Fast and Energy-Efficient OLAP Data Management on Hybrid Main Memory Systems


Published in:
IEEE Transactions on Computers

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
© 2019 IEEE. This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Download date: 11. Oct. 2019
Fast and Energy-Efficient OLAP Data Management on Hybrid Main Memory Systems

Ahmad Hassan, Dimitrios S. Nikolopoulos, Senior Member, IEEE, and Hans Vandierendonck, Senior Member, IEEE.

Abstract—This paper studies the problem of efficiently utilizing hybrid memory systems, consisting of both Dynamic Random Access Memory (DRAM) and novel Non-Volatile Memory (NVM) in database management systems (DBMS) for online analytical processing (OLAP) workloads. We present a methodology to determine the database operators that are responsible for most main memory accesses. Our analysis uses both cost models and empirical measurements. We develop heuristic decision procedures to allocate data in hybrid memory at the time that the data buffers are allocated, depending on the expected memory access frequency. We implement these heuristics in the MonetDB column-oriented database and demonstrate performance improvement and energy-efficiency as compared to state-of-the-art application-agnostic hybrid memory management techniques.

Index Terms—Non-volatile memory, hybrid main memory, database management system

1 INTRODUCTION

Online analytical processing (OLAP) is a key component of modern data processing pipelines and aims at analyzing large data sets to inform business decisions. OLAP consists of long-running, complex transactions that are read-intensive. To maintain high processing throughput, data sets are kept in main memory during OLAP processing to avoid I/O bottlenecks. This approach, known as in-memory processing, puts severe pressure on the required main memory capacity as data sets are continuously growing. It is estimated that data sets world-wide double in size every year [20]. As such, multi-terabyte main memory setups are desired [48].

DRAM, the dominant technology for main memory chips, has hit power and scaling limitations [22], [64]. DRAM-based main memory consumes 30-40% of the total server power due to leakage and refresh power consumption [3], [30], [34]. The background power consumption of DRAM moreover scales proportionally with the DRAM size, adding to the total cost of ownership (TCO). Moreover, it is uncertain whether DRAM technology can be scaled below 40nm feature sizes [22], [64]. It is thus unlikely that DRAM will continue to be the dominant memory technology. The techniques like High-Bandwidth Memory (HBM) and Intel’s Knight’s Landing processor provide ways of mitigating DRAM limitations [25] [53].

An alternative setup uses a combination of conventional DRAM and novel byte-addressable non-volatile memory (NVM) chips that are interface-compatible with DRAM [27], [28], [46], [47]. Emerging NVM technologies such as Phase-Change Memory (PCM), Spin Transfer Torque RAM (STT-RAM) and Resistive RAM (RRAM) [19], are compelling alternatives due to their high density and near-zero leakage power [8], [13], [37]. NVM technologies however have several drawbacks, broadly involving increased latency and increased dynamic energy for NVM accesses, reduced memory bandwidth and a fast wear-out of memory cells [8], [28], [46], [65]. On the plus side, NVM is predicted to scale to smaller feature sizes [8] and has around 100× lower static energy due to the absence of refresh operations [13]. Moreover, NVM is persistent, which is important to the design of DBMSs. This work, however, focuses on the orthogonal aspects of energy-efficiency and performance.

Hybrid memory systems typically have a fast and small component and a large and slow component, each with distinct energy consumption. In the case of a DRAM+NVM hybrid memory, DRAM provides high performance but it is not feasible to have large amounts of it. On the other hand, NVM can be scaled up to provide large amounts of memory, but it cannot be accessed as efficiently as DRAM. Hybrid memory systems raise the key question of how to distribute the data over the memory types?

State-of-the-art techniques propose data migration policies for hybrid memory where data migration is decided on by the hardware [28], [46], [53], [62], the operating system [12], [50] or a mixture of both [47]. These techniques try to second-guess the properties of the workload and migrate large chunks of data, typically at the page granularity of the virtual memory system. This reactive approach introduces runtime overhead and energy consumption due to monitoring and migration.

In this paper, we propose a radically different approach: Building on intricate knowledge of the structure and memory access patterns of column-oriented database, we propose that the DBMS manages data placement in the hybrid main memory. We present a methodology to analyze a DBMS in order to identify a good strategy for data placement. Our analysis uses a tool [18] that instruments all program variables and memory allocations and collects statistics on
their size, their lifetime and on the number of memory accesses made for each. Using these statistics, we formulate a strategy for data placement in a hybrid main memory system. In our solution the DBMS analyzes the query plan to predict which data sets will incur frequent memory accesses and should thus be placed on DRAM. It then allocates these data sets on DRAM as they are generated. By predicting the best placement with high accuracy, our solution does not require migration of data between DRAM and NVM to fix wrong data placements. This contributes to its performance and energy-efficiency.

The key contributions of this paper are:

- Derivation of runtime data placement heuristics for inputs and intermediate allocations of hash join using the access patterns of binary algebraic operators. The hash join is the single most important operation in our OLAP workloads. Our solution, optimized for OLAP workloads, requires only two heuristics for MonetDB [6], a state-of-the-art column-oriented database. Our approach is, however, not limited to MonetDB.
- Demonstrating the performance and energy-efficiency of our data placement heuristics for hybrid memory in MonetDB. We furthermore compare our heuristics to state-of-the-art application-agnostic page placement and migration techniques.
- Demonstrating that query plan partitioning yields intermediate data sets that are small enough to fit in DRAM without incurring significant overhead.

The remainder of this paper is organized as follows. Section 2 reviews the organization of memory systems and their key performance and energy properties. Section 3 describes the system organization assumed in this work and the programming interface. Section 4 analyzes what algebraic operators cause frequent memory accesses. Section 5 develops data placement heuristics for the most memory-intensive algebraic operators. Section 6 experimentally evaluates our solution.

2 BACKGROUND AND RELATED WORK

2.1 Column-Oriented Databases

A column-oriented database stores relational tables as a collection of individual columns rather than as a collection of rows [7], [55]. The columnar layout has proven specifically efficient on large, read-mostly data sets and on analytical workloads such as online analytical processing (OLAP) [7], [21]. The benefits of columnar databases arise from loading only relevant fields of tables into CPU caches. Fields that are not required for a query are stored on separate columns and are not accessed during evaluation of the query. Similar to record-oriented databases, columnar databases break down SQL queries in a query execution plan that explains how to evaluate the query using elementary operations (the algebra) as a function of the database state. A binary algebra [6] applies algebraic operations such as select (filter), join, union, etc, to two columnar data sets. The query optimizer transforms the initial query plan into an equivalent plan that is expected to execute faster. The query optimizer relies on heuristics, e.g., to apply filter operations as early in the query plan as possible in order to reduce data volumes quickly.

Columnar databases require materialization, which is the act of “gluing” together the columns that make up a relation [1]. Early materialization glues together all columns that are needed in the evaluation of the query. This then is similar to executing the query in the same style as a record-oriented database where records are initially shortened to contain only the columns relevant to the query. Late materialization postpones gluing together the columns as long as possible. Algebraic operators are not applied to the data columns themselves. Instead they operate on columns that hold the indices of the relevant records.

