-based method for join size estimation to our formulated memory access cost estimation, and implement it in a lightweight manner.

5.1 Modeling memory access costs

We model the memory access cost at the granularity of hash buckets. In our model, records with the same hash value associate with the same hash bucket. The memory access cost of the build is modeled in Equation 1, where b_x refers to a single hash bucket set with a size of $|b_x|$ records, and BW refers to the memory bandwidth of the underlying NUMA node. This models the process of reading records from the memory and writing them to their corresponding hash bucket set in the memory.

$$T_{build}(b_x) = \frac{2|b_x|}{BW} \tag{1}$$

The memory access cost of a probe is modeled in Equation 2, where r_x, and w_x refer to records that are hashed to b_x from the input relation R and matched results found after comparing r_x with b_x, respectively. During the probe, all records from the probe relation R are hashed to their corresponding hash buckets. r_x is a set of such records hashed to the same bucket b_x in the hash table. For each record in r_x, it is compared with records in b_x for matches. Thus, the total number of comparisons for b_x and r_x is $|r_x| \times |b_x|$. For each comparison, there are two read operations, and at most one write operations.

$$T_{probe}(r_x) = \frac{2|r_x||b_x| + |w_x|}{BW} \tag{2}$$

5.2 A sampling-based optimization

It is costly to scan the entire input relations for the calculation of Equation 1 and 2. In the following, we formulate the estimation problem of the probe and adopt the state-of-the-art sampling-based estimation method to address it. This solution is extended to the build later.

The estimation problem in our case is similar to join size estimation, which counts a match between two records if their keys have the same value. The estimated join size is $\sum |R_v||S_v|$, where R_v and S_v refer to keys with the key value v from the relation R and S. This can be converted to our estimation modeled in Equation 2, because $|R_v||S_v|$ and $|r_x||b_x|$ have the same structure, and their values are close to each other if the key v is hashed to the bucket x with few hashing conflicts.

Join size estimation is a classical problem in databases. We adopt the state-of-the-art sampling-based join estimation method from Chen et al. [6] to estimate the cost in our case. This method first samples records from input relations of a join and then calculate the number of matches using their join size estimation model. In our case, the sampling process for relations S and R are slightly different. For both relations, each thread processes an equal-sized chunk of records and sample within its own local chunk. For S, because its hash table is the target of memory accesses during probe, the sampled records of each thread are merged together. For R, there is no need to merge sampled records from different threads because each thread processes its own chunk in the probe phase. After we apply the sampling technique, keys are sampled into a set of buckets for both relations. We then replace the estimation model [6] with Equation 1 and 2 for build and probe, respectively. This method also estimates the cardinality of each hash table partition.

6 Hash Table Placement

With memory access costs estimated, we determine the placement of hash buckets across all NUMA nodes. Given N hash buckets and P NUMA nodes, there are P^N different placement decisions for these buckets in total. P is determined by the undelrying hardware. N is configurable by tuning the hash function. If the hash function has larger fan-outs (i.e., larger N), we have more opportunities to schedule hash buckets for a balanced workload distribution. However, it also leads to a big solution space. We formulate this placement problem as an NP-hard partition problem and propose a heuristic-based solution as follows. The problem is to partition all the hash buckets to P subsets (P is the number of NUMA nodes in the hybrid memory architecture).

The hash buckets of subset i is stored in the NUMA node i. For a set of buckets, $B = \{b_0, b_1, b_2, ..., b_{N-1}\}$. Partition it to P subsets, $B_0, B_1, B_2, ..., B_{P-1}$, so that $\sum_j^{|B_0|} w_{0,j} = \sum_j^{|B_1|} w_{1,j} = ... = \sum_j^{|B_{P-1}|} w_{P-1,j}$, where $w_{i,j}$ is the estimated memory access cost of bucket b_j in the subset B_i.

For each hash bucket to be placed, we first apply the HBM-aware heuristic to determine whether it is suitable for the HBM or the main memory. Next, we further determine which NUMA node the hash bucket should be placed to within the scope of the determined memory type. Because errors in the memory access cost estimation may lead to sub-optimal hash table placement that hurts the performance, we also propose online migration and replication of buckets to further improve the placement.

6.1 HBM-aware Heuristic

Because of the differences between the main memory and the HBM, we first use a hardware-conscious heuristic to decide whether a bucket should be placed to the HBM or the main memory while building a hash table. There are two major differences that influence our decision here. Firstly, HBMs have wider memory buses and higher bandwidth for sequential memory accesses. Thus, accessing a dense array of buckets sequentially (a region in the hash table while most slots are occupied) can utilize more of the high bandwidth of the HBM. Secondly, the HBM is much smaller than the main memory capacity, and both of them have similar memory access latency.

Thus, we develop an HBM-aware heuristic. We sort all buckets (b_x) by their estimated cost (w_x) and then assign a bucket to the HBM if the accumulated cost of buckets in the HBM is within the α (workload dividing ratio) percentage of the total cost, in a descending order of estimated costs. The buckets assigned to HBM should not exceed the size of HBM. The rest buckets are assigned to the main memory. Thus, our cost model considers the size limitation of HBM as well as the difference in performance between HBM and main memory.

