Near-Data Filters: Taking Another Brick from the Memory Wall

Diego G. Tomé[*]
CWI, The Netherlands
diego.tome@cwi.nl

Tiago R. Kepe
UFPR and IFPR, Brazil
trkepe@inf.ufpr.br

Marco A. Z. Alves
UFPR, Brazil
mazalves@inf.ufpr.br

Eduardo C. de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

ABSTRACT

In this paper, we use the potential of the near-data parallel computing presented in the Hybrid Memory Cube (HMC) to process near-data query filters and mitigate the data movement through the memory hierarchy up to the x86 processor. In particular, we present a set of extensions to the HMC Instruction Set Architecture (ISA) to filter data in-memory. Our near-data filters support vector instructions and solve data and control dependencies internally in the memory: internal register bank and branch-less evaluation of data filters transform control-flow dependencies into data-flow dependencies (i.e., predicated execution). We implemented the near-data filters in the select scan operator and we discuss preliminary results for projection and join. Our experiments running the select scan achieve performance improvements of up to 5.64× with an average reduction of 80% in energy consumption when executing a micro-benchmark of the 1 GB TPC-H database.

1. INTRODUCTION

In the past decades, the disparity between processor performance and main memory latency has grown tightly, a well-known problem called the "memory wall" [17]. The "memory wall" arises from technology limitations: performance improves up to 70% per year (30% recently) for processors, while only 10% per year for DRAM memories [7]. This increasing performance gap has a direct impact on large scale data processing, especially for in-memory databases. Although smart-SSD devices are being researched for database management systems (DBMS) [6], such approaches only benefit when the storage cannot fit in primary memory.

In-memory databases became popular over the years due to the dropping cost per bit of DRAM together with an important storage capacity growth from megabyte to terabyte of data. However, in-memory query processing suffers from the interconnection and cache latency required to move large amounts of data around the memory and cache hierarchy (i.e., hit the "memory wall"). The data movement across the memory and cache hierarchy accounts for $40 - 80\%$ of the execution time to validate query filters in resource stalls, memory stalls and branch mispredictions [1]. This problem becomes clear if we consider that most query filters are performed over large volumes of data (bigger than cache memories), creating a data streaming effect in the memory hierarchy, which does not benefit from the cache sub-system.

The emergence of smart memories, such as the Hybrid Memory Cube (HMC) [3], inverts the common data processing approach by moving computation to where data resides with many benefits, like providing faster response times and reducing energy consumption [12]. The HMC is a 3D die-stacked technology with stacks of DRAM logically split into 32 independent vaults interconnected using Through-Silicon Vias (TSVs) [4] to a logic layer at the end of the stack. The logic layer executes near-data instructions with high level of parallelism. Nowadays, commercial hardware is already shipped with the HMC, like, Intel Xeon Phi, Fujitsu SPARC64 XIfx and Xilinx Kintex FPGA, and the consortium to develop the technology also includes Samsung, Micron, NVidia, ARM, Open-Silicon and IBM.

In this paper, we spark the discussion of near-data query processing in HMC. Our goal is to mitigate the data movement of in-memory databases using the parallel computing potential of the HMC to process near-data query filters.

Overall, we provide the following main contributions: (1) We analyze the impact of the data filtering of the current database query processing (column/row-store) over the traditional x86 processor using the HMC as ordinary DRAM. (2) We extend the HMC ISA to reduce DRAM access and data movement when validating in-memory data. We also extend the HMC ISA to support data dependencies and branch-less decisions when evaluating query filters: merging multiple basic-blocks removing branches and annotating the instructions [5]. These extensions allow that control dependencies, such as the evaluation of filters, could be transformed into in-memory data dependencies, with less inter-

[*] This author contributed to this work while Master Student at UFPR.

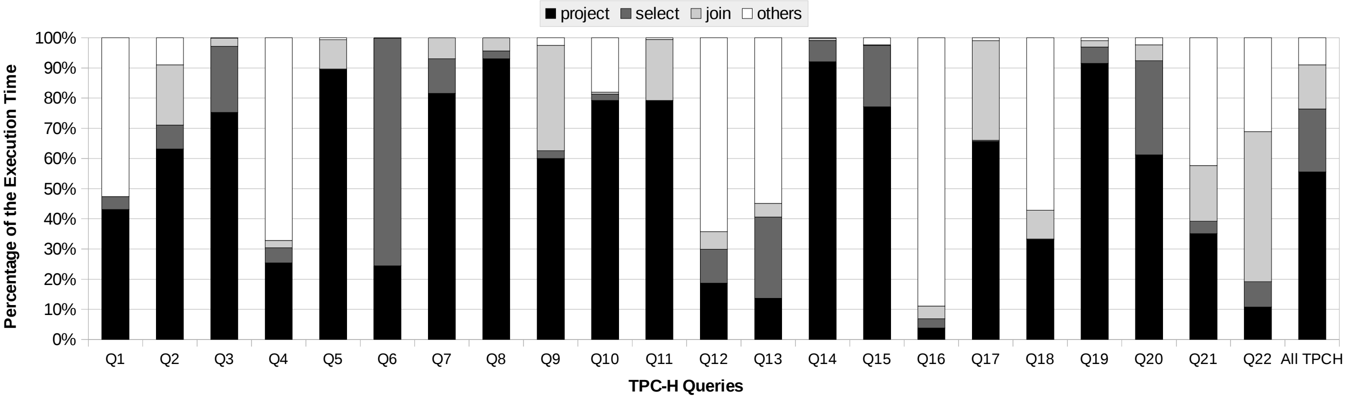


Figure 1: Top time consuming database operators in MonetDB [9] running the 100 GB TPC-H benchmark.

leaving between HMC and the processor. In [16], we validated the HMC Instruction Predication Extension (HIPE) from the hardware architecture perspective. In this paper, we describe the underpinnings to implement database operations over HIPE. (3) We perform a deep investigation of pros/cons of our near-data filter extensions now assuming column-store databases (the best case for OLAP) using the maximum degree of parallelism in the HMC. Cycle accurate simulations mimicking column-stores show promising performance improvements up to 5.64× with an average reduction of 80% in energy consumption when executing a micro-benchmark of the 1 GB TPC-H database.