We focus our work on MonetDB [6], a highly efficient columnar database. MonetDB uses late materialization, which results in high memory locality [6]. Moreover, MonetDB evaluates each operator at a time, i.e., it processes the full input columns to produce the output(s) before proceeding to evaluate the next operator.

2.2 The Memory Hierarchy

The goal of cache memories (or caches for short) is to create the impression of a large and fast memory, where in reality large memories are slow and fast memories must be small. Cache memories create this impression using the principle of reference locality, an empirical observation that programs tend to repeatedly access only a limited amount of data during any period of time. This limited amount of data is called the working set of the program. Only the first access to the working set is slow. Subsequent accesses are fast as the working set is then available in the cache. We call a cache access that finds the data in the cache a cache hit. A cache access failing to find the data is a cache miss.

Caches are typically organized in multiple levels to form a cache memory hierarchy. These caches are searched one after the other until the data is found. The search starts with the cache closest to the CPU, which we also call the level 1 cache. This cache is the smallest and fastest. Subsequent levels are numbered accordingly as level 2 and level 3 cache and are increasingly larger and slower. The last level of cache is conveniently referred to as the last level cache or LLC.

When a cache miss occurs in the LLC, the missing cache block is fetched from main memory. Main memory accesses are off-chip, implying a high performance penalty in comparison to an on-chip cache hit. In the remainder of this work, we will focus on LLC cache misses, i.e., those load/store instructions that result in actual memory accesses.

When data is retained in on-chip caches, load/store instructions are served from the cached data and do not require main memory accesses. Consequently, small objects, which tend to be maintained in on-chip caches, may be allocated on NVM. This is true even if they are frequently accessed.

The replacement policy determines which data is maintained in the last-level cache, and which is evicted [52]. The least recently used (LRU) replacement is a common policy that replaces the least recently used cache block [52]. Instead of moving a block to the most recently used position...
TABLE 1: Latency (ns) and dynamic energy (nJ) for a 64-byte access and leakage power (mW/GB) for NVM technologies.

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Dyn. Energy</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R/W, [ns]</td>
<td>R/W, [nJ]</td>
<td>mW/GB</td>
</tr>
</tbody>
</table>

upon access, it is proposed to insert it in the LRU position, or in a median position [24], [45]. Adaptive policies select different insertion positions for workloads with mainly capacity misses compared to those with mainly conflict misses [24], [45]. While these policies aim to minimize cache miss rates, replacement policies have been proposed that minimize write-back traffic [15], [60]. These techniques are orthogonal to the present work as we aim to allocate data on appropriate memory types based on estimated capacity misses.

2.3 Memory Access Costs
Memory accesses incur costs in performance as well as in energy. These properties are relevant to all caches and also for main memory.

- **Latency** is the time that elapses between initiating a data access and the result returning in the CPU. Latency increases as data is found in cache levels further away from the CPU and increases sharply when main memory is accessed.

- **Bandwidth** is the volume of data that can be transferred per unit of time. Bandwidth decreases as one moves away from the CPU. Moreover, DRAM has higher bandwidth than NVM.

- **Static energy**, also called background energy, is consumed continuously regardless of whether the memory is accessed by software. Static energy consists of leakage energy in the memory cells and in peripheral logic. In DRAM, the memory cells must be refreshed frequently due to leakage. This contributes also to the static energy.

- **Dynamic energy** is incurred on a memory access. Dynamic energy consists of multiple components, depending on the state of the memory. It includes transfer of a command over the bus, opening a page, i.e., reading the contents of a row of memory cells into the row buffer, and selecting and sending data from the row buffer over the bus. After some time has elapsed, pages are closed which implies that the row buffer contents are written back to the actual memory cells, if modified.

It is reasonably accurate to reason about static and dynamic energy consumption of main memory using the capacity of the memory and the volume of data transferred [54].

2.4 Non-Volatile Main Memory Technologies
Table 1 shows key performance and energy consumption characteristics of some NVM technologies that are relatively close to market, i.e., they are in a good state of maturity and expected to ship in volume in the next few years. As such, precise characteristics of the memory chips are still unknown. However, regardless of the technology, it may be expected that reading and writing will take longer for NVM than for DRAM and will consume more energy, with writing suffering more than reading [28], [64]. Consequently, NVM is not a drop-in replacement for DRAM.

This paper focuses on the difference in memory access latency and bandwidth, as well as the trade-off between static and dynamic energy. These properties dictate that NVM can be used to build a large memory (due to the low static energy) but accessing the data stored in NVM is expensive both in time and energy. In contrast, DRAM can be accessed at a higher rate with less energy. However, building a large DRAM memory consumes a prohibitive amount of energy due to leakage and refresh operations.

These differences motivate the design of a hybrid main memory, featuring a relatively small DRAM component where the majority of read and write activity should occur, and a large NVM component where reads and writes are less frequent. DRAM thus gravitates towards the role of a cache, which is designed to hold the working set of an application.

2.5 Hybrid Memory Organization Options
CPU caches are typically physically organized as a hierarchy of caches where the highest-level cache is connected directly to the CPU data path using a wide and fast bus. The second-level cache is connected to the controller of the highest-level cache and is accessible only indirectly in the event of a cache miss in the highest-level cache. In contrast, in a hybrid memory system, DRAM and NVM are physically organized side-by-side and are connected to the CPU using a bus in a physically symmetric way. Nonetheless, DRAM can functionally be organized as a cache, where cache placement and replacement policies are in effect to control data allocation in DRAM (placement) and to evict data from DRAM (replacement).

A DRAM cache is governed by the same parameters and properties as CPU caches. However, data is moved in or out at a larger granularity. Typically the block size of a DRAM cache is comparable to the virtual memory page size. The DRAM cache can be managed either using hardware implementation of the placement and replacement policies, implementation of those policies in the operating system or in application software.

**Hardware cache management** implies that a hardware controller decides when to migrate pages between DRAM and NVM. Page migrations are expensive in both time and energy, which implies they must be performed with care. For instance, Lee et al [28] buffer a sequence of CPU writes to the same memory locations in DRAM before updating the main copy in NVM. This mitigates the high cost of NVM writes. Yoon et al [62] store those rows of data in DRAM that incur frequent misses in PCM row-buffers. They predict hit PCM rows that cause frequent row-buffer misses and move these rows to DRAM cache. Such caching policy shows 14% performance improvement and energy savings up to 10%. Their approach requires learning to predict hot data. These techniques require a directory to determine where data blocks are stored, which adds complexity.

**Operating systems** can implement page migration algorithms to decide which pages should be migrated to/from DRAM and NVM. These techniques require that the hardware monitors the frequency of memory accesses for each...
page. Dhiman et al. [12] show energy savings up to 30% and performance degradation up to 6% using these ideas. Shin et al. [50] adaptively group pages with similar access characteristics. They migrate similar pages in group to reduce overheads and demonstrated average energy savings up to 36% and less than 8% performance performance overhead. Li et al. [31] propose an OS and compiler-assisted approach to identify write-intensive parts of the application. In their approach the compiler adds new instructions to the application to guide hardware data distribution in hybrid memories. Ramos et al. [47] use multi-level queues to rank pages according to the benefit of storing them in DRAM. Their Rank-based Page Placement (RaPP) policy improves performance and reduces wear-out of NVM cells by minimizing the writes to NVM.