We formulate the value of α as the minima of the cost function of the build and probe, with which the total cost is minimize. Assuming that the relation S is initially in the main memory, the costs of build and probe are shown in Equation 3 and 4, respectively. $|B|$ is the total size of hash buckets to be visited. BW_D^{build} (BW_D^{probe}) and BW_H^{build} (BW_H^{probe}) are the measured memory bandwidth while building (probing) a hash table on the main memory alone, and the HBM alone, respectively. $|Out|$ is the size of all results. The α value shall be the critical point of minimizing the value of the corresponding cost function. Equation 3 calculates the larger one of the two costs on the main memory and the HBM, since they can work simultaneously. Equation 4 also includes the cost of outputting the results. For a simple hash join, its cost function f_{join} equals to $f_{build} + f_{probe}$.

$$f_{build} = max\{\frac{1-\alpha}{BW_D^{build}}, \frac{\alpha}{BW_H^{build}}\}|S| \tag{3}$$

$$f_{probe} = max\{\frac{1-\alpha}{BW_D^{probe}}, \frac{\alpha}{BW_H^{probe}}\}(|R| + |B| + |Out|) \tag{4}$$

6.2 NUMA-aware Heuristic

After we decide the suitable type of memory for buckets using the HBM-aware heuristics, we consider several heuristic algorithms to further decide the destination NUMA node for each hash bucket including dynamic programming and many approximation algorithms [19]. We find that the greedy algorithm delivers the best result in a similar time budget compared with other algorithms. Thus, we choose it as the NUMA-aware heuristic algorithm in this solution, as introduced in Algorithm 1. It first sorts all buckets by their associated workload, and then assign buckets to the subset with the lowest total workload in the descending order. The total workload of node j is maintained in Sum_j. This heuristic tends to separate large buckets and place them across different nodes at the beginning. Then, we allocate the hash buckets to NUMA nodes so that NUMA nodes have a balanced distribution. Particularly, we constantly add smaller buckets to the node with the lowest sum until it is full. This algorithm can be easily applied on a NUMA architecture with more than two nodes.

6.3 Online Replication and Migration

Because of potential inaccurate memory cost estimations, there are possibilities that a bucket that should be placed to

```
input   : $P$ NUMA nodes. $U^S$, the set containing all
          buckets from relation $S$. $W^S$, the set containing
          the estimated memory access cost of each
          bucket from relation $S$.
initialize $Sum_j$ of node $j$ to 0.
sort buckets in $U^S$ by their costs in $W^S$.
foreach bucket $i$ in $U^S$ in the descending order of their
  estimated cost do
    choose node $x$ with the smallest $Sum_x$.
    $Sum_x = Sum_x + w_i$.
    place bucket $i$ to node $x$.
end
```

Algorithm 1: NUMA-aware placement algorithm

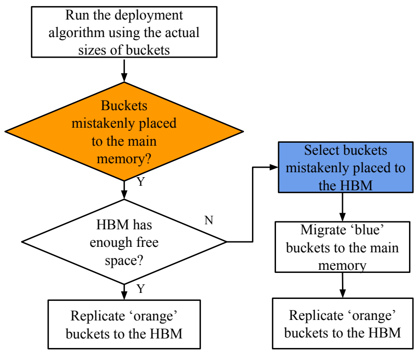

Run the deployment algorithm using the actual sizes of buckets
Buckets mistakenly placed to the main memory?
Y
HBM has enough free space?
N
Y
Replicate 'orange' buckets to the HBM
Select buckets mistakenly placed to the HBM
Migrate 'blue' buckets to the main memory
Replicate 'orange' buckets to the HBM

Figure 3: The online process of replications and migrations

the HBM are mistakenly placed to the main memory. Such placements may lead to an imbalanced distribution of memory access costs. After building the hash table, we know the exact size of each hash bucket. We further propose to replicate or migrate hash buckets between the main memory and the HBM in the runtime in order to fully utilize their bandwidth.

Figure 3 illustrates the decision-making process for online migrations and replications. We start with running the deployment algorithm again using the actual sizes of all buckets. Then, we check whether there are buckets that are mistakenly placed to the main memory, which are labeled as *orange* buckets in the figure. For such buckets, if there is enough free space in the HBM, we decide to replicate them to the HBM. In case the HBM does not have enough space, we select those buckets that are mistakenly placed to the HBM first, which must exist because the deployment algorithm determines that there is one mistakenly placed bucket in the HBM for one mistakenly placed bucket in the main memory. Such buckets are labeled as the *blue* ones in the figure. We then migrate these blue buckets to the main memory and then replicate 'orange' buckets to the HBM.

Table 2: List of hash join variants

Variant	HBM mode	Sources
	Simple hash joins	
SHJ (DDR)	DDR only	[1, 7, 28]
SHJ (Cache)	HBM configured as a cache	[1, 7, 28]
SHJ (Int.)	Data interleaved across NUMA nodes	[1, 7, 28]
HSHJ	All nodes managed by the proposed optimizations	This paper
	Partitioned hash joins	
PHJ (DDR)	DDR only	[1, 7, 28]
PHJ (Cache)	HBM configured as a cache	[1, 7, 28]
PHJ (Int.)	Data interleaved across NUMA nodes	[1, 7, 28]
HPHJ	All nodes managed by the proposed optimization	This paper

For all migrations from the HBM to the main memory, we execute them before the probe in a hash join. For replication, when a bucket marked to be replicated is probed for the first time, we place it to the hash table in the NUMA node of the HBM with the smallest sum of the workload in the greedy fashion. We then update the bitmap (as shown in Figure 2) accordingly and insert its records to the hash table using the same hash function h_t, so that further probes can find these buckets in the HBM.

7 Evaluation

7.1 Experimental setup

Testbeds. Die-stacked memory has been implemented on some emerging processors such as Intel KNL, and will be more common in future processors, such as AMD Ryzen APU. Still, it is a hot research topic on how to integrate die-stacked memory into the future processors [33]. Therefore, in this study, we use two complementary test beds for our evaluations: the real system implementation on Intel KNL processor which has die-stacked memory on-chip, and simulations to study the impact of future die-stacked memory enabled memory systems.