Outline: Section 2 discusses where the execution time goes in different query operators and gives an overview of the HMC architecture, it also describes the predicate processing in row-stores and column-stores. Section 3 presents our architectural extension to perform near-data filters with the predicated execution, the full row-buffer and maximum parallelism available in the hardware. Section 5 discusses preliminary results of our next steps. Section 4 describes the experimental setup and evaluation results running a TPC-H micro-benchmark. Section 6 presents related work and Section 7 presents conclusions.

2. QUERY PROCESSING IN X86

In this section, we analyze the impact of each query operator in the execution time observing in particular the operators that require data-filters. Then, we overview the architecture of the HMC and discuss the execution of query filters over the current x86 architecture using the HMC as ordinary DRAM main memory.

2.1 Query Processing: Where Does Time Go?

In read-mostly database workloads, the large data movement required for filtering presents direct impact on performance [15]. Figure 1 shows the top time consuming database operators when we run a 100 GB TPC-H using the in-memory database MonetDB. We perform the experiments on a Intel quad-core i7-5500U@2.40GHz with RAM of 16 GB (DDR3L 1333/1600) and L3 cache size of 4MB running OpenSuse Leap 42.3 on Linux kernel 4.4.76-1-default. We obtained the performance trace of each TPC-H query using the TRACE statement modifier of MonetDB that includes the execution time of every database primitive. The last bar to the right in Figure 1 sums up the most time consuming operators of the TPC-H: projection, selection, join, and the remaining ones grouped into the category "others". The selection operator moves data around the memory hierarchy up to the processor to validate filter conditions on database columns representing around 20% of the execution time. The projection operator in turn represents around 56% of the execution time to materialize intermediate and final data of other operators, such as select and join. In practice, the projection operation filters data on intermediate results to project the columns in the query plan. Therefore, we consider projections as a data filtering operation.

Ailamaki et al. [1] show the amount of time taken by different operations in read-mostly databases. In particular, $40-80\%$ of the execution time to validate filters in the select scan is spent in three time consuming components: resource stalls with $20-35\%$ of the execution time (i.e., control-flow dependency and functional units), memory stalls with $15-30\%$ (i.e., data movement in the memory hierarchy) and branch mispredictions $5-15\%$. In our paper, we explore the potential of the HMC to minimize the effect imposed by these components in query filters.

2.2 The Hybrid Memory Cube Architecture

The HMC reduces the bandwidth gap between processors and memory by integrating memory and logic dies in a single 3D stack. Figure 2 illustrates the overall system architecture of the HMC. It integrates multiple DRAM bank layers and a single logic layer. Typical DDR-3 DRAM modules are organized in 8-KB rows, while the HMC DRAMs uses small rows of 256 B providing lower energy consumption and faster accesses to sparse addresses. Every set of 3D stacked banks forms a vertical group called vault interconnected by a Through-Silicon Vias (TSV) bus. In the current specification [10], the HMC provides 32 vaults that can independently access DRAM banks with a potential internal high bandwidth up to 320 GB/s.

The logic layer is placed on the base of the HMC and implements the vault controller, a mechanism that receives all the requests to the DRAM layers. Each vault also has their independent functional units enabling in-memory processing. The vault controller performs arithmetic and logical update atomic instructions with operands size of up to 16 bytes. They are variants of read-modify-write operations supported by some memory controllers [13]. This kind of operation normally reads data from any memory location, operates over data, then writes the result back to the same memory location. Once the operation finishes one of the

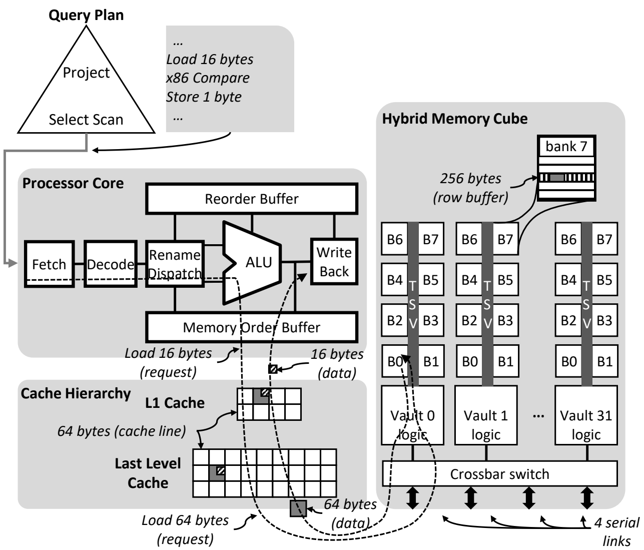


Figure 2: Traditional x86 processing to validate a filter condition with the HMC installed as conventional DRAM.

following is returned back to the processor: the old data or; the modified data or; the operation status.

In the current ISA, the HMC allows in-memory processing, but the processor still needs to trigger the instructions and wait for the results in order to send the resulting data to another HMC instruction (i.e., data-flow dependency) or to take decisions such as request other operations to be processed (i.e., control-flow dependency). The iteration between the HMC and the processor due to data-flow dependency increases data movement through the interconnection, while the control-flow dependency stalls the pipeline. Both situations increases the execution time and energy spent during computations.

2.3 Query Processing in HMC

Now we draw attention to the processing of queries in the HMC starting with the following question: "What happens when database systems run the near-data filtering over the current x86 architecture using the HMC as traditional DRAM?" Let us consider the "column-at-a-time" execution in the Decomposition Storage Model (DSM or column-store) and assume the data filtering executed by the select scan operator in the query plan. Figure 3 exemplifies the current approach of data filtering in the select scan operator over three columns of a table. In "Column 0", a full scan is required when evaluating the first filter in the query plan. The output of the scan is a bitmap with 1's for matched entries and 0's for not matched entries. The x86 processor loads only the matching entries to perform the second scan in "Column 1". This processing repeats for "Column 2" as we move on in the query plan.

We refer again to Figure 2 to illustrate the data movement scenario to process the validation of filters with the current x86 architecture instructions and the HMC placed as main-memory. In this scenario, the assembly instructions to process the filter conditions stay unmodified.

Initially, the x86 instructions are allocated in the processor pipeline. In current x86 architecture supporting AVX-128 extensions, an instruction may request up to 16 bytes of data to the cache memory. In general, this is only sufficient for a chunk of the database column and multiple requests are required to scan the entire column. In the first access, a cache miss in L1 and LLC requires a memory access. The processor then requests 64 bytes to main memory due to the size of a cache line, but the HMC placed as main-memory provides 256 bytes per access in the buffer: this means that $\frac{3}{4}$ of the time and energy to access the memory buffer may be wasted. Afterwards, only 64 bytes are returned back to cache and, when data is positioned in cache, the processor operates only over 16 bytes per instruction to finally evaluate query filters for the column chunk. Even considering the full cache line for query filtering, we waste energy and time installing elements inside the line for a single access only.