Application software can contribute to the placement and migration of data in hybrid memories. Effectively, the placement and replacement policies are now implemented in software in an application-specific way. From a hardware perspective, the DRAM and NVM components are now independently accessible. This moreover has the benefit that these policies can leverage insight in the future activity of the application provided that the application is well understood.

Wang et al. [59] optimize data placement using integer linear programming in the context of a real-time multitasking system. While theoretically tractable, they assume that each task accesses only one type of memory. This is unnecessarily restrictive. In [17], the memcached key-value caching program is modified to decide data migrations. The authors extended memcached’s replacement algorithm such that hot key-value pairs are retained in DRAM.

Hassan et al. [18] use off-line profiling information for each variable or memory allocation, called objects, in the application to decide a static placement. They do not perform migration during execution as they observed that objects in their applications are typically too short-lived to warrant migration. Moreover, they argue that placing objects in hybrid memories is more effective than placing virtual memory pages as the latter often contain several objects with distinct memory access patterns. Wei et al. [61] assume a similar setup but extend it with migration. They first place objects in the hybrid memory using the off-line profiled ratio of reads over writes for each object as a guideline. During execution, they again measure the read/write ratio and migrate pages as appropriate. The read/write ratio is however an inaccurate predictor for energy consumption as energy consumption relates linearly to the number of reads and writes [58]. It does not depend solely on the ratio of reads to writes.

This paper manages the DRAM cache from within the application, in our case the DBMS. When applicable, this approach has the highest potential. Application-agnostic techniques incur overhead to learn memory access patterns and migrate data between DRAM and NVM to correct a bad allocation. In contrast, we will demonstrate that we can accurately allocate data in a column-oriented DBMS immediately in the right type of memory by analyzing the query execution plan and estimating what operators will incur frequent memory accesses.

2.6 Persistence Aspects of NVM

The persistent nature of NVM is attractive to optimize transaction processing workloads. Coburn et al. [11] and Guerra et al. [16] provide programming API’s with ACID properties applicable to generic data structures such as trees and hash tables. Jorge et al. showed a speed-up of 83% for the SQLite application. Arulaj et al. [2] investigate how three different DBMS designs can exploit the persistence of NVM. Pelley et al. [42] reduce the frequency of persistence barriers by exploiting the byte-addressability of main memory NVM. We do not utilize the persistence properties of NVM in this work. Oukid et al. [40] proposed a database storage engine on top of DRAM and NVM to achieve data recovery through NVM persistence. Similarly, Oukid et al. [41] used NVM persistence to eliminate database logging mechanism in contrast to traditional database systems running on top of DRAM-only system. As such, the ideas above can be integrated with our techniques.

2.7 NVM Durability

Due to the changes in physical structure when writing NVM memory cells, NVM is typically less durable, i.e., it has a shorter lifetime than DRAM. The durability of NVM can be improved in various ways, including changes to the internal organization of memory chips [28], [46], encoding energy-efficient encoding of bit patterns based on the energy difference between writing a 0 (RESET) and a 1 (SET) [36], or by avoiding redundant writes where the new value equals the overwritten one [64]. These techniques will make NVM more attractive and less expensive. They are otherwise orthogonal to this work.

3 System Organization and Programming Interface

The programming interface exposes the hybrid nature of the main memory (Figure 1) to the programmer [18]. At the hardware level, the DRAM and NVM memory chips are assigned distinct physical address regions. This ties in with how the BIOS reports DIMMs and their physical address ranges to the OS. In this case, the BIOS additionally reports the technology used to implement the DIMMs. The OS thus knows which memory chips are DRAM and which are NVM and can map virtual memory pages accordingly.

The operating system allows the application programmer to decide if data should be allocated on NVM or on DRAM. The main mechanism to create new virtual memory pages is through the mmap system call. We add a new option to the flag argument of mmap to indicate whether the new pages should be mapped on DRAM or on NVM (default). The OS records the target memory request and allocates the physical page on DRAM if space is available. If space is not available, the physical page is allocated on NVM.

Applications often use finer-grain mechanisms for memory allocation such as the malloc family of functions. The malloc functions manage a set of virtual memory pages obtained through mmap and allocate small objects from these pages. In order to differentiate between DRAM and NVM allocations, we implement two distinct memory allocators in the application, one managing pages mapped to DRAM,
the other pages mapped to NVM. Each memory allocator provides the standard memory allocation functions prefixed with either “dram_” (e.g., dram_malloc) or “nvm_”. The standard function names are retained and are mapped to the NVM allocator.

The proposed solution is orthogonal to the Persistent Memory Programming (pmem.io), which exposes a set of user space libraries that provide APIs to allocate memory from within the applications while taking care of the persistence memory management [44]. Persistent memory builds on the Direct Access (DAX) feature of Linux enabling to run load/store instructions on a memory mapped file. The solution proposed in this paper is an extension of pmem.io, where the application memory allocators are aware of the hybrid memory and pmem.io is used to support load/store instruction access to NVM.

For the purposes of this work, it suffices to consider only dynamically allocated memory through malloc or malloc. We assume that static (global) variables and stack-allocated variables are mapped to NVM. These variables are typically small and have good CPU cache locality [9], [29]. As such, few main memory accesses are directed to these variables and placing them in NVM is energy-efficient and does not incur important additional memory access latency.

If migration is necessary, the programmer can allocate a new copy of the object on the opposite memory type and copy the data. In this paper, we find that migrations always coincide with existing memory copying code. This happens as migrations are performed only when data set sizes grow to exceed the LLC capacity. As data set sizes are hard to predict in advance, data is stored in incrementally growing buffers. Growing these buffers involves allocating a new buffer and copying the data over. We leverage these existing copies to avoid additional migration cost.

4 Analysis

Our model for utilizing hybrid memory requires to identify specific memory allocations, through malloc or malloc, that should be placed on DRAM. These objects should have a high number of anticipated off-chip memory accesses. In order to determine these objects, we analyse what data and code in the DBMS is responsible for the majority of main memory accesses.

The analysis described in the current section is to a degree specific to the type of DBMS that we analyse. We analyse, specifically, MonetDB, a highly efficient column-oriented database [6], [38]. The technique of the analysis, however, applies without modification to other DMBS. Our analysis consists of the following steps, automated by a tool:

1) We perform dynamic analysis of the DBMS through instrumenting all memory access instructions in the executable. We execute the instrumented DBMS on real hardware and track all memory accesses, heap allocations and de-allocations and function calls and returns. This analysis identifies all memory objects in the application, which are program variables and (heap) memory allocations. The dynamic analysis is implemented through instrumentation of the DBMS with a custom LLVM pass [32].

2) We focus on the hot objects, those with the highest number of main memory accesses, as determined through CPU cache simulation during dynamic analysis. The cache simulator is a simple, single-level cache simulator that models a 20 MB 4-way set-associate cache with 64-byte blocks and least-recently used (LRU) replacement. The simulator makes the simplifying assumption that a hierarchy of inclusive caches may be approximated by a single cache level consisting of the LLC [52].