We conduct experiments on two complementary platforms as introduced above. The KNL CPU is Xeon Phi 7210, with 64 cores running at 1.30 GHz (256 threads) and 16GB HBM. The system has 96GB DRAM. On KNL, there are eight NUMA nodes, where nodes 0-3 and 4-7 represent the main memory and the HBM, respectively. All the cores are evenly distributed in nodes 0-3, while nodes 4-7 contain the HBMs only.

Our simulation is based on a state-of-the-art cycle-accurate DRAM simulator called Ramulator from Kim et al. [17]. We extend Ramulator to support multi-socket CPUs with die-stacked HBMs. The adapted simulator enables us to explore different architectural features of future HBMs which existing hardware platforms do not support. Our simulations capture the hardware features of HBMs beyond KNL. For example, we simulate different memory types, socket architectures and memory latencies/throughput. Thus, the simulator can be used in future studies of databases on the hybrid memory architectures. The implementation details about simulator can be found in the Appendix.

Implementation details. We apply the deployment algorithm to the state-of-the-art simple hash join (SHJ) and partitioned hash joins (PHJ), and compare them with other state-of-the-art hash join implementations. All hash join variants considered in this paper are listed in Table 2. We have used the latest version of the code available in the previous studies. For all implementations, existing software optimizations such as prefetching, software-managed buffers, multi-pass partitioning have been applied and tuned for their best performance, following the previous studies [2, 7]. All algorithms are vectorized using AVX-512 SIMD instruction set [7, 28]. For example, SIMD arithmetic instructions calculate the hash values of 16 keys altogether, and SIMD gather/scatter instructions are applied to resolve hashing conflicts [28].

For both SHJ and PHJ, we execute them using three different HBM modes: (1) the DDR only mode in which only the main memory is used, (2) the cache mode in which the HBMs are configured as an LLC, which is transparent to the software, and (3) the flat mode where the HBMs are exposed as NUMA nodes as part of the memory space and managed by the GNU NUMA utility *numactl*. This utility is able to interleave data across all NUMA nodes attempting to balance the workload. Finally, we apply our proposed optimizations on SHJ and PHJ, resulting in two variants: HSHJ and HPHJ. We take a focus on evaluating simple hash joins, especially on the simulated platform, which involve remote random memory accesses.

To reduce the sampling overhead, we vectorize the sampling algorithm using AVX-512 SIMD intrinsics and parallelize in multiple threads. The parallelization of the sampling algorithm is straightforward. Each thread takes a chunk from input relations, executes the sampling algorithm on this chunk, and finally estimates the memory access costs.

The deployment algorithm consists of two heuristics. The HBM-aware heuristic needs to tune the parameter α to minimize the cost modeled by relevant cost functions. The NUMA-aware heuristic requires sorting all buckets and a single pass over sorted ones. We adopt the state-of-the-art SIMD sorting implementation here to realize the sort operation [28].

In partitioned hash joins, hash tables are built for each cache-resident partition, thus they do not need to be placed across NUMA nodes. However, we can apply the same deployment algorithm to partitions instead of hash buckets. This is meaningful because partitions are also subject to skewness in input relations. In this paper, we estimate the memory access cost of each partition and place them according to the deployment algorithm in the same way used for hash buckets.

Table 3: Workload characteristics

	Size (10^6 records)	**Key Distribution**
Small	$\lvert R\rvert = \lvert S\rvert$ =128	Uniform
Medium	$\lvert R\rvert = \lvert S\rvert$ =1024	Uniform
Large	$\lvert R\rvert = \lvert S\rvert$ =1536	Uniform
Skewed	$\lvert R\rvert = \lvert S\rvert$ =128	Zipf distribution in relation R

Workloads. In all experiments, input relations reside in the main memory. We perform equi-join queries on relations R and S (in the form of “SELECT R.key, R.payload, S.payload FROM R, S WHERE R.key == S.key”), which is the same with previous studies [16, 28, 32]. Matched results are materialized in the main memory. Both keys and payloads are random 32-bit integers. We use four types of workload as shown in Table 3 according to the needs of each evaluation, on which we vary the sizes of relations and the distribution of keys. The small-sized, medium-sized and large-sized workload are smaller than, equal to, and larger than the HBM capacity, respectively. We set the workload size in this way to evaluate how hash join variants perform with different memory space constraints from the perspective of the HBM capacity. For the skewed workload, we vary the zipf factor to evaluate how hash join variants cope with skewness and how our proposed optimizations improve workload balance given skewed keys.

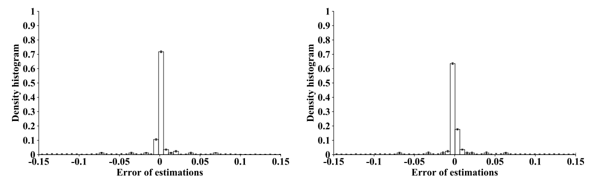
Result Outline. In this paper, we mainly present the results on real platforms, and leave the results on simulations to Appendix. Overall, the simulation results show that our design can significantly improve hash join performance on different hybrid memory architectures.

7.2 Evaluation of cost estimation

We now evaluate the accuracy of the cost estimation. Because the cost estimation provides inputs for other optimizations, its accuracy has a direct impact on how well the hash table deployment can perform and how often we need online migration and replication for hash entries. Overall, our cost estimation is sufficiently accurate and effective in guiding the hash table deployment. We use HSHJ as an example in this evaluation.