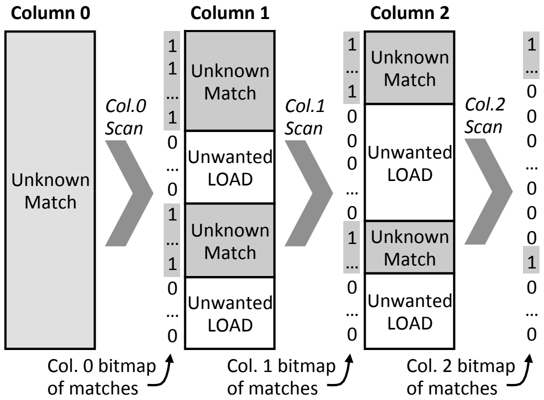


Figure 3: Select scans validating matching bitmaps in the "column-at-a-time" strategy.

We run a micro-benchmark to understand the filtering of data in the architecture of Figure 2 in two ways: (1) the x86 processor validating the selection filters with the HMC as DRAM and (2) the HMC validating the selection filters replacing the x86 instructions for those supported by the current ISA. We discuss modifications in the assembly code of the query and their execution later on in this paper, but basically we swap x86 comparison instructions to HMC comparison instructions and the execution pipeline sends the instruction to execute in the HMC. In this motivation experiment, we execute the TPC-H Query 06 with the traditional strategies: "tuple-at-a-time" and "column-at-a-time".

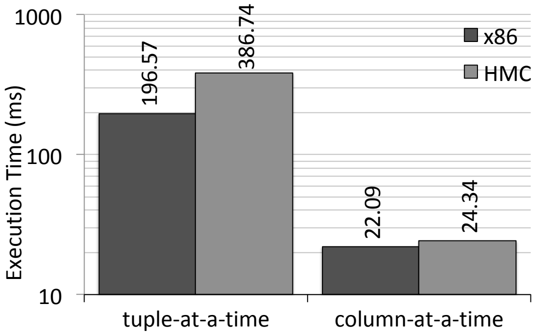
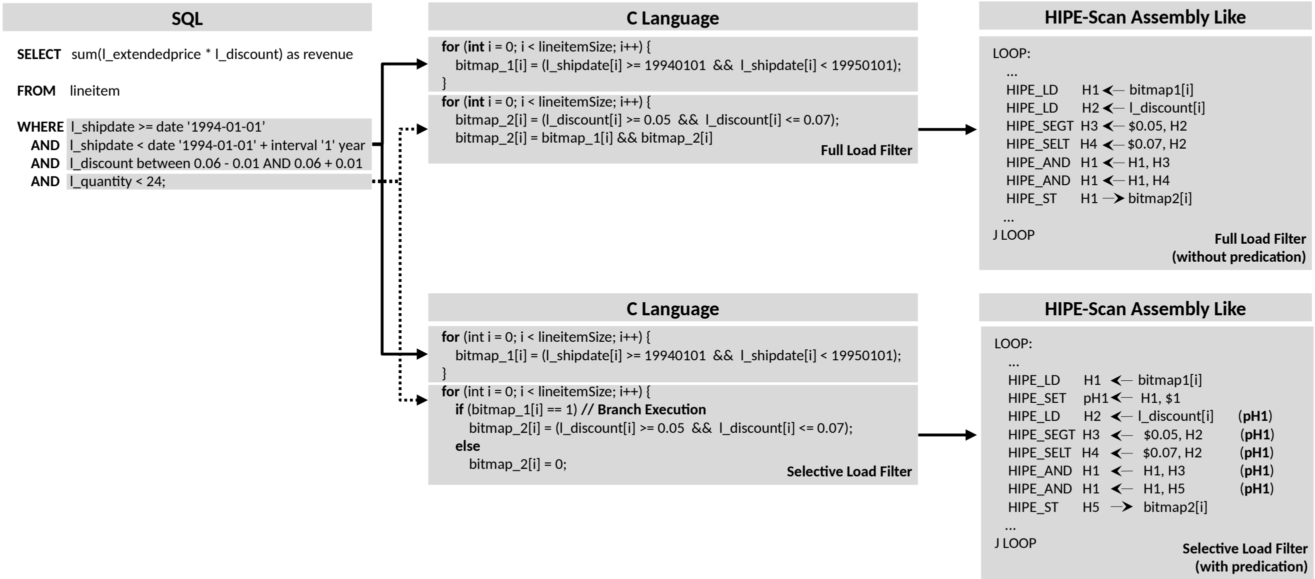


Figure 4: The execution time of the TPC-H Query 06 in the current x86 and HMC architectures with two storage organizations (tuple/column-store).

Figure 4 shows the response time of Query 06 in 1 GB database in the x86 and in the HMC architectures. We



SQL

```
SELECT   sum(l_extendedprice * l_discount) as revenue

FROM   lineitem

WHERE l_shipdate >= date '1994-01-01'
   AND  l_shipdate < date '1994-01-01' + interval '1' year
   AND  l_discount between 0.06 - 0.01 AND 0.06 + 0.01
   AND  l_quantity < 24;
```

C Language

```
for (int i = 0; i < lineitemSize; i++) {
   bitmap_1[i] = (l_shipdate[i] >= 19940101  &&  l_shipdate[i] < 19950101);
}
for (int i = 0; i < lineitemSize; i++) {
   bitmap_2[i] = (l_discount[i] >= 0.05  &&  l_discount[i] <= 0.07);
   bitmap_2[i] = bitmap_1[i] && bitmap_2[i]
```

Full Load Filter

HIPE-Scan Assembly Like

```
LOOP:
   ...
   HIPE_LD     H1 <— bitmap1[i]
   HIPE_LD     H2 <— l_discount[i]
   HIPE_SEGT  H3 <— $0.05, H2
   HIPE_SELT  H4 <— $0.07, H2
   HIPE_AND   H1 <— H1, H3
   HIPE_AND   H1 <— H1, H4
   HIPE_ST     H1 —> bitmap2[i]
  ...
J LOOP
```