3) We analyse the code that is responsible for creating and accessing the hot objects, which is identified by the dynamic analysis tool.

4.1 Column-Oriented Data Types

We have applied our dynamic analysis tool to MonetDB and identified the following types of objects, all of which are dynamically allocated:

- **Columns**: These store the columns of the relational database as it is mapped on the physical storage. All data within these data structures are backed by files on the physical storage. There is relatively little reuse in columns as they are mostly touched once per query in analytical workloads.

- **Intermediate data**: These are intermediate and final results sets that are computed by operators in the query execution plan. Intermediate result sets require further processing by operators in the query execution plan. Many small intermediate data sets are generated during execution of a query.

- **Schema Metadata**: These are database columns holding meta-data on the schema stored in the database. As the metadata describes the structure of the database, its volume is typically small in relation to the contents of the database.

DBMSs perform numerous other memory allocations. Any memory allocation not listed above has marginal relevance to the volume of memory addressed and the total number of off-chip memory accesses made.

4.2 Frequently Accessed Objects

We analyzed all objects created by MonetDB while executing the TPC-H queries. Objects incurring frequent main memory accesses should be placed on DRAM, while objects that
are infrequently read or written in main memory should be stored in NVM. Note that this property is not the same as frequency of load and store operations at the application level. The CPU caches filter out most of the load and store operations, such that only a small percentage actually reaches main memory. As such, our analysis depends on system properties. We assume that modeling only the last-level cache provides an accurate depiction of the cache hits and misses that would occur in a more detailed memory hierarchy with higher-level caches. This assumption results in sufficient accuracy for the purposes of our analysis. Moreover, it results in a significant speedup of profiling which allows us to profile the complete execution of the database. We assume a 20 MB last-level cache. We analyse MonetDB version 11.15.7 executing the TPC-H workload at scale factor 5.0. We have analyzed also scale factors up to 100 and remark that the conclusions presented here are valid at least up to scale factor 100.

Our measurements confirm that over 99% of the loads and stores are issued on the database objects listed in Section 4.1. These objects have, however, distinct characteristics. Some objects have very good cache locality while others have few overall accesses. We distinguish between two types of objects: objects storing the columns of the database and objects storing intermediate results. Figure 2 shows that the majority of off-chip accesses are incurred for intermediate results. The actual columns contribute on average to less than 10% of the main memory accesses. This is a logical consequence of query plan optimization where cache-efficient operators such as select are performed early on in order to reduce intermediate data set sizes. As such, columns tend to be accessed in a cache-efficient way.

Implications for placement: Most objects in MonetDB benefit from a default placement on NVM. E.g., objects storing schema metadata tend to be small and are accessed primarily during the construction of the query plan.

Our analysis shows that columns may be mapped to NVM with limited overhead. This is a helpful result because the columns hold the actual data in the database. As such, they may be very large compared to the available DRAM.

Objects holding intermediate data should be distributed between DRAM and NVM, as only a subset of these objects incur a high number of main memory accesses. Note that there is no guarantee that intermediate objects may actually fit within the amount of DRAM available. However, the sizes of intermediate data sets can be controlled by fragmenting the query execution plan [23]. We will demonstrate that this allows us to make excellent use of available DRAM.

4.3 Operators Causing Main Memory Accesses

Now we know what objects cause main memory accesses, we turn to the question what code is making those main memory accesses. We calculated a break-down of main memory accesses by type of operator executed. Because the results are highly skewed, we report results only on the join operators. Figure 3 shows what percentage of main memory accesses are incurred by join operations versus other database code. Over 90% of the main memory accesses are incurred by join operations. This result is in line with previous observations that join operations are the most critical and problematic database operations [7].

Figure 3 also shows the impact of query plan fragmentation. Query plan fragmentation is a technique to enlarge the degree of parallelism within the query plan [23]. It breaks down the columns in smaller fragments, resulting in more operations in the query plan that are independent of one another. In this work, we will use query plan fragmentation to constrain the size of intermediate result sets such that they fit in DRAM. Note that other databases may use other techniques to control parallelism and working set sizes. Our key interest is in limiting intermediate data set sizes such that they fit in DRAM. Figure 3 shows that breaking down columns in 50 fragments slightly reduces the percentage of main memory accesses concentrated in join operations. This effect is, however, negligible. Figure 4 shows that, for the 5 GB database, up to 150 fragments incur no noticeable performance overhead. The baseline is no fragmentation. At 200 fragments, the overhead increases sharply due to an increased number of operators in the query execution plan.

Practically, fragmentation is applied by identifying a limit to the size of partitions of columns. Fifty-way fragmentation results in fragments of columns of the TPC-H data set at scale factor 5 no larger than 80 MB. As such, 4 column fragment fit simultaneously in a 256 MB DRAM cache, which is the size assumed in our experimental evaluation. More importantly, intermediate data sets are typically much smaller, so many more can fit simultaneously in the DRAM cache.

Modern databases implement various join algorithms, each optimized to different boundary conditions [49]. The three most frequently occurring join operations in our analysis are: the fetch join (a selection-based join able to join tables with a single pass), the hash join (which summarizes the smaller relation and joins through lookup in the hash table) and merge join (which merges relations that are sorted on the join key).

Some types of join operations have, by nature of the algorithm, higher memory complexity than others. However, there is also a large difference between join operations of the same type. Figure 5 shows what percentage of each type of join operation are responsible for 99% of the off-chip memory accesses. Figure 5 gives three insights: (i) The
Fig. 3: Percentage of main memory accesses from join operators and other code in MonetDB Columnar Store. ‘NF’ corresponds to No Fragmentation and ‘F’ corresponds to Fragmentation.

Fig. 4: The overhead of query fragmentation on 4 TPC-H queries.

Fig. 5: The percentage of join calls serving 99% of off-chip memory accesses, broken down by the type of join algorithm. ‘NF’ corresponds to No Fragmentation and ‘F’ corresponds to Fragmentation.

The majority of the join calls find their data in on-chip caches, which means that their operands and output data set can have NVM as a backing memory; (ii) Hash join is the most critical join among them because over 25% of the hash join calls are memory-intensive; (iii) 10% of merge joins are memory-intensive, however, this number is strongly reduced when fragmenting the query execution plan. Note that merge join operates on sorted data sets and is thus linear-time. Smaller data sets are naturally retained in cache.

We have thus identified that the main objects in MonetDB that should be placed on DRAM are intermediate results that are used by hash join operators. Let us study the hash join operator more closely to understand its memory access patterns. Algorithm 1 depicts the structure of a partitioned hash join algorithm for equi-joins.

**Algorithm 1:** Pseudo-code of partitioned hash join algorithm for equi-joins.

- **input:** Relations \( R \) and \( S \) where \( S \) is the smaller relation.
- **output:** Relation \( D \) which is the join of \( R \) and \( S \).

initialize empty relation \( D \);
partition \( S \) in \( n \) parts;
for each partition \( S_i \) of \( S \) do
  build hash index \( H \) on \( S_i \);
  for every tuple \( t \) in \( R \) do
    bucket = lookup \( t \) in \( H \);
    for every tuple \( t' \) in bucket do
      emit joined tuple \((t,t')\) to \( D \);
  end
end
return \( D \);
processing a partition [26, 63].