Figure 4 shows the histograms of estimation errors for two extreme cases where keys are either uniformly distributed (zipf factor=0) or highly skewed (zipf factor=2). Here, we define estimation error e_x for a sampled bucket as in Equation 5, where N is the total number of sampled buckets, w_x and w'_x refer to the estimated and the actual memory access costs of a sampled bucket x (x=0, 1, âĂę, $N-1$), respectively. We choose to use the relative difference because optimizing the workload balance is sufficient for our problem.

$$e_x = \frac{w_x}{\sum_x^N w_x} - \frac{w'_x}{\sum_x^N w'_x} \tag{5}$$



(a) zipf factor=0 (b) zipf factor=2

Figure 4: Histogram of estimation errors in HSHJ while processing the skewed workload

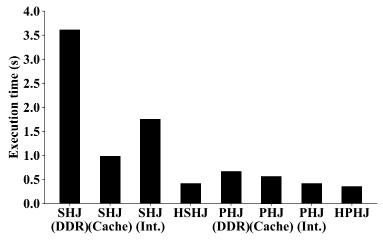


Figure 5: Execution time of hash join variants processing the small-size workload.

In Figure 4, more than 70% and 60% estimations have almost zero in these two cases, respectively. Most other estimations have very low errors. Similar results have been achieved when varying the zipf factor, which we omit here. We find that the adopted two-level sampling-based estimation method is largely accurate while estimating the costs in our case for the purpose of balancing the memory accesses.

7.3 Overall comparison

We first present an overall performance comparison between the state-of-the-art hash join implementations and our optimized hash joins in Figure 5.

We make the following remarks on the results.

Firstly, SHJ and PHJ behave differently in the flat mode using the NUMA utility. SHJ and PHJ become about 80% slower, and 16% faster than the cache mode, respectively. This shows that the partitioning phase helps to improve the performance by dividing the relations into equal-size partitions, which balances the workload across so many NUMA nodes. However, the NUMA utility is not able to distribute the workload of accessing a global hash table of the SHJ across NUMA nodes, resulting in a poor performance.

Secondly, regarding of our proposed optimizations, HPHJ and HSHJ are about 20% and 3 times faster than their fastest state-of-the-art algorithms, respectively. Although the proposed optimizations reduce the execution time for both algorithms, it is more impactful on non-partitioned simple hash joins. This is because our proposed optimizations help to balance the workload like the partitioning phase in the partitioned hash join, and all threads access local nodes after the partitioning phase.

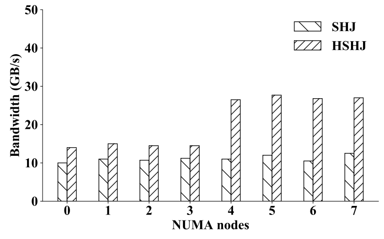


Figure 6: Average bandwidth utilization of HSHJ.

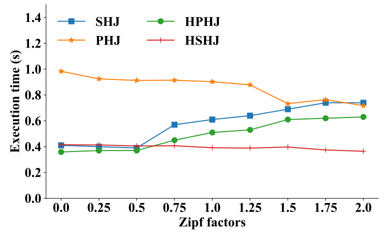


Figure 7: Execution time of hash join variants on skewed workloads

Bandwidth utilization. We compare the average bandwidth of SHJ and HSHJ per NUMA node in Figure 6. Please note that NUMA nodes 0 to 3 and nodes 4 to 7 consist of the main memory and the HBM, respectively. SHJ only utilizes about 10 GB/s and 11 GB/s bandwidth on the main memory and the HBM, respectively. HSHJ increases the bandwidth per node to about 15 GB/s and 27 GB/s for both memories, respectively. These results demonstrate that it is indeed helpful to place the data and threads carefully in the HBM and the many-core processor.

Impact of skewness. We show the impact of skewness on the overall performance in Figure 7. Skewness in input relations influences cost estimations and hash table placement. Here, we consider the fastest baselines from SHJ and PHJ as found out in Figure 5, and compare them with our optimized hash join variants. Keys in the input relation *R* follow the zipf distribution. We vary the zipf factor from 0 (no skewness) to 2 (high skewness).

As relations become more skewed, SHJ generally becomes faster and PHJ becomes slower. This is caused by the increasing data locality of skewed hot keys. Overall, HPHJ is the fastest when the zipf factor is smaller than 0.75. HSHJ outperforms HPHJ and becomes the best among all the four algorithms when the zipf factor is larger than 0.75. Also we have two more observations. First, comparing PHJ and HSHJ, HSHJ is more resilient to skewness and is about 40% faster than PHJ when the zipf factor is 2. Second, comparing SHJ and HSHJ, HSHJ is at least 2 times faster for all cases.

Impact of workload sizes. In Figure 8, we show the execution time of simple hash join variants processing the three classes of workload introduced in Table 3. We omit results

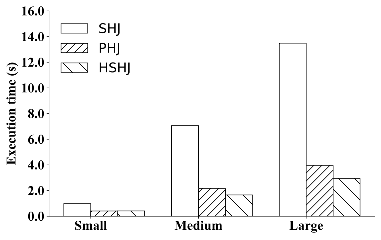


Figure 8: Execution time of different hash join variants processing the small-size, medium-size and large-size workload.

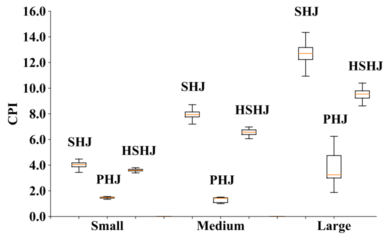


Figure 9: Distribution of CPI per core while executing different hash join variants.

