HIPE: HMC Instruction Predication Extension Applied on Database Processing

Diego G. Tomé[‡§], Paulo C. Santos[†], Luigi Carro[†], Eduardo C. Almeida[‡], Marco A. Z. Alves[‡]
[‡]Department of Informatics – Federal University of Paraná – Curitiba, Brazil
[†]Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[§]Centrum Wiskunde & Informatica (CWI) – Amsterdam, The Netherlands
Email:[‡]{dgtome, mazalves, eduardo}@inf.ufpr.br [†]{pcssjunior, carro}@inf.ufrgs.br

***Abstract*—The recent Hybrid Memory Cube (HMC) is a smart memory which includes functional units inside one logic layer of the 3D stacked memory design. In order to execute instructions inside the Hybrid Memory Cube (HMC), the processor needs to send instructions to be executed near data, keeping most of the pipeline complexity inside the processor. Thus, control-flow and data-flow dependencies are all managed inside the processor, in such way that only update instructions are supported by the HMC. In order to solve data-flow dependencies inside the memory, previous work proposed HMC Instruction Vector Extensions (HIVE), which embeds a high number of functional units with a interlock register bank. In this work we propose HMC Instruction Prediction Extensions (HIPE), that supports predicated execution inside the memory, in order to transform control-flow dependencies into data-flow dependencies. Our mechanism focus on removing the high latency iteration between the processor and the smart memory during the execution of branches that depends on data processed inside the memory. In this paper we evaluate a balanced design of HIVE comparing to x86 and HMC executions. After we show the HIPE mechanism results when executing a database workload, which is a strong candidate to use smart memories. We show interesting trade-offs of performance when comparing our mechanism to previous work.**

***Index Terms*—Processing-In-Memory; Predicated Instructions; Hybrid Memory Cube; Near-Data Database;**

I. Introduction

Over the past few years, the processing of read-mostly workloads with all data located in-memory became popular, such as database (DB) systems. However, these workloads hit the "memory wall" when moving large amounts of data around the memory hierarchy, suffering from the interconnection and cache latency. This data movement also increases the cache pollution, once the new line that will never be used needs to be installed inside the cache by removing potentially useful data. To tackle the problems of data movement, the processing-in-memory (PIM) approach [1], [2], [3] inverts the data processing path by moving computation to where data resides. PIM presents many benefits such as reducing energy consumption and providing faster response times [4].

Recently, the release of the Hybrid Memory Cube (HMC) made PIM tangible for data intensive applications [5]. The HMC can be used as a simple main memory, providing on average 10× better performance and 70% lower energy consumption. In order to execute instructions inside the HMC, the processor needs to send it specific instructions, and wait for a status or answer to return. This design keeps most of the pipeline complexity (front-end and graduation) inside the processor. Thus, control flow dependencies and data flow dependencies are all managed inside the processor, in such way that only update instructions (read-operate or read-modify-write) are supported by the HMC [6]. Moreover, the current set of HMC instructions is not favorable to operate over read-intensive applications, such as databases [7], since the only comparison implemented on HMC is performed by compare-swap instructions which overwrites the original data in memory. Furthermore, the small HMC instruction size (16 byte wide) is also a limiting factor for database systems.

Although more instructions could be easily supported by HMC, it would still be limited to update instructions, as control-flow and data-flow dependencies are still solved inside the processor. In order to solve data-flow dependencies inside the memory, previous work proposed HIVE [8], which embeds big vector functional units together with a register bank.

In this work we propose HIPE in order to support predicated instructions inside the memory. The predicated instructions enable the compiler to transform control flow dependencies into data flow dependencies [9]. Data flow dependencies are convenient to perform PIM data streaming operations, which include a wide range of applications (e.g., databases, sensors, monitoring). As far as we know, we are the first to propose and evaluate a technique to solve control-flow dependencies on smart memories without the usage of a full processor inside the memory. This paper presents the following main contributions:

- We evaluate a balanced design of previous work to take full advantage of the DRAM architecture provided by the HMC when processing the select scan operations in DB systems.
- We performed loop unroll in order to study its impact on the better usage of the vault parallelism inside the HMC.
- We use the simple design of the predicated execution in order to support decisions inside the memory, changing processor oriented control-flow by in-memory data-flow.

Evaluations with HIPE show that executing data-intensive applications with control-flow converted to predicated execution, HIPE is 6.46× better than x86 and loses 15% of performance compared to HIVE. Nevertheless, our proposal enables 3% DRAM energy savings on average.

The authors gratefully acknowledge the support of CNPq and CAPES.