We conclude that the majority of main memory accesses in the DBMS occur while executing hash join operators and that these memory accesses are associated to the larger relation that is joined. As such, memory placement is optimized when the larger relation is placed in DRAM and the smaller relation is placed in NVM. However, when both relations are sufficiently small they will simultaneously fit in the last-level cache. In this case, cache misses are rare and both relations should be placed in NVM.

5 Data Placement Heuristics

We develop the FEED algorithm (Fast and Energy-Efficient Data placement) for placing intermediate results used by join operators in DRAM. These intermediate results can be the output of the join operation, in which case we place them on the appropriate type of memory when the join operator is initiated. Placing data on the appropriate memory type at allocation time avoids the need for data migration, which can be costly in terms of execution time and energy consumption. Intermediate results can also incur a large number of main memory accesses when they are used as inputs to the join operator. In this case, we decide on placement of the result set when it is allocated, i.e., at the start of the operator that is producing the result set.

Algorithm 2 - FEED algorithm for placement of hash join output

```
ALGORITHM 2: FEED algorithm for placement of hash join output

input: Input data set IDs R and S
output: Placement decision for hash join output
N_R = tuple_count(R);
N_S = tuple_count(S);
/* Estimate number of tuples in result set */
N_E = min( max(N_R,N_S), 32*min(N_R,N_S));
/* Estimate total memory footprint in bytes */
footprint = N_R * tuple_size_in_bytes(R) + N_S * tuple_size_in_bytes(S) + N_E * tuple_size_in_bytes(result_set);
/* Place in NVM if it fits in the LLC cache */
if footprint < \lambda \cdot Size_{LLC} then
  return NVM-placement;
else
  return DRAM-placement;
end
```

5.1 Join Output

Algorithm 2 shows the procedure to decide the most appropriate backing memory type for the output of the join operator. We calculate the total memory footprint of the join operator by adding the sizes of the inputs to the estimated output size. The size of an individual result set is calculated as the product of the number of tuples and the width of a tuple (how many bytes are required to store the tuple). The size estimation contains a parameter (32) that was heuristically determined to maximize the accuracy of predicting the tuple size [7]. If the memory footprint exceeds the last-level cache (LLC) size then we place the output on DRAM, else we place it on NVM. The rationale behind this heuristic is that large outputs will be re-read from DRAM, while small outputs will be re-read from the last-level cache.

FEED accounts for inaccuracy in the estimates. We decide that the footprint exceeds the LLC capacity when it is larger than a fraction \lambda < 1 of the LLC size. The fraction \lambda guards against conflict misses (as opposed to capacity misses which are determined by the size of the cache [51]). Moreover, there are other small auxiliary data structures that also occupy LLC space. We experimented with values for \lambda in the range 80-100% and choose \lambda = 90%. Choosing a larger value of \lambda may inadvertently increase the cache miss rate and thus result in excess accesses to NVM. Choosing too small a value for \lambda will under-utilize NVM and increase power consumption by reverting accesses to DRAM.

Algorithm 3 - Placement heuristic for hash join inputs

```
ALGORITHM 3: Placement heuristic for hash join inputs

input : Input data set IDs R and S
output: Placement decision for R given that S is other input
if S not allocated yet then
  /* Case 1: Allocate 'R' when 'S' not yet created */
  if footprint = 3 * N_R * tuple_size_in_bytes(R);
  else
    return NVM-placement;
else
  /* Case 2: Allocate 'R' when 'S' already created */
  if footprint < \lambda \cdot Size_{LLC} then
    return NVM-placement;
  else
    return DRAM-placement;
end
```

We note that in our experiments the output of the join operator may also be reused internally within the operator under specific circumstances. This effect is specific to how the internal data structures are organized in the database we experimented with. In particular, data sets are invariably represented using (a number of) arrays, each stored sequentially in memory. When these arrays are allocated,
the database uses cost models to estimate their size. This is
dependent on the query and the database content. As such,
the estimate is conservative and there is a procedure to grow
the arrays. The arrays grow through allocating a larger array
and copying the data over. Arrays may be grown repeatedly
during the execution of an operator, causing the data to be
reused during execution.

Each time the output set is grown, we re-apply the
heuristic of Algorithm 2 using the updated size of the output
set instead of the initial estimate. As such, the output of a
join operator may initially be allocated on NVM because
the total footprint is small, but the output set may later grow
large enough to warrant allocation on DRAM. Such
migrations are rare and occur at most three per query. Note that
migrating the output from NVM to DRAM incurs minimal
overhead over the baseline database implementation as the
operation coincides with an existing memory copy.

Implementation consideration: Copying data from
NVM to DRAM may take longer compared to copying data
from DRAM to DRAM. The details depend strongly on
system design. On the one hand, NVM access latencies are
higher than DRAM latency, which suggests that DRAM-to-
DRAM copy will be faster than NVM-to-DRAM copy. It is,
however, likely that NVM and DRAM chips interface with
a distinct memory controller due to electrical and timing
differences between the memory types. As such, an NVM-
to-DRAM copy can avail of higher bus bandwidth and does
not incur changes in bus operation (swapping between read
and write), which cost a few bus cycles per switch. The
actual costs will become clear when hybrid memory systems
become available. We make conservative assumptions in our
evaluation in favor of DRAM systems.

5.2 Join Inputs

Most join operators access their inputs using a single se-
quential scan as shown in Section 4. An exception is the
hash join algorithm, which uses its inputs in distinct ways.
The larger input is repeatedly scanned over. For every tuple
in the larger input, the appropriate key is looked up in the
smaller input relation. In order to make this lookup fast, a hash index is computed on the smaller input prior
to the start of the algorithm. It is furthermore possible to
partition the smaller input and process one partition at a
time. This minimizes cache misses and TLB misses on the
hash index, but requires repeated sequential scans of the
larger input [33].

Knowing this behavior of the hash join algorithm, we
define a heuristic that places the larger input in DRAM and
the smaller in NVM. The hash index itself has excellent
locality by design of the algorithm and typically resides
in the CPU cache [6]. We aim to to place data on the
appropriate type of memory start at creation time. This
way, we can avoid migration of data, which can negatively
impact execution time and energy-efficiency. The placement
heuristic (Algorithm 3) considers two cases: (i) if the total
footprint of the join operator fits in the LLC, then we allocate
the inputs on NVM; (ii) otherwise, we allocate the smaller
input on NVM and the larger input on DRAM. In order
to make these decisions, we need to estimate the memory
footprint size of the join operator. The difficulty of this
estimation depends on whether one or none of the join
operator’s inputs have already been created. We consider
these cases separately.

5.2.1 Placement of Last-Created Input

This is Case 2 in Algorithm 3. We assume input \( R \) is created
and \( S \) already exists. We estimate the number of tuples that
\( R \) will contain, depending on the properties of the operator
that is generating \( R \) and its inputs. Note that this operator
is about to execute. As such, it has exact knowledge of its
inputs. As such, we do not need to estimate result sizes
recursively, which would have poor accuracy. On the basis of
the size of (estimated) input sizes for \( R \) and \( S \), we can then estimate
the footprint of the join output using the same formula as
Algorithm 2. We then proceed to place \( R \) on NVM if the total
footprint of the join operator fits in the cache. Otherwise, we
place \( R \) on NVM if it is the smaller relation.