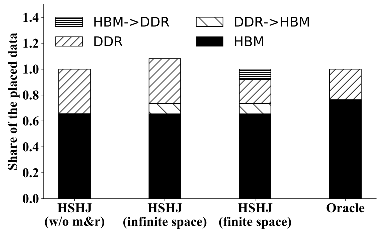


Figure 10: Breakdown of the hash table placement for HSHJ processing the small-size workload

from partitioned hash joins because our proposed algorithms are not impactful for them with uniformly distributed keys as explained above. HSHJ is 2.39, 3.60, and 4.60 times faster than the SHJ for the three classes of workload, respectively. This ascending trend of speedups shows that HSHJ scales better than SHJ when the size increases. Meanwhile, compared with PHJ, HSHJ performs 29% and 34% faster for medium- and large-size workload, respectively. While the partitioning overhead increases with the input sizes, our optimized HSHJ can gradually outperform PHJ.

In Figure 9, we show the box plots of CPI (cycles per instruction) per core measured while executing hash join variants processing the three classes of workload, respectively. The box plot is able to tell whether the memory access costs are uniform across different cores. Compared with SHJ, HSHJ manages to reduce the CPI per core by 11%, 18%, and 25% in these three classes, respectively. Meanwhile, CPIs of all the cores in HSHJ are much closer to their average, demonstrating the impact of the proposed optimizations on improving workload balance.

For simple hash joins, CPIs generally rises with the workload size. It is not surprising that partitioned hash joins have lower CPIs than simple hash joins, due to sequential memory accesses in the partitioning phase and data locality in the build and probe phases. Different to simple hash joins, CPIs of partitioned hash joins only increases significantly when the workload size exceeds the HBM's capacity. This is because the HBM is incapable to store enough partitions in order to get a share of the workload that is proportional to its high bandwidth.

7.4 Impact of individual techniques

Impact of the hash table placement. To evaluate the efficiency of the proposed hash table placement algorithm, we compare its output with what we derived in the oracle case throughput this section, where we apply the optimal hash table placement experimentally that minimizes the total execution time. In Figure 10, we break down the distribution of hash entries between the main memory and the HBM. For HSHJ, we consider three scenarios (1) "HSHJ (w/o m&r)" where the online migration and replication is disabled, (2) "HSHJ (infinite space)" where we assume the HBM has infinite free space so that no migration is needed, and (3) "HSHJ (finite space)" where assume no more free space is available after the build phase thus we need to migrate some buckets from the HBM to the DDR. In the Figure, "DDR" and "HBM" means the data is placed in the corresponding memory and stays there during the execution. "HBM⟶DDR" and "DDR⟶HBM" means the data is migrated to the DDR and replicated to the HBM during the runtime, respectively.

Comparing the HSHJ (w/o m&r) and the oracle case, our hash table placement method tends to place more hash buckets to the DDR, which implies that some buckets are underestimated so that they are not qualified to be placed to the HBM. Once we enable the online migration and replication, about 10% of the total buckets are replicated to the HBM in HSHJ (infinite space). With a tight capacity constraint enforced, some buckets are migrated to the DDR in HSHJ (finite space) to make space for such replications. These results are in line with the accuracy of the cost estimation which determines that the majority of hash buckets are well placed.

These placement decisions result in different overall execution time. In Figure 11, we show the execution time for each corresponding scenario. We can see that HSHJ (infinite space) space is the closest to the oracle case, showing the little overhead of online replications. With the capacity constraint, HSHJ (finite space) becomes 11% slower. However, it is about 10% faster than HSHJ (w/o m&r), showing the benefits of the online migration and replication method.

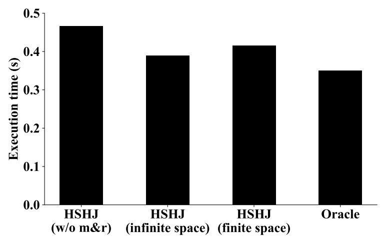


Figure 11: Execution time of HSHJ with different placement scenarios

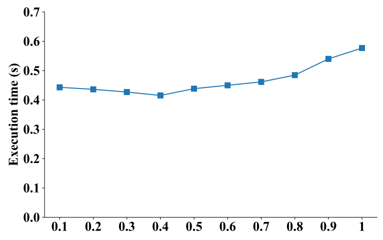


Figure 12: Execution time of simple hash join with different threshold ratio for a bucket to be replicated to the HBMs.

Sensitivity study of online migration and replication. In Figure 12, we show the execution time of the simple hash join with different threshold ratio for a bucket to be placed to the HBMs. While varying this ratio from 0 to 1, we find that the best performance is achieved when the ratio is 0.3, where we replicate a mis-estimated bucket to the HBMs if its memory access cost is larger than the 30th percentile among all buckets. Before and after this ratio, either more buckets or fewer buckets are replicated to the HBMs, resulting in a less balanced distribution and increasing the execution time.

Impact on workload balance. We evaluate the workload balance in HSHJ in Figure 13 by comparing the distribution of memory accesses in the oracle and HSHJ. These results form another view of what is presented in Figure 10 by showing the distribution of memory accesses across NUMA nodes. Although both the oracle and HSHJ achieve a roughly balanced distribution within each type of the memory using the hash table placement algorithms, HSHJ with and without online migrations and replications (denoted as "m&r") utilize more of the main memory and less of the MCDRAM. This is because some buckets that are placed to the HBM by the oracle are underestimated by HSHJ which results in placing them in the main memory. However, the online migrations and replications have improved the balance and made HSHJ closer to the oracle.