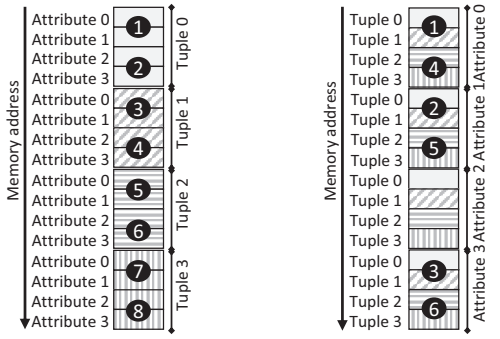
II. Background

A. The HIVE Architecture

In order to take advantage of HMC's intrinsic parallelism, HIVE architecture proposes to insert a instruction sequencer, a register bank and a set of vector functional units inside the HMC [8]. These functional units execute HIVE instructions, while the register bank enables the operation answer to be stored in an address different from the source data. Moreover, the interlock register bank allows computation to be overlapped with memory accesses.

B. Query Execution in Database Systems

Traditionally, the storage layout implemented by most of the database management systems (DBMS) is the N-ary Storage Model (NSM) or row-store. Figure 1a illustrates the storage layout implemented in row-stores and the processing flow. Considering a tuple (or row) composed of four attributes (or columns), for each step of the common tuple-at-a-time [10] processing, the select scan loads the entire tuple (one at a time), but only applies the operation in a small part of the tuple (i.e., only a few columns). In read-mostly databases, the tuple-at-a-time processing wastes memory bandwidth and causes huge occurrence of misses in cache, because the cache lines are filled with lots of irrelevant columns [11].



(a) Row-store model. (b) Column-store model.

Fig. 1: Database storage and selection scan execution order.

In order to avoid cache pollution for read-mostly DB is the Decomposition Storage Model (DSM) [12] or column-store. In this layout, a partition of the tuples and attributes are stored contiguously in memory. Thus, only the needed attributes traverse the memory hierarchy and the contiguous storage guarantees good cache locality. Figure 1b presents the data organization and processing for column-stores. Considering a database query evaluating Attributes 0, 1 and 3, the Attribute 2 is not load during the processing.

The traditional strategy to process query operations in columns-stores is the "column-at-a-time" [13]. The column-at-a-time is illustrated in Figure 1b. In this query processing strategy all the entries of Attribute 0 are evaluated generating an intermediate result to be used by Attribute 1 and then the same happens to evaluate Attribute 3. However, large volumes of intermediate results reside in memory for each attribute with increasing misses in cache and I/O overhead [14].

III. HIPE: HMC Instruction Predication Extension

When using HMC-based operations the processor is responsible for triggering only the correct execution path of the source code. This means that during control flow decisions, the processor can only trigger HMC/HIVE instructions after the branch is executed. Previous work could only solve such problem by inserting a full processor inside the memory, which have a huge area overhead. We propose to change control-flow by data-flow inside the memory by using predicated execution.

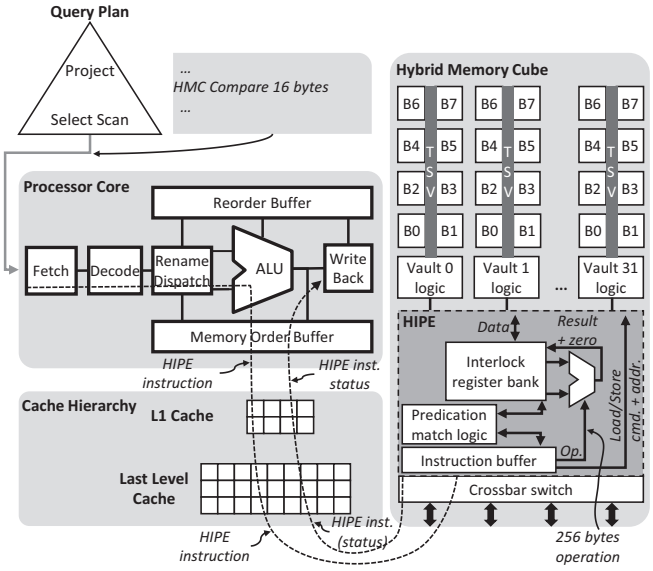


Fig. 2: HIPE architecture executing a database query plan.

Figure 2 presents the HIPE architecture executing a database query. HIPE is formed by a buffer to keep incoming instructions into the mechanism. A register bank, formed by 36 registers of 256 bytes each (total of 9 KB). An ALU based on neon-ARM functional units and the predication match logic.