Full Load Filter
(without predication)

C Language

```
for (int i = 0; i < lineitemSize; i++) {
   bitmap_1[i] = (l_shipdate[i] >= 19940101  &&  l_shipdate[i] < 19950101);
}
for (int i = 0; i < lineitemSize; i++) {
   if (bitmap_1[i] == 1) // Branch Execution
      bitmap_2[i] = (l_discount[i] >= 0.05  &&  l_discount[i] <= 0.07);
   else
      bitmap_2[i] = 0;
```

Selective Load Filter

HIPE-Scan Assembly Like

```
LOOP:
   ...
   HIPE_LD     H1   <—  bitmap1[i]
   HIPE_SET    pH1 <—  H1, $1
   HIPE_LD     H2   <—  l_discount[i]      (pH1)
   HIPE_SEGT  H3   <—   $0.05, H2         (pH1)
   HIPE_SELT  H4   <—   $0.07, H2         (pH1)
   HIPE_AND   H1   <—   H1, H3            (pH1)
   HIPE_AND   H1   <—   H1, H5            (pH1)
   HIPE_ST     H5   —>   bitmap2[i]
  ...
J LOOP
```

Selective Load Filter
(with predication)

Figure 5: The translation of the TPC-H Query 06 to C and simplified assembly version. Our two versions consider the HIPE-Filter instruction set performing *full load filter* or performing *selective load filter* using predicated instructions.

provide further details of the execution environment in Section 4. We observe that the x86 processor still presents the best response times to validate filter conditions compared to the HMC no matter the query engine. The problem when simply validating data filters in the HMC is related to the current HMC ISA. First, there is only single memory address operations (update instruction). Only operations between address and immediate are possible (e.g., *attribute* == *constant*). This narrows the implementation of different types of filters in databases, like, operations between two distinct addresses (e.g., *attribute* == *attribute*). Second, comparisons are only executed with *compare-and-swap* instructions. This instruction is costly to operate over data, since data is modified after every evaluation of a query filter. Third, the instructions operate over 16 bytes of data at a time wasting the potential of the DRAM row-buffer. Compared to the row-wise query engine, we observe that the column-wise suffers less impact with the small 16 bytes load request, because only the requested columns in the query statement move data around, while row-wise engines require moving tuples that are bigger than 16 bytes in OLAP databases. Fourth, the processor only triggers HMC instructions enough to use few parallelism between the vaults (i.e., only a small portion of parallelism is explored).

3. NEAR-DATA FILTERING

Data filtering is broadly used in many applications to validate data (e.g., analytics, business transactions, scientific simulations), however, we focus on filtering relational data once it covers most of those applications. In this section, we expose what happens to the data filtering when extrapolating some architectural limitations of the HMC. In previous work [15, 16], we presented hardware extensions that backed the contributions of this paper. In particular, we benefit from these extensions: (1) to use the maximum loop unroll depth of 32x and benefit from the HMC parallelism; and (2) to implement predication in the query execution pipeline changing processor oriented control-flow by near-data dataflow.

3.1 HMC Instruction Predication Extension (HIPE-Filter)

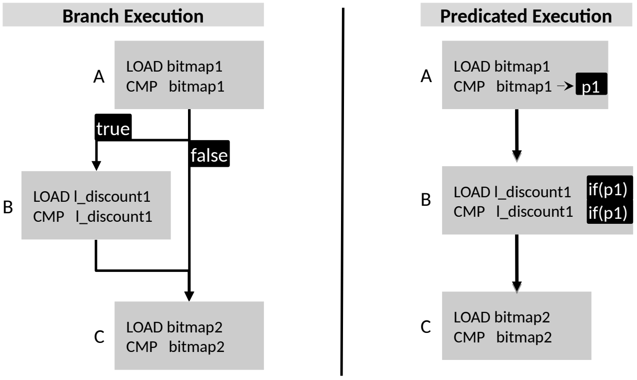


Figure 6: Branched vs. predicated execution of query filters.

Compared to [16] we change our focus from the hardware extensions and present in greater detail the predication in the near-data execution of query plans.

We refer to Figure 5 to exemplify the interaction between the x86 processor and the HMC while running the "Full Load Filter" version of the TPC-H Query 06. The evaluation of filters occurs independently in each column and does not take into account intermediate matching entries between columns (i.e., unwanted load operations are irrelevant).

The "Selective Load Filter" on the other hand, evaluates the intermediate results between each column to decide whether the values in the next column will be evaluated. Let us consider the branched execution of a bitmap depicted by Figure 6. The bitmap resulted from the first predicate processing is used to evaluate the data in the second column or assigning 0 to the second bitmap. In control-flow decisions, the evaluation of conditions leads to the correct branch and the rest of the code only executes after the branch finishes its instructions. The problem with the branched code is twofold: (1) the number of CPU cycles required to determine the next memory address to fetch when traversing a branch (i.e., jump instruction), which wastes memory bandwidth; and (2) more branch mispredictions with direct impact on performance [5]. Previous work could only solve such problem inserting a full processor inside the memory with a huge physical area overhead.

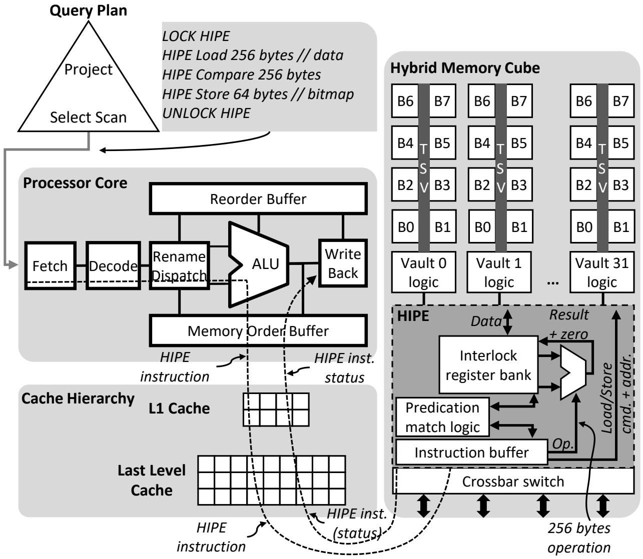


Figure 7: The architecture of the HIPE-Filter.