5.2.2 Placement of First-Created Input

This is Case 1 in Algorithm 3. In this case we can estimate
again the size of the input \( R \), but for \( S \) we have no information
as it has not been created yet. Lacking any information,
we estimate the footprint of the join result on the basis that
both \( S \) and the result will have the same size as \( R \) (hence
the factor 3). Depending on whether this footprint fits in the
LLC, \( R \) is placed accordingly.

Note that this heuristic works very well when the inputs
have highly different sizes. Assume \( R \) is allocated first. If \( R \)
is small, it will be allocated on NVM. If the real size of \( R \)
is indeed much smaller than \( S \), then FEED will place \( S \) on
DRAM if \( S \) is sufficiently large. On the other hand, if \( R \) is
sufficiently large it will be allocated on DRAM. If the real
size of \( R \) is much larger than that of \( S \), then FEED follows
up with allocating \( S \) on NVM.

As FEED only places data and does not migrate, it re-
quires foresight to place the inputs of the hash join operator
in the correct type of memory when the inputs are created.
To this end, FEED annotates the query execution plan to
show every output value that is consumed by a hash join
operator. The operator passes this annotation on to the FEED
memory allocation algorithm when allocating memory to
store its output. Annotations can be inserted in the query
execution plan either during construction, or by performing
a linear-time post-processing step. Every operator is repre-
sented by a node in the query execution plan. Annotations
are applied by scanning the plan backwards and labeling
the nodes that feed into join operators.

5.3 Managing DRAM Contention

DRAM contention is mitigated through query plan fragmen-
tation. This causes intermediate data sets to be sufficiently
small such that all DRAM allocations can be simultaneously
satisfied. For some queries, however, this may not be suf-
ficient. An alternative is to actively manage DRAM con-
tention, by moving intermediate data sets between DRAM
and NVM when DRAM is full. We found, however, that
there is limited scope for such data movement.

Assume that at any moment in time a memory allocation
is characterized by \( A = (S, N) \) where \( S \) is the size in
bytes and \( N \) is the average number of reads per byte. We
assume that every allocated object is write-once, which is the case for the objects we desire to place on DRAM. Assume further that allocation $A_1$ was made in DRAM and is a possible candidate for eviction to NVM at the time when allocation $A_2$ is made. We can then estimate the average memory access time (AMAT) when retaining $A_1$ in DRAM and allocating $A_2$ in NVM as

$$N_1 S_1 T_{r,dram} + N_2 S_2 T_{r,nvm} + S_2 T_{w,nvm}$$  \hspace{1cm} (1)

where $T_{r,tech}$ and $T_{w,tech}$ are the average delay of reading, respectively writing, a 64-byte block of data in memory of technology $tech$. The AMAT when moving $A_1$ to NVM can be estimated as:

$$N_1 S_1 T_{r,nvm} + N_2 S_2 T_{r,dram} + S_2 T_{w,dram} + S_1 (T_{r,dram} + T_{w,nvm})$$  \hspace{1cm} (2)

Moving $A_1$ reduces AMAT when Eq. 2 < Eq. 1. Assuming $S_1 = S_2$ and using the technology parameters of RRAM, the best-performing NVM variant (Table 1), migration is beneficial when:

$$64N_1 + 120 < 64N_2$$  \hspace{1cm} (3)

which implies that $N_2$ must exceed $N_1$ about three times or more. We measured $N$, the expected number of times a byte in an object is read from main memory during the lifetime of the object. As one can see, reuses from main memory, are infrequent. When the objects with zero reuses are read, the reads hit in the on-chip CPU caches.

As there are few case where migration is beneficial (1 object in the case of the Q11 query, see Figure 6), we decided to develop strong placement heuristics and abstain from migration. While fragmentation of queries limits intermediate data set sizes, it cannot be ruled out that an allocation cannot be served from DRAM. Such allocations, which were unobserved in our experiments, have limited impact as it is rare to re-read objects from memory frequently.

While FEED implements placement in the hybrid memory hierarchy, this does not prohibit the programmer to additionally perform migration on selected objects. As migration requires a copy of data between DRAM and NVM, it can be effected by performing a fresh memory allocation on the target memory and then copying the data over.

![Graph showing number of main memory reads per byte in each heap-allocated object of TPC-H query Q11.](image)

**6 EXPERIMENTAL EVALUATION**

**6.1 Hybrid Memory Simulator**

We use the state-of-the-art GEM5 [4] cycle accurate simulator for the validation of data placement. Table 2 summarizes the memory system properties. We configure GEM5 with two memory controllers, one interfacing with 256 MB DRAM and one interfacing with 8 GB RRAM. The timing parameters for DRAM and RRAM are derived from the Micron DDR3 datasheet [35] and prior literature [13, 28, 43]. Specifically, we derived the timing parameters for RRAM following Lee et al [28]. $t_{RCD}$ is the number of clock cycles required between active Row Access Strobe (RAS) and Column Access Strobe (CAS) and represents read delay. For RRAM, we configured this parameter as $2 \times$ DRAM. $t_{RP}$ is the number of cycles needed to terminate access to an open row and open access to the next row. $t_{RP}$ represents write delay. For RRAM, we configured $t_{RP}$ as $2.2 \times$ DRAM cycles. Moreover, we modified the GEM5 memory controller to avoid write-back of clean cache blocks to memory. This not only improves the endurance of RRAM, but also saves write energy. $t_{RRDPrel}$ represents the delay between writes and $t_{RRDAct}$ represents delay between reads due to power budget constraints. Following the model in [28] we derived $t_{RRDPrel}$ 6 ns and $t_{RRDAct}$ 6.5 ns for RRAM. Both DRAM and RRAM memory controllers use open page management policy with maximum 4 accesses per row before closing.

We extend GEM5 to expose the DRAM and RRAM virtual address ranges to the key memory allocation routine $mmap$ (our malloc library builds on $mmap$) and to enforce the placement of global variables. We implement a new instruction in X86 instruction set which makes a special system call from an application to GEM5 in order to register starting address and size of each memory allocation. The memory controllers use this information to decide whether to send read or write operations to these ranges to DRAM or RRAM. During the simulation phase, we fast forward the first 1 G instructions in atomic mode and simulate next 6 G instructions in detailed mode with 30% sampling rate. Performance is reported on the basis of the cycles-per-instruction metric.

In order to save simulation time and by-pass the operating system, we perform all the validation in GEM5 System Call Emulation (SE) mode. In order to run MonetDB on top of GEM5 SE mode, we extend GEM5 with missing Linux system calls such as $mmap$, $munmap$, readdir and getdents.