The instructions are executed in-order, and each HIPE instruction belong to one of three classes: lock/unlock, load-/store, ALU operation. The lock/unlock are used to gain access to the HIPE, avoiding conflicts to the register bank. The load/store instructions perform data transfers between the DRAM and the register bank. The ALU operations will perform computations inside the HMC. The load/store and ALU instructions can have predicate, it means, that they will only be executed if some register matches the wanted value.

The register bank stores not only the result value, but also the zero flag from each operation. This flag is used during predicated execution. Moreover, it is implemented with an interlock mechanism, in order to continue the execution during loads, only stopping the execution on real data dependencies. The following modifications are required by HIPE:

Workload: No source code change is required, but it needs to be compiled to use HIPE instructions, similarly to AVX.

Processor: The processor needs an extension to its ISA to provide the execution of HIPE instructions (similarly to HMC support). The instructions pass the pipeline in the same way

TABLE I: 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);
HIVE 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 (up to 8192 B originally); Register bank: 36x 256 B; (16x 8192 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;

as a memory load operation. The requests of loads work with virtual addresses, although the addresses have to be translated by the Translation Look-aside Buffer (TLB) in order to respect a correct permission policy to access the given address range.

HMC: We based our implementation in the modifications proposed on HIVE [8]. Our goal is to take advantage of the execution of data-flow and control-flow dependencies inside the memory, reducing thus the interaction with the processor.

IV. Experimental Evaluation

Methodology and Setup We used the SiNUCA cycle-accurate in-house simulator [15] to evaluate our proposal. Table I shows the main parameters used in our study.

x86 baseline: This baseline is inspired by the Intel Sandy Bridge processor micro-architecture and referred to as x86. It was modeled with AVX-512 instruction set capabilities with all the instructions executed in the x86 processor. It uses the HMC version 2.1 as simple main memory.

HMC baseline: The second baseline uses the current set of operations support by HMC ISA extending it to different operator sizes from 16 bytes up to 256 bytes. In this work we extend the HMC update instructions to provide other instructions more convenient to execute our benchmark, for instance, the compare instruction is considered.

Benchmark: In our experiments, we use the TPC-H database with 1 GB running the Query 06. This query implements complex boolean expressions during the select scan operation. It also consists of conjunctions without join operations in the largest table called *lineitem*. We let join operations for future work as it requires understanding the impact of each one of the many different join algorithms on HMC.

Experiments Implementation: To evaluate the execution of the select scan with the tuple-at-a-time execution, the matched tuples are materialized as intermediate results. In the column-at-a-time execution, the predicate is performed for the first column, and it stores a bitmask with "1" for match and "0" for no match to be used ahead by the further predicates. We considered that each tuple in the table occupies 64-bytes, which is equal to the cache line size. The *store* instructions are executed with cache assistance in both x86 and HMC baselines. However, the *load-compare* instructions are processed inside the memory for HMC baseline. When using HIVE or HIPE both *load-compare* and bitmask *store* instructions are executed in the logic layer of HMC.

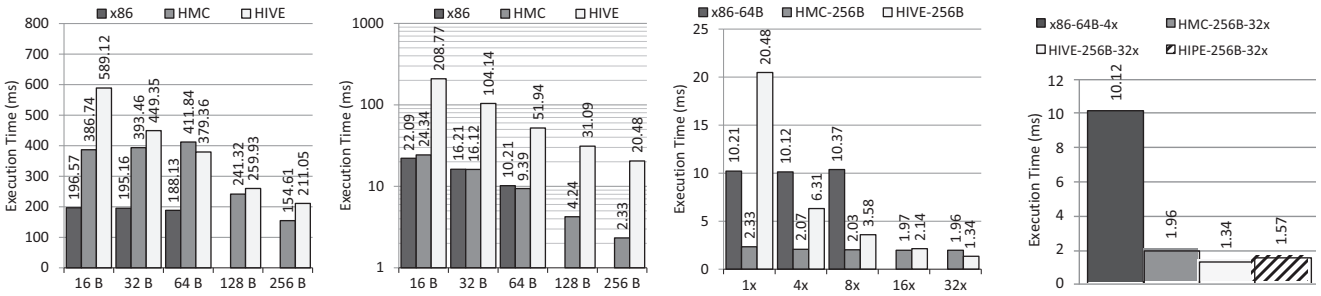
A. Experimental Results

1) Varying Operation Size: We evaluated the TPC-H Query 06 using HMC and HIVE using five different data operation sizes from 16 B to 256 B, while we set the x86 up to 64 B (i.e. the instruction size of AVX-512).