7.5 Overhead of the deployment algorithm

We compare the performance of HSHJ and its oracle case, which is derived by replacing the estimated cost with the real

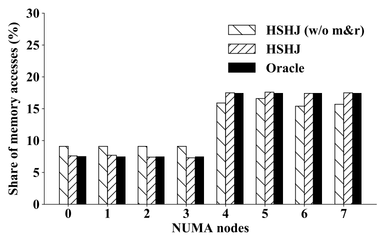


Figure 13: Workload distribution across eight NUMA nodes

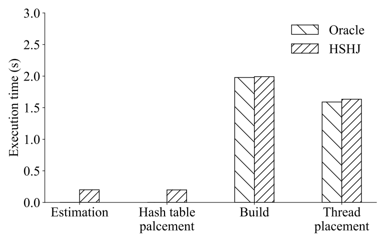


Figure 14: Time breakdown

costs measured in the runtime and removing the overhead of the hash table placement from the total execution time. We break down the execution time of them in Figure 14. The overhead of our optimizations is very low because both the adopted two-level sampling estimation method and the hash table placement are highly parallel. The probe phase of the oracle is slightly faster than HSHJ because of HSHJ's online migration and replication overhead. Overall, HSHJ is about 15% slower than the oracle.

8 Conclusions

The emerging die-stacked HBMs have introduced significant opportunities given their much higher memory bandwidth compared with the main memory. A hybrid memory architecture comprising both memory types is promising with challenges on improving database performance. As a starting point, we study building and probing hash tables, which are particularly challenging to exploit because of expensive random memory accesses that are difficult to predict and different types of memories in a single system. Specifically, we propose a new deployment algorithm for hash tables on this hybrid memory architecture of die-stacked HBMs and apply it to hash joins. Our evaluations on both a real hardware platform and a simulated platform show that the proposed deployment algorithm is effective in the workload balancing and performance improvements for hash joins.

Acknowledgments

This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824), Tier 2 grant (MOE2017-T2-1-122) and the National Research Foundation under its IDM Futures Funding

Initiative in Singapore. Dr. Eric Lo's work is supported by Hong Kong General Research Fund (14200817, 15200715, 15204116), Hong Kong AoE/P-404/18, Innovation and Technology Fund ITS/310/18. We thank Intel for hardware accesses and donations.

References

[1] Cagri Balkesen and et al. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*.

[2] Cagri Balkesen and et al. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96.

[3] R. Barber and et al. 2014. Memory-efficient Hash Joins. *PVLDB* 8, 4 (Dec. 2014), 353–364. https://doi.org/10.14778/2735496.2735499

[4] Ronald Barber and et al. 2014. Memory-efficient hash joins. *PVLDB* 8, 4 (2014), 353–364.

[5] Claude Barthels and et al. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 517–528.

[6] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 759–774.

[7] Xuntao Cheng and et al. 2017. A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. In *CIKM*.

[8] Biplob Debnath and et al. 2016. Revisiting Hash Table Design for Phase Change Memory. *SIGOPS Oper. Syst. Rev.* 49, 2 (Jan. 2016), 18–26. https://doi.org/10.1145/2883591.2883597

[9] Andi Drebes and et al. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *PACT*.

[10] Jana Giceva and et al. 2014. Deployment of query plans on multicores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 233–244.

[11] Mike P. (Intel). 2016. An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing .

[12] Mike P. (Intel). 2016. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[13] "JEDEC Solid State Technology Association". 2014. WIDE I/O SINGLE DATA RATE (WIDE I/O SDR), JESD229. https://www.jedec.org/system/files/docs/JESD229.pdf.

[14] "JEDEC Solid State Technology Association". 2015. HIGH BANDWIDTH MEMORY (HBM) DRAM, JESD235A. https://www.jedec.org/system/files/docs/JESD235A.pdf.

[15] Djordje Jevdjic and et al. 2013. Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 404–415.

[16] Saurabh Jha and et al. 2015. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB* (2015).

[17] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2016), 45–49.

[18] Richard E. Korf. 1998. A complete anytime algorithm for number partitioning. *ARTIFICIAL INTELLIGENCE* 106 (1998), 181–203.

[19] Richard E Korf. 2009. Multi-Way Number Partitioning.. In *IJCAI*. 538–543.

[20] Reinhard Kutzelnigg. 2006. Bipartite random graphs and cuckoo hashing. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 403–406.

[21] Viktor Leis and et al. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.

[22] Feilong Liu and et al. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*.

[23] J. Macri. 2015. AMD's next generation GPU and high bandwidth memory architecture: FURY. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. 1–26. https://doi.org/10.1109/HOTCHIPS.2015.7477461

[24] Darko Makreshanski and et al. 2018. Many-query Join: Efficient Shared Execution of Relational Joins on Modern Hardware. *VLDBJ* (2018).

[25] M. R. Meswani and et al. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *HPCA*.

[26] J Thomas Pawlowski. 2011. Hybrid memory cube: breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem. In *Hot Chips*, Vol. 23.

[27] W Wesley Peterson. 1957. Addressing for random-access storage. *IBM journal of Research and Development* 1, 2 (1957), 130–146.

[28] Orestis Polychroniou and et al. 2015. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*.

[29] Iraklis Psaroudakis and et al. 2015. Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1442–1453.

[30] S. Ramos and T. Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *IPDPS*. 297–306. https://doi.org/10.1109/IPDPS.2017.30

[31] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proc. VLDB Endow.* (2015).

[32] Stefan Schuh and et al. 2016. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*.