Instead, we benefit from the predicated execution to implement the near-data filtering (see Figure 6). Using predicated instructions we can merge multiple basic-blocks into a single super-block: many branches are removed and the instructions annotated [5]. Annotated instructions are only executed under a certain condition: in the case a predicate is false, the predicated instructions are squashed by the logic layer, i.e. these instructions are converted to *NOP* operations and simply discarded avoiding side effects.

3.2 Hardware extensions

Figure 7 depicts our extensions to the HMC architecture to allow the predicated execution. HIPE is formed by an instruction buffer to keep incoming instructions into the mechanism. A register bank, formed by 36 registers of 256 bytes each (total of 9 KB). The instructions are executed in-order, and each HIPE instruction belongs to one of the three classes: lock/unlock, load/store, ALU operation. The lock/unlock are used to gain access to the HIPE structure, avoiding conflicts to the register bank among multiple threads. The load/store instructions perform data transfers between the DRAM and the register bank inside the HIPE. The ALU operations perform computations inside the ALU using operands from the register bank. Before execution, the load/store and ALU instructions check the predicated register to proceed.

The register bank implements an interlock mechanism, which means that each register has a valid flag, in order to continue the execution during independent loads, only stopping the execution on real data dependencies. Each register stores not only the result value from the operations, but also the is-zero flag from each operation to be used during the predicated execution.

The predication match logic is responsible for checking if the predicate is true or false before the execution of an instruction. In case the predicate is true, the predicated instruction can be triggered normally. In case the predicate is false, the instruction is transformed into a *NOP*.

We are now ready to discuss the assembly codes in Figure 5 and replace control-flow to data-flow dependency in the HMC. In the assembly code of the "Full Load Filter" version, all the instructions execute sequentially, but more importantly comparisons require the decision from the x86 processor to move on in the execution. Now, the assembly code of the "Selective Load Filter" version presents the predicated execution, where no instructions annotated with $pH1$ are executed if the query filter evaluates false.

The following modifications are required to implement the HIPE-Filter: **(Database)** we require no changes in the source code of the database system to implement the HIPE-Filters, but it needs to be recompiled to use HIPE instructions. **(Processor)** the processor needs an extension to its ISA to provide the execution of HIPE instructions by the pipeline and the TLB, in a similarly required by HMC ISA. **(HMC)** we extended the current HMC ISA to include compare instructions, as it only supports compare-and-swap instruction (HMC_SWP) to evaluate values.

3.3 Understanding the Impact of Predication

In this section, we present simplified version of the reality using an execution flow diagram to explain the trade-off between the HIPE-Filters with and without predication. In this example, we consider 1 CPU cycle for every instruction inside HIPE and the load/store latency varies between 80 cycles with few operations running in parallel (low contention) and 100 cycles with multiple operations running in parallel (high contention). It is important to notice that the simulator [2] used in our experiments, considers different latency and more component details. Thus, simulation results differ from the estimations present in this section.

Figure 8 describes the execution of *Full Load Filter* on the *l_discount* column with the bitmap generated by the scan on the *l_shipdate* column. Here, we illustrate a select scan unrolled 4×. After the lock instruction, during each clock cycle, a 256 bytes-wide load is requested, loading the bitmap of the *l_shipdate* column scan. Considering the DRAM access latency with high contention, the validation of the filters of the *l_discount* column are performed after the cycle 102. From cycle 106, the results are compared with the bitmap of the *l_shipdate* column generating the next bitmap to be stored in memory. Therefore, the data-dependency between the first *LD* and the *CMP* instructions produces an stall of almost 100 cycles per column in each lock/unlock block. The scan over 4× 256 bytes takes 192 cycles due to the data-dependency between the first load with the first comparison and the store with the unlock instruction.

Figure 9 illustrates the execution of the predicated *Selective Load Filter* for high selectivity. Here, we present the worst-case scenario: an increase of 36% in the execution time compared to the *Full Load Filter*. We observe an extra DRAM read latency included in the critical path, because the *Selective Load Filter* first loads and compares the bitmap from the *l_shipdate* column before issuing the load for the *l_discount* column.

Also in Figure 9, we can infer the best-case scenario, which usually happens for very low selectivity queries. Whenever

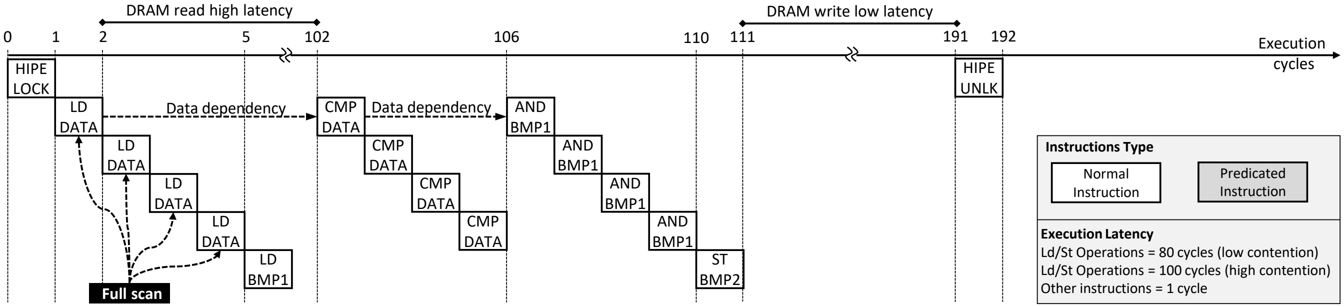


Figure 8: HIPE-Scan executing the *Full Load Filter* (no predication, with loop unrolling).