**6.2 Workload**

We analyze two standardized benchmarks for Decision Support Systems and Big Data [56]. TPC-H [56] models a complex decision support system. It consists of 22 analytical queries. TPC-DS [39], [57] is a more modern analytical benchmark. It contains a large number of tables and

### Table 2: Simulation parameters

<table>
<thead>
<tr>
<th>Component</th>
<th>configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC cache</td>
<td>20 MB, 4-way set-associate, 64 byte blocks, LRU</td>
</tr>
<tr>
<td>DRAM</td>
<td>256 MB, $t_{RCD} = 5$, $t_{RP} = 5$, $t_{RRDPrel} = 3$, $t_{RRDAct} = 3$</td>
</tr>
<tr>
<td>NVM</td>
<td>8 GB RRAM, $t_{RCD} = 10$, $t_{RP} = 11$, $t_{RRDPrel} = 6, ns$, $t_{RRDAct} = 6.5, ns$</td>
</tr>
</tbody>
</table>

We analyze two standardized benchmarks for Decision Support Systems and Big Data [56]. TPC-H [56] models a complex decision support system. It consists of 22 analytical queries. TPC-DS [39], [57] is a more modern analytical benchmark. It contains a large number of tables and
columns that fully test the indexing and query optimization techniques of databases for large data sets. It consists of 99 analytical queries. We use these benchmarks at scale factor 5.0. We demonstrate our memory management techniques in the MonetDB database [7]. We compile MonetDB from source with the CLANG compiler and ‘-O2’ optimization.

6.3 Application Data Placement

We implement a custom memory allocator for DRAM and NVM by extending the dlmalloc memory allocator1, version 2.8.6. We build two custom memory allocators, one each for DRAM and NVM. Both allocators use LINUX memory management2 to allocate large distinct virtual memory address (VMA) regions. In a 64 bit OS, 48 bits are used for virtual address. The number of available virtual addresses in 64 bit OS are $2^{48}$ which is more than what is needed in the existing systems. Large part of that address space remain un-used. We use this un-used virtual address space to allocate two distinct regions for DRAM and NVM and let the custom memory allocators manage those explicitly to serve memory allocation requests from the application. The custom allocators claim distinct virtual memory address regions for each memory type using the MAP Fixed feature of the Linux mmap system call which allows us to control the starting address of each region.

6.4 Impact on Performance

For TPC-H, Figure 7 shows the performance of our heuristic placement policy called FEED. It compares this against 3 other configurations: an NVM-only system, a hybrid system with oracle based quantitative placement methodology and a hybrid system with the RaPP hardware scheme. Performance shown is performance degradation over a DRAM-only system. Lower bars are better. Figures 7 and 8 correspond to the 22 TPC-H queries. Lower bars are better. Numbered cases correspond to TPC-H query numbers.

Fig. 7: TPC-H Performance degradation of an NVM-only system (NVM), oracle placement, the FEED heuristic and RaPP compared to a DRAM-only system. Lower bars are better. Cases 1-22 correspond to the 22 TPC-H queries.

Fig. 8: TPC-DS Performance degradation of an NVM-only system (NVM), FEED and RaPP compared to a DRAM-only system. Lower bars are better. Numbered cases correspond to TPC-DS query numbers.

6.3 Application Data Placement

We implement a custom memory allocator for DRAM and NVM by extending the dlmalloc memory allocator1, version 2.8.6. We build two custom memory allocators, one each for DRAM and NVM. Both allocators use LINUX memory management2 to allocate large distinct virtual memory address (VMA) regions. In a 64 bit OS, 48 bits are used for virtual address. The number of available virtual addresses in 64 bit OS are $2^{48}$ which is more than what is needed in the existing systems. Large part of that address space remain un-used. We use this un-used virtual address space to allocate two distinct regions for DRAM and NVM and let the custom memory allocators manage those explicitly to serve memory allocation requests from the application. The custom allocators claim distinct virtual memory address regions for each memory type using the MAP Fixed feature of the Linux mmap system call which allows us to control the starting address of each region.

1. ftp://gee.cs.oswego.edu/pub/misc/malloc.c

1. ftp://gee.cs.oswego.edu/pub/misc/malloc.c
For TPC-DS, Figure 8 summarizes the performance of NVM-only system, hybrid memory with our ‘FEED’ approach and hybrid system with RaPP in comparison to a DRAM-only system. The performance decrease is relative to the number of off-chip accesses made by each individual query. Similar to TPC-H, NVM-only system has the worst performance for all the TPC-DS queries in comparison to DRAM-only system. The maximum performance decrease is 62% for TPC-H Q42. RaPP does not perform well either for any TPC-DS query. Q48 shows the maximum performance degradation up to 34%. FEED performs the best of all with the performance degradation less than 3.3% for all the TPC-DS queries (see Figure 8).

6.4.1 Analysis of Energy Savings

Our analysis shows that, for a column-oriented database, static energy dominates dynamic energy. More than 95% of the total energy is the static energy for TPC-H workload for scale factor 5 running on MonetDB with underlying memory system of size 8 GB. Consequently, the different in static energy dictates the energy savings in hybrid memory system regardless of the dynamic energy. The increase in dynamic energy due to NVM does not change the overall energy savings by much as shown in our results. Figure 9 shows the energy savings of NVM-only system, oracle placement, FEED and RaPP in comparison to DRAM-only system. NVM-only system has the highest energy savings because static energy of NVM is significantly lower than DRAM. However, NVM-only system performs worse in terms of performance. So the NVM-only system cannot be practical. On the other hand, oracle placement and heuristics have slightly less energy savings than the NVM-only system but these solutions ensure that the performance degradation is less than 5% as shown in figure 7. RaPP performs worse in energy savings because more pages are placed on NVM by the ranking algorithm and energy consumed during migration of pages between memories.

Similarly for TPC-DS workload, Figure 10 shows the energy savings of NVM-only system, FEED and RaPP in comparison to DRAM-only system. NVM-only system gives the highest energy savings but at the expense of increase CPI (see Figure 8). However, FEED gives energy savings up to 81.5% with maximum performance degradation up to 3.3%. RaPP does not perform well for energy savings in contrast to FEED approach (Figure 10).

6.4.2 DRAM vs NVM Reads and Writes

Figure 11 demonstrates the effectiveness of FEED to direct memory accesses to DRAM. It shows what percentage of the read, respectively write, accesses to main memory are serviced from DRAM. The remainder are serviced from NVM. At least 83% and up to 98% of the main memory accesses are mapped to DRAM across all queries. This high accuracy explains why our FEED approach achieve performance and energy close to the oracle.

Figure 11 moreover illustrates how the varying working set sizes for the queries affects the fraction of memory accesses that are served from DRAM. When the working
set size is larger, a lesser fraction of DRAM objects fits in DRAM and more memory accesses are served from NVM.

It is worth pointing out that servicing all memory accesses from DRAM may not be an optimal configuration, as the bandwidth of NVM would be unutilized. This problem was analyzed for a tiered memory system with near (3D die-stacked) and far DRAM memory [10]. A system is proposed whereby part of the near memory is disabled in order to force more accesses to the far memory. Bandwidth concerns were not included in the design of FEED, but could prove an interesting topic for future research.

6.5 Ease of Implementation

There are around 3250 dynamic allocation call sites in MonetDB that use malloc or mmap. Our default policy is to use NVM. There are only 7 call sites where a decision must be made to allocate on DRAM or NVM for OLAP workloads. As such, we were able to adapt a state-of-the-art database to hybrid memory with minimal code changes.