[33] M. J. Schulte and et al. 2015. Achieving Exascale Capabilities through Heterogeneous Computing. *IEEE Micro* 35, 4 (July 2015), 26–36. https://doi.org/10.1109/MM.2015.71

[34] Anil Shetty, Josephine Suganthi, and Prakash Khemani. 2014. Systems and methods for distributed hash table in a multi-core system.

[35] Ahsen J Uppal and Mitesh R Meswani. 2015. Towards workload-aware page cache replacement policies for hybrid memories. In *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 206–219.

[36] Jeffrey Scott Vitter. 1983. Analysis of the search performance of coalesced hashing. *Journal of the ACM (JACM)* 30, 2 (1983), 231–258.

[37] Stavros Volos and et al. 2016. *An Effective DRAM Cache Architecture for Scale-Out Servers*. Technical Report. Technical Report, MSR-TR-2016-20, Microsoft Research.

[38] Xiangyao Yu and et al. 2017. Banshee: Bandwidth-Efficient DRAM Caching Via Software/Hardware Cooperation. *arXiv preprint arXiv:1704.02677* (2017).

Appendix A: Implementation of Test beds

Die-stacked memory has been implemented on some emerging processors such as Intel KNL, and will be more common in future processors, such as AMD Ryzen APU. Still, it is a hot research topic on how to integrate die-stacked memory into the future processors [33]. Therefore, in this study, we use two complementary test beds for our evaluations: the real system implementation on Intel KNL processor which has die-stacked memory on-chip, and simulations to study the impact of future die-stacked memory enabled memory systems. Our simulation is based on a state-of-the-art cycle-accurate DRAM simulator called Ramulator from Kim et al. [17]. We extend Ramulator to support multi-socket CPUs with die-stacked HBMs. The adapted simulator enables us to explore different architectural features of future HBMs which existing hardware platforms do not support. Our simulations capture the hardware features of HBMs beyond KNL. For example, we simulate different memory types, socket architectures and memory latencies/throughput. Thus, the simulator can be used in future studies of databases on the hybrid memory architectures.

In the following, we present the implementation details of our simulation test bed, and leave the details of real hardware test bed in the experiment section.

The adapted simulator. The current Ramulator assumes a single socket multi-core CPU and a single memory controller [17]. Each type of memory is represented as a class (implemented in C++) with customizable parameters on timing and memory status transition etc. Its original implementation includes some common memory types such as DDR3, DDR4, GDDR5, and HBM.

In this study, we extend Ramulator to simulate multi-socket CPUs and the latest HBM2 standard. Figure 15 shows the overview of the adapted simulator. We make the following changes to the original Ramulator.

Firstly, we extend the support of HBM with different latency and bandwidth configurations in Ramulator. Particularly, we introduce a new die-stacked HBM model, the HBM2 (the 2nd generation of the HBM), to the simulator by adjusting the parameters of the previous generation of the HBM represented in its classes of current simulator. We acquire the parameters for the HBM2 standard from the corresponding JEDEC standard, JESD235A [14].

Secondly, in order to support the scaling of future hybrid memory systems, we extend Ramulator to support multiple controllers, with each memory controller associated with an independent NUMA node consisting of a specified memory type such as DDR4 or HBM. This simulates a machine with both the main memory and the HBM distributed across NUMA nodes in multiple sockets. In the implementation, we add a *NUMA node scheduler* in the simulator that diverts memory accesses recorded in the trace file to corresponding NUMA nodes. The NUMA node scheduler takes the memory addresses from the trace as inputs and directs each memory access to the memory controller that is linked to the corresponding NUMA node. In the original Ramulator, there are no remote memory accesses, thus all memory access latencies are the same. In the adapted simulator, we prepare a *latency matrix* in this NUMA node scheduler. For a memory access issued to a remote NUMA node, its latency is set according to this matrix to simulate the NUMA effect. This matrix is determined by the underlying NUMA topology which we have measured in selected multi-socket CPUs to simulate the NUMA effect in existing processors.

Thirdly, we modify the logic of each core so that it can issue requests to multiple memory controllers and multiple

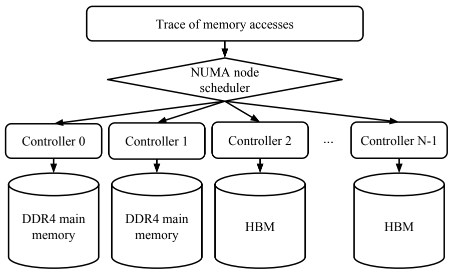


Figure 15: Overview of the adapted simulator.

NUMA nodes, including both the local and remote ones. This is to simulate the multi-socket environment, which is common in modern servers. In this simulation, we are also able to study the impact of different NUMA topologies.

Memory traces. The adapted simulator uses the same format of memory traces as Ramulator. Although Ramulator assumes a single memory controller and a single type of memory, it does not change the unified memory space, and thus we can use the same trace for simulating the hybrid memory system of die-stacked HBM. The memory access traces are collected while executing on a real machine. In our study, we execute hash join implementations on a machine with an Intel KNL processor. No data structure is split among NUMA nodes in the traces.

Appendix B: Implementation of Test beds

We focus on SHJ and HSHJ, which achieve similar results with partitioned hash joins that are omitted. Work balancing for partitioned hash joins are less sensitive to changes in the underlying hardware due to better locality brought by partitioning.

Impact of multi-socket design. In Figure 16, we show the performance comparison of SHJ and HSHJ in terms of normalized cycles using different numbers of NUMA nodes of the DDR4 type (denoted as “D”) or the HBM type (denoted as “H”) within a single socket. Here, like Intel KNL, a single socket can have multiple NUMA nodes for main memory, and NUMA nodes for HBM too. For example, 1D-1H denotes that there are one DDR4 NUMA node and one HBM NUMA node in this socket.