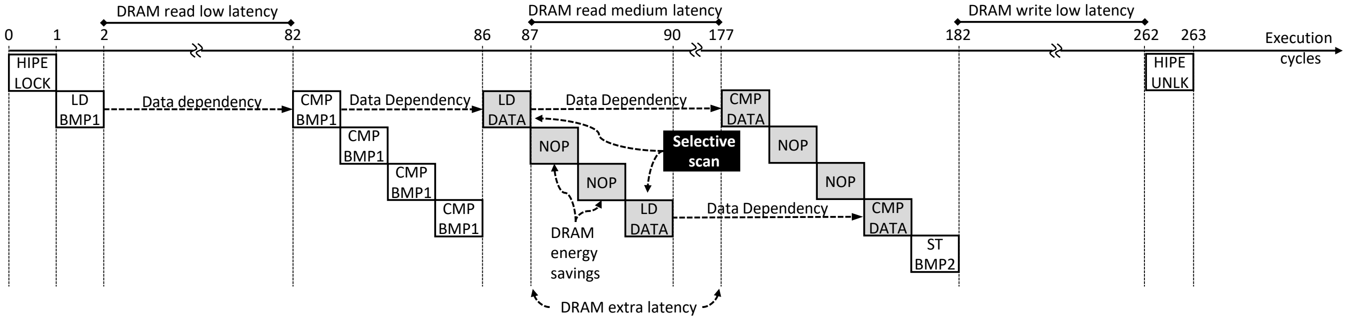


Figure 9: HIPE-Scan executing the *Selective Load Filter* (with predication and loop unrolling) worst/average case.

the previous bitmap presents no match on the unrolled loop, no load (indicated as selective scan) is issued, saving 87 cycles (between cycles 90 and 177 in this example). This scenario turns out to be a common case due to the low number of matches in the *l_shipdate* column, saving in our example 9% of the execution time compared to the *Full Load Filter*.

In summary, the *Full Load Filter* increased the DRAM contention and also wasted energy loading non-required data in the low selectivity scenario. On the other hand, the *Full Load Filter* performed better due to the lack of extra data-dependency for the high selective scenario.

4. EXPERIMENTAL EVALUATION

In this section, we provide a thorough investigation of pros/cons of our near-data filters compared to our initial work on HMC [15, 16]. This section presents the simulation environment, the experimental methodology and the results with the implementation of the near-data filtering.

4.1 Methodology and Setup

We used the SiNUCA cycle-accurate in-house simulator [2] to evaluate our proposal. SiNUCA was validated against real machines and allows modelling our custom architectural modifications inside the HMC to understand the system behavior when executing the near-data filters, considering an aggressive out-of-order processor, advanced multi-banked and non-blocking caches together with the HMC. Table 1 shows the major system parameters used in our study.

The baseline architecture is inspired by the Intel Sandy-Bridge processor micro-architecture referred to as x86. The Sandy-Bridge was modeled with the AVX-512 instruction set capabilities, and in all cases, the main memory used was the HMC version 2.1 [8]. For this baseline (x86), all the instructions are executed in the x86 processor.

In our experiments, we evaluate our near-data filters in a 1 GB TPC-H database running boolean expressions to filter data. We implemented our filters in the *select scan* due to the amount of data movement to validate filter conditions, as depicted by Figure 1. We implemented two versions of the *select scan* query operator: the HIPE-Scan implemented without predication (i.e., the x86 continues to trigger the instructions to the pipeline) and the HIPE-Scan implemented with predication. In particular, we run a micro-benchmark with the TPC-H Query 06, because the largest data movement in this query focus on the push-down of the most selective predicates. More details are present in the next section.

Table 1: Simulation parameters for evaluated systems.

OoO Execution Cores	16 cores @ 2.0 GHz, 32 nm; 6-wide issue; 16 B fetch; Buffers: 18-entry fetch, 28-entry decode; 168-entry ROB; MOB entries: 64-read, 36-write; 1-load, 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle); 1 branch per fetch; Branch predictor: Two-level GAs. 4,096 entry BTB;
L1 Data + Inst. Cache	32 KB, 8-way, 2-cycle; Stride prefetch; 64 B line; MSHR size: 10-request, 10-write, 10-eviction; LRU policy;
L2 Cache	Private 256 KB, 8-way, 4-cycle; Stream prefetch; 64 B line; MSHR size: 20-request, 20-write, 10-eviction; LRU policy;
L3 Cache	Shared 40 MB (16-banks), 2.5 MB per bank; LRU policy; 16-way, 6-cycle; 64 B line; Bi-directional ring; Inclusive; MOESI protocol; MSHR size: 64-request, 64-write, 64-eviction;
HMC v2.1	32 vaults, 8 DRAM banks/vault; DRAM@166 MHz; 8 GB total size; 256 B Row buffer; Closed-page policy; 8 B burst width at 2:1 core-to-bus freq. ratio; 4-links@8 GHz; DRAM: CAS, RP, RCD, RAS, CWD cycles (9-9-9-24-7); Per vault func. units (logical bitwise & integer); Latency: 1 cpu-cycle; Operation size (bytes): 16, 32, 64, 128, 256 (up to 16-B originally);
HIPE Logic	Unified func. units (integer + floating-point) @1 GHz; Latency (cpu-cycles): 2-alu, 6-mul. and 40-div. int. units; Latency (cpu-cycles): 10-alu, 10-mul. and 40-div. fp. units; Op. sizes (bytes): 16, 32, 64, 128, 256; Register bank: 36x 256 B;

4.2 Implementation Details

We assume the column-store model in our experiments. The evaluation of filters in the column-store model is performed with a column-at-a-time bitmap along the predicates: matches store "1" and no matches store "0" (as presented by Figure 3).

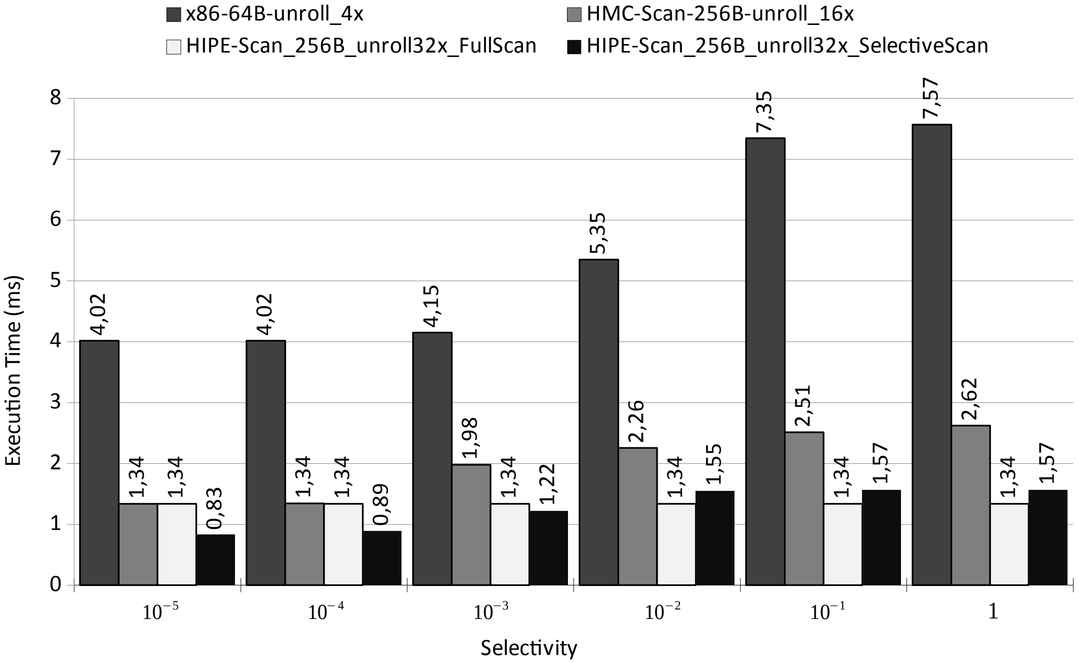


Figure 10: Evaluating execution time of TPC-H Q6 varying the selectivity factor in the different hardware architectures.