7 Conclusion

Non-volatile memory is crucial in order to sustain growing data sets for in-memory data intensive computing. As NVM has lower performance and higher energy consumption on access compared to DRAM, it is necessary to build a hybrid main memory system. This paper explores how databases can control the placement of data in either DRAM or NVM in order to optimize performance and minimize energy consumption for analytic workloads.

We have performed an in-depth analysis of a state-of-the-art column-oriented database and found that materialized intermediate data sets are the major contributor of total off-chip accesses. We have demonstrated that joins in general and hash-join in particular are the most critical database operators that cause most of the accesses to main memory for standardized OLAP benchmarks.

We propose online heuristics for dynamic allocation of the inputs and the output of critical join operators. Using cycle accurate simulation, we demonstrate that heuristic placement is as good as static oracle placement and much better than state-of-the-art application-agnostic data migration techniques. Only a handful of memory allocation sites in the DBMS need to be modified to allocate data sets on DRAM. The remainder of the allocations use the default allocator, mapping data onto NVM.

This work is a first step in exploring application-specific data placement in hybrid memories. We have made minimal adjustments to the MonetDB database to demonstrate the feasibility of this approach, however, further optimizations will enhance the effectiveness of the proposed techniques, e.g., query plan optimization may reduce the lifetime of fragments, fragmentation should control the size of the intermediate and depend on the memory type holding the intermediate. Moreover, it is an open question how much DRAM cache storage is required for effective operation. This will likely vary with the total database size and with the amount of NVM.

The techniques developed in this work apply to many types of hybrid memory. DRAM+NVM and die-stacked memory [5] are just two design points that will be realized in the near future. Future work is to apply our approach to other workloads where other operations, such as sorting and aggregation, may become more important bottlenecks.

Acknowledgments

This work is supported by the European Community’s Seventh Framework Programme (FP7/2007-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744) and the NanoStreams project (grant agreement 610509).

References

A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos. Software-
managed energy-efficient hybrid DRAM/NVM main memory. In 
Proceedings of the 12th ACM International Symposium on 
Computer Architecture, Anchorage, AK, USA, April 2015. ACM.

A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos. Software-
managed energy-efficient hybrid DRAM/NVM main memory. In 
Proceedings of the 12th ACM International Symposium on 
Computer Architecture, Anchorage, AK, USA, April 2015. ACM.

Y. Ho, G. M. Huang, and P. Li. Nonvolatile memristor memory: 
Device characteristics and design implications. In Proceedings of 
The 2015 International Conference on Computer-Aided Design, 


S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Ker-
sten, et al. Monetdb: Two decades of research in column-oriented 
database architectures. Bulletin of the IEEE Computer Society 

ITRS. International Technology Roadmap for Semiconductors, 2011 

M. Ivanova, M. Kersten, and F. Groffen. Just-in-time data distribution 
for analytical query processing. In Proceedings of the 16th East 
European Conference on Advances in Databases and Information 

A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High per-
formance cache replacement using reference interval prediction (rip). In 
Proceedings of the 37th Annual International Symposium on 


M. Kitsuregawa, H. Tanaka, and T. Moto-Okazawa. Application of 
hash to data base machine and its architecture. New Generation 

G. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 
Evaluating STT-RAM as an energy-efficient main memory alterna-
tive. In Performance Analysis of Systems and Software (ISSPASS), 2013 

change memory as a scalable dram alternative. In Proceedings of 
The 36th annual international symposium on Computer architecture, 
ISCA ’09, pages 2–13, New York, NY, USA, 2009. ACM.

H.-H. S. Lee and G. S. Tyson. Region-based caching: An energy-
delay efficient memory architecture for embedded processors. In 
Proceedings of the 2000 International Conference on Compilers, 
Architecture, and Synthesis for Embedded Systems, CASES ’00, pages 
120–127, New York, NY, USA, 2000. ACM.

C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and 
T. Keller. Energy management for commercial servers. Computer, 

Y. Li, Y. Chen, and A. K. Jones. A software approach for combating 
ACM/IEEE International Symposium on Low Power Electronics and 


S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join 
on modern hardware. IEEE Transactions on Knowledge and Data 

server idle power. In Proceedings of the 14th international conference 
on Architectural support for programming languages and operating 
systems, ASPLOS XIV, pages 192–202, New York, NY, USA, 2009. ACM.

Last consulted: Nov 2014.

A. Mirhoseini, M. Potkonjak, and F. Koushanfar. Coding-based 
energy minimization for phase change memory. In Proceedings of 
The 49th Annual Design Automation Conference, DAC ’12, pages 68– 
76, New York, NY, USA, 2012. ACM.

J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating sys-
tem support for nvm+dram hybrid main memory. In Proceedings of 
The 12th Conference on Hot Topics in Operating Systems, HotOS ’09, 


R. O. Nambiar and M. Poess. The making of TCP-DS. 
In Proceedings of the 32Nd International Conference on Very Large Data 

Sofort: A hybrid scm-dram storage engine for fast data recovery. In 
Proceedings of the Tenth International Workshop on Data Management 
on New Hardware, DaMoN ’14, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

Instant recovery for main memory databases. In CIDER 2015, 
Seventh Biennial Conference on Innovative Data Systems Research, 

S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage 
mangement in the nvram era. Proc. VLDB Endow., 7(2):121–132, 

T. Perez and C. De Rosa. Non-volatile memory: Emerging tech-
nologies and their impacts on memory systems. Technical report, 

2015.

Adaptive insertion policies for high performance caching. In 
Proceedings of the 34th Annual International Symposium on Computer 

Ahmad Hassan

Ahmad Hassan is a senior software Architect in SAP UK Limited. He is currently working on SAP Commerce Cloud with main emphasis on commercialisation of Context Driven Services to SAP customers. SAP Commerce Cloud provides personalisation of commerce experience by capturing and analysing every moment of user behaviour. His work is to ensure that the product is designed as Cloud Microservices based system running over Kubernetes and scale well with growing number of customers sending millions of merchandising events per day on SAP Commerce Cloud. He holds a PhD in Computer Science and has more than 10 years of experience in working for prestigious organisations like CERN, HP, Google and SAP.

Dimitrios S. Nikolopoulos

Dimitrios Nikolopoulos is Professor in the School of Electronics, Electrical Engineering and Computer Science and Director of the Institute of Electronics, Communications and Information Technology (ECIT) at Queen’s University Belfast. His research interests are in system software, architecture, and performance engineering of large-scale computing systems. He serves on the Strategic Advisory Board on eInfrastructure for the UK Engineering and Physical Sciences Research Council. He is a Fellow of the Royal Academy of Engineering and a Distinguished Member of the ACM and Senior Member of the IEEE. He completed his PhD in 2000 at the University of Patras.

Hans Vandierendonck

Hans Vandierendonck is Associate Professor (Senior Lecturer) in the School of Electronics, Electrical Engineering and Computer Science, and a Fellow of the Institute on Electronics, Communications and Information Technology at Queen’s University Belfast. His research aims to build efficient and scalable computing systems for data-intensive applications. He is a Senior Member of the IEEE and a Senior Member of the ACM. He holds a PhD degree from Ghent University.