We first observe that the overall performance scales well with the number of HBM nodes, resulting in significant speedup. The 4D-8H configuration is 13.25 and 12.6 times faster than the 1D-1H one for SHJ and HSHJ, respectively. HSHJ is 2.5 times faster than SHJ on the same 4D-8H configuration, which is similar to what we have observed on the Intel KNL processor. Secondly, using more HBM nodes (e.g., 1D-4H) rather than the DDR4 nodes (e.g., 4D-1H) always leads to performance improvements.

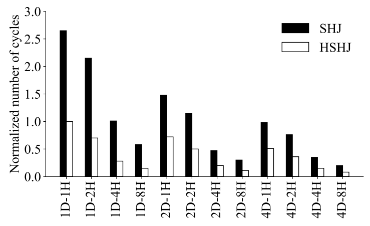


Figure 16: Performance comparison of SHJ and HSHJ using different numbers of main memory nodes and HBM nodes within a single socket.

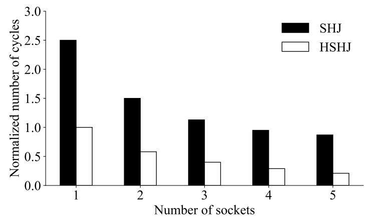


Figure 17: Performance comparison of SHJ and HSHJ using different numbers of 4D-8H sockets.

We further study benefits of more sockets in scaling in multi-socket CPUs. In Figure 17, we increase the number of sockets while each socket is a 4D-8H configuration. Although adding more sockets always lead to reductions in the total number of cycles, the improvement is marginally decreasing.

Impact of the HBM with different latencies. We conduct the experiments when the memory access latency for accessing an 8-byte record is varied from 200 ns (similar to the latency of the current HBM in KNL) to 50 ns (similar to the latency of L3 caches). This simulates the potential latency improvement of future die-stacked HBMs [15, 25]. We find that the overall performance of hash joins on the hybrid memory system is very sensitive to such improvements in latencies. The workload scheduled to the HBM varies from about 74% when the latency is at 200 ns, to about 87% when the latency is reduced to 50 ns. In the meantime, for both algorithms, reducing the latency from 200 ns to 50 ns has increased the performance by about 5 times. Thus, the low memory access latency of die- stacked HBMs are going to be very impactful in future systems. Detailed figures are presented in the supplementary file.

Impact of NUMA architectures. Figure 18 shows the workload distribution of HSHJ executed on a 4-socket machine, where we vary the NUMA topology from all-to-all to ring. In the ring topology, a NUMA node only has direct access to its adjacent neighbor along either the horizontal or vertical direction. A 2-hop transfer is needed for a node to access another node in the diagonal direction. In this experiment, all input relations are originally placed in node 0. We

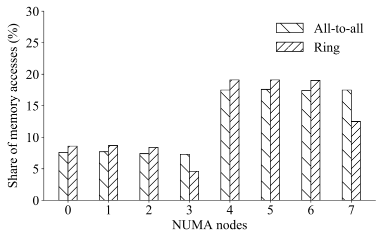


Figure 18: Workload distribution of HSHJ on a 4-socket machine with either the all-to-all or ring interconnect for the NUMA topology.

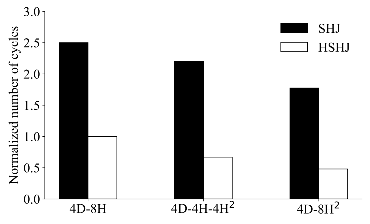


Figure 19: Performance comparison of SHJ and HSHJ using the HBM and the HBM2.

can observe from Figure 18 that our deployment algorithm adapts changes in the NUMA topology by scheduling less workload to node 3 and 7 (i.e., the diagonal neighbor of node 0 for the main memory and the HBM, respectively) in the ring topology. This is caused by the slower remote memory bandwidth of node 3 and 7 caused by the NUMA effect. This shows that our deployment algorithm can adapt to different NUMA typologies that help to mitigate the impacts of NUMA.

Impact of memory types. In Figure 19, we show that the proposed algorithm adapts to other types of HBMs, where H^2 denotes the HBM2 memory type. SHJ and HSHJ on 4D-4H^2 are 1.4 and 2.8 times faster than their 4D-4H baselines, respectively. This shows that our deployment algorithm can make better performance improvement on future HBM2.

Impact of the HBM with different latencies. Figure 20 shows how the workload dividing ratio α for the best performance of HSHJ changes when the memory access latency for accessing an 8-byte record is varied from 200 ns (similar to the latency of the current HBM in KNL) to 50 ns (similar to the latency of L3 caches). This simulates the potential latency improvement of future die-stacked HBMs [15, 25]. We find that the overall performance of hash joins on the hybrid memory system is very sensitive to such improvements in latencies. The workload scheduled to the HBM varies from about 74% when the latency is at 200 ns, to about 87% when the latency is reduced to 50 ns. In the meantime, for both algorithms, reducing the latency from 200 ns to 50 ns has increased the performance by about 5 times (Figures

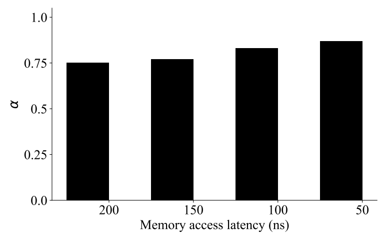


Figure 20: The impact of memory access latency on the workload dividing ratio α.

are omitted here). Thus, the low memory access latency of die- stacked HBMs are going to be very impactful in future systems.