In our implementation, we assume the biggest operand size for each architecture: 64 Bytes for the x86 and 256 Bytes for the HMC. With big operand sizes, less instructions are necessary to operate over the dataset reducing the total amount of executed instructions. However, the processor still waits for the return status before triggering further instructions to the HMC. We also assume the biggest loop unroll depth for each architecture to take advantage of the parallelism: 8x for x86 and 32x for HMC.

4.3 Evaluation Results

4.3.1 Impact of Selectivity

In this section, we observe the impact of the filter selectivity in our near-data filters. Figure 10 describes the execution time varying the selectivity of the *l_shipdate* column in the TPC-H query 6. We did not vary the selectivity of the other two columns.

The selectivity greatly impacts the x86 architecture. The execution time increased almost 90% comparing the 10^{-3} and 1 selectivity factors. The execution time difference between x86 and HMC-Scan varies from 2.98× in the factor of 10^{-3} to up to 2.88× in the factor of 1.

For the HIPE-Scan performing the *Full Load Filter* algorithm, the selectivity factor has no impact in execution time as both wanted and unwanted data are requested and checked independent of the selectivity. When compared to the x86 architecture it shows an improvement of 3.00× for the 10^{-3} factor and 5.64× for the 1 factor.

The evaluation of the HIPE-Scan performing the *Selective Load Filter* algorithm achieves a better result when operates with selectivity smaller than 10^{-2}. HIPE-Scan using predication avoids many DRAM accesses improving by 4.84× for the 10^{-3} selectivity factor, and 4.82× for factor 1.

We notice a trade-off between the *Full Load Filter* and the *Selective Load Filter* considering different selectivity factors. Such trade-off shows us that the database scheduler needs adaptations to maximize the gains from the HIPE-Scan.

4.3.2 Energy Consumption

Figure 11 presents the energy consumption of the total DRAM accesses normalized by the x86 execution. The X-axis is divided into four architectures: “x86”, “HMC” with original functional units, “HIPE_Full” with vectorized instructions without predication, and “HIPE_Selective” with predication based on column selectivities. In each architecture the selectivity of the *l_shipdate* column varies among $0,001\%$ up to 100%. The energy estimations consider the DRAM values for HMC [10], focusing on the three main components, read, write and data transfers energy.

The *Selective Load Filter* is dramatically more efficient in energy consumption than the x86 (almost 3×). When compared to the HMC-Scan and the *Full Load Filter*, the *Selective Load Filter* is 1% and 4% more efficient respectively, because HIPE allows more instructions in the pipeline to evict stalls. When considering the extra energy used by the x86 to transmit data through the off-chip links, the *Selective Load Filter* saves 80% on average.

5. NEXT STEPS: PROJECTION AND JOIN

Now, we briefly discuss our next steps implementing our near-data filters as part of the projection and join operators. For the join, we implemented the Nested Loop Join (NLJ) due to its streaming behavior that benefits from the HMC parallelism. Other join algorithms (hash and sort-merge) generate random memory accesses that inhibit the potential of the HMC. For the projection, our implementation traverses the bitmap vector to filter the projection column: one load of 256-bytes of the input bitmap and, in case of matched entries, it executes up to 32 parallel loads of 256-bytes of the projection column and stores the values into the result vector. Those load and store instructions are on-chip

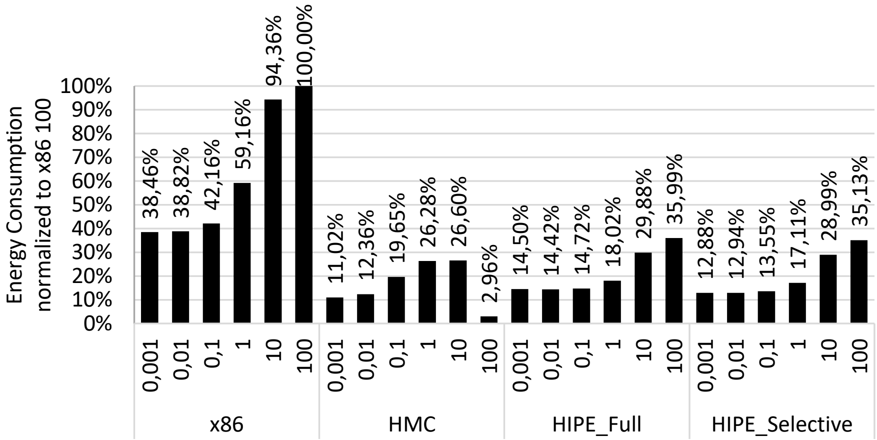


Figure 11: Energy consumption for select scan.

memory operation, i.e., the load instruction gets data from the DRAM dies to the HIPE registers within HMC and the store instruction does the inverse.

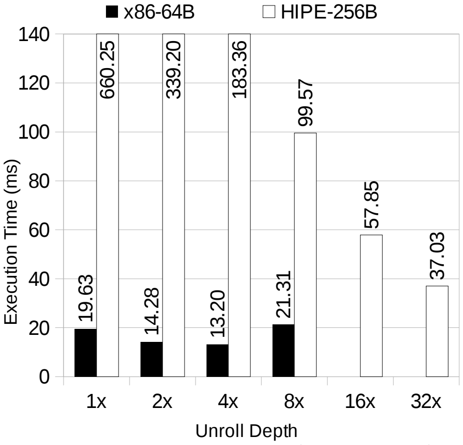


Figure 12: Execution time using the HIPE-Filters in the Nested-Loop join varying the loop unroll depth.

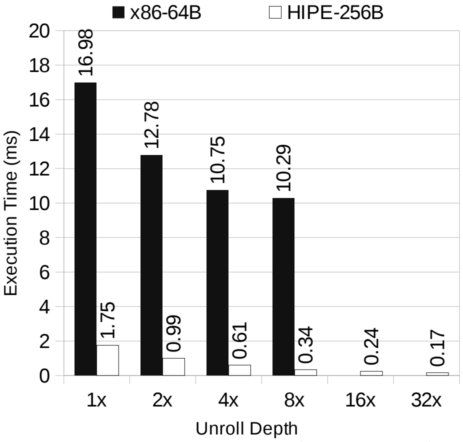


Figure 13: Execution time using the HIPE-Filters in the column projection varying the loop unroll depth.

Figure 12 and Figure 13 present preliminary results. The near-data join reaches poor performance against the x86. Although the NLJ streams the join columns, it enforces data reuse when repeatedly traversing the inner loop, as in this experiment, the inner table fits in cache. We plan new experiments to analyze the performance when the inner table does not fit in cache. The near-data projection improves the execution time by more than 1 order of magnitude compared to the x86 processor. The streaming behavior of projections causes low data reuse and less amount of off-chip data transfers during the copy of data (materialization). However, in this version of our near-data projection, we do not consider the impact of the Selective Store problem [14]: writing filtered values of non-continuous memory addresses to a new continuous memory address.

6. RELATED WORK

The impact of the memory-wall problem motivated several works over the past decades (caching, SSDs, NUMA), but we stick to near-data processing in DRAM due to space constraints.

The work developed in [18] presents the JAFAR, an external DRAM accelerator to push down select scan operations near-data in DDR-3. JAFAR processes a 64-bit word at a time by intercepting memory requests from the CPU in the DRAM I/O buffer. However, the data access must be coordinated to avoid collisions with CPU requests. Besides, JAFAR runs outside the processor and requires specific address translation to perform operations over the correct data inside the DRAM. In contrast, we take advantage of the logic layer of the HMC to execute the near-data filters without the necessity of coordination to access external hardware. This design choice maintains the common out-of-order execution and allows different ranges of word sizes up to 256 bytes to better use the HMC row-buffer.

The use of the HMC as main memory was evaluated by placing an accelerator inside the logic layer of the HMC to support join algorithms [11]. This work redesigns the hash and merge join algorithms to minimize the single word access (e.g., 16 bytes) and avoid the row buffer re-access. Unfortunately, it does not consider the necessary modifications in HMC to perform such operations for row-stores neither has evaluated the parallelism provided by the HMC.

[15] presents the usage of huge functional units to process large amounts of data with register banks inside the HMC. However, this design requires a fine control from the processor to choose the best operand size. Moreover, this design

is highly expensive to implement, because it requires lots of extra logic to provide serial access to HMC, extra interconnection and routing through the vaults, register banks and the extra control. To the best of our knowledge none of the past work analyze the current processing support of the HMC over read-mostly database workload, neither they analyze further modifications in the logic layer to fully execute database instructions. Our near-data filters extends our initial work [15, 16] on HMC exploring the logic layer to process data with native HMC operations.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present near-data query filters in the Hybrid Memory Cube (HMC). These near-data filters aim to reduce DRAM accesses achieving less data movement in the memory hierarchy (i.e., mitigating the "memory wall" problem), while providing high levels of parallelism.

To this end, we present extensions to the HMC Instruction Set Architecture (ISA). In particular, we extend the state-of-the-art HMC architecture [15] with predicated execution to transform control-flow into data-flow dependencies.

We evaluated our near-data filters against the main query execution engines in the x86 with AVX-512 instructions. We observed that choosing the correct filter algorithm based on the selectivity factor presents great impact on the performance, which may occur to other metrics inside the database. The execution of our near-data filters inside the HMC outperforms the column-at-a-time by 5.64× when compared to the baseline x86 execution using the HMC as conventional DRAM. When analyzing the energy consumption, we achieved average reductions of up to 80% due to reduced amount of off-chip data-transfers between the processor and memory.

Our results support that future in-memory database systems can benefit from near-data processing architectures, like the HMC. Our next steps include understanding our near-data filters in other database operations to design a HMC-aware database scheduler with query operations being dispatched to the most convenient hardware. Preliminary results with projections and joins encourage such design.

8. ACKNOWLEDGMENTS

This work was partially supported by the Serrapilheira Institute (grant number Serra-1709-16621).

9. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, and M. D. e. a. Hill. Dbmss on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[2] M. A. Z. Alves, C. Villavieja, M. Diener, and t al. Sinuca: A validated micro-architecture simulator. *HPCC*, pages 605–610, 2015.

[3] R. Balasubramonian, J. Chang, T. Manning, and et al. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro*, pages 36–42, 2014.

[4] E. Beyne, P. D. Moor, W. Ruythooren, and et al. Through-silicon via and die stacking technologies for microsystems-integration. *IEDM*, 2008.

[5] Y. Choi, A. D. Knies, L. Gerke, and T. Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *MICRO-34*, pages 182–191, 2001.

[6] J. Do, Y.-S. Kee, J. M. Patel, and et al. Query processing on smart ssds: Opportunities and challenges. In *SIGMOD*, pages 1221–1230.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[8] HMC-Consortium. Hmc specification 2.1, 2017.

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 2012.

[10] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *(VLSI)*, pages 87–88, 2012.

[11] N. S. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot. Sort vs. hash join revisited for near-memory execution. In *ASBD*, 2015.

[12] O. Mutlu. Memory scaling: A systems architecture perspective. In *Memory Workshop (IMW)*, pages 21–25, 2013.

[13] R. Nair, S. Antao, C. Bertolli, and al. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM JRD*, 2015.

[14] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[15] P. C. Santos, G. F. Oliveira, D. G. Tome, E. C. de Almeida, M. Zanata, and L. Carro. Operand size reconfiguration for big data processing in memory. In *DATE*, pages 710–715, 2017.

[16] D. G. Tome, P. C. Santos, L. Carro, E. C. de Almeida, and M. A. Z. Alves. HIPE: HMC instruction predication extension applied on database processing. In *DATE*, pages 261–264, 2018.

[17] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH*, 23(1):20–24, 1995.

[18] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. Beyond the wall: Near-data processing for databases. In *DAMON*, pages 2:1–2:10, 2015.