Figure 3a presents the results for the tuple-at-a-time execution in the NSM storage layout. When executing HMC-16 B the select scan execution time increased in 97% compared to x86. This behavior also occurred for 32 B and 64 B width operations, with increases of 1.02× and 1.19× respectively. This performance degradation was due to the small operation size, which requires multiple instructions to fully use the overall row-buffer size (i.e. 256 B). HMC-256 B achieves the best performance, with 18% gains compared to the best x86, such result happened because the select scan could process 4 contiguous tuples per operation without suffering from any extra cache latency. The execution with HIVE-16 B resulted in an increase of 3× in the execution time when compared to x86. For HIVE-256 B the execution time was still 11% bigger than x86. The increasing in execution time occurs due to the control-dependency of each isolated lock/unlock block when performing streaming operations with HIVE.

Figure 3b presents the results for the column-at-a-time execution in the DSM storage layout. When executing HMC-256B the execution time was reduced by 4.38× compared to x86. On the other hand, executing HIVE-256 B still takes 2× more when compared with the best case of x86 execution (AVX-512). Notice that after processing the first column, the processor needs to fetch the previous generated bitmask to decide the portions of the second column it needs to process. This generates data dependency and delays the execution of HIVE instructions as more DRAM accesses need to be performed, in contrast to cache access for x86 and HMC.

2) Different Unrolling Depths in Column-at-a-time : Figure 3c presents the results for the column-at-a-time in the DSM layout varying the loop unroll depths to increase parallelism. HMC and HIVE used five different unrolling depths: 1x to 32x, while we set the x86 up to 8x (i.e. the deepest unroll used by compilers due to the reduced number of general purpose registers). HMC-256 B with 32x loop unrolling could



(a) Tuple-at-a-time execution varying operation size.

(b) Column-at-a-time execution varying operation size.

(c) Column-at-a-time execution varying loop unrolling depth.

(d) Best cases of each architecture compared to HIPE.

Fig. 3: Execution of TPC-H Query 06 selection scan varying operation size and loop unrolling depth for x86, HMC and HIVE.

improve performance in 5.15× compared to x86. Meanwhile, HIVE-256 B with same unrolling caused an speedup of 7.57×. When the loop is unrolled, HIVE overlaps DRAM latency with parallel requests, thanks to the interlock register bank.

3) Predicated Execution: The main goal of the predicated execution is to reduce the amount of executed instructions by performing data and control flow inside the HMC itself. Figure 3d shows the speedup of the predicated execution of 5.15× in HMC, 7.55× in HIVE and 6.46× in HIPE compared to x86. While HIVE performs full scan in columns, HIPE only performs load and compare on required column regions. During the evaluation of the columns, HIPE guarantees that only useful data are loaded and compared. It means that, during the select scan, if the first attribute did not match the query condition the second attribute for that same tuple will not be loaded and compared. HIPE is 5% more efficient in energy consumption than x86 and compared with HMC and HIVE, it is 1% and 4% more efficient respectively.

V. Related Work

Previous work proposed an external DRAM accelerator called JAFAR [16] to execute near-data DB select scan operations 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. In contrast, we take advantage of the logic layer of the HMC to execute the select scan.

The work proposed in [17] places an accelerator inside the logic layer of the HMC, to support DB join operations. This work redesigns the hash and merge join algorithms in order to minimize row buffer re-access. However, this related work does not support such operations for row-stores.

The usage of huge vectorial functional units with register banks inside the HMC [8] called HIVE, was also proposed. However, such design does not solve control-flow dependencies inside HMC. Our approach has a more balanced design than HIVE, with only 256 byte operation size (96% smaller) and register bank with 32 registers of 256 bytes (94% smaller than the original proposal).

Moreover, the use of predicated execution was already investigated in several work [18], [19], but none of these implemented predication on smart memories.

VI. Conclusions

In this work, we presented the HMC Instruction Predication Extension (HIPE) to transform control-flow dependencies into data-flow dependencies. HIPE removes the high latency iteration between the processor and the HMC during the execution of branches that depends on data processed by memory. We showed tradeoffs comparing HIPE to previous work running database operations. HIPE performed 6.46× better than x86, but loses 15% of performance compared to HIVE due to additional data dependencies. However, preliminary results shows up to 5% DRAM energy savings using HIPE.

References

[1] W. H. Kautz, "Cellular logic-in-memory arrays," *IEEE Trans. Comput.*, 1969.

[2] D. Patterson, T. Anderson, N. Cardwell, and t al., "A case for intelligent ram," *IEEE Micro*, 1997.

[3] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational ram: Implementing processors in memory," *IEEE Des. Test*, 1999.

[4] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.

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

[6] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSI*, 2012.

[7] P. C. Santos, G. F. Oliveira, D. G. Tome, and et al., "Operand size reconfiguration for big data processing in memory," in *DATE*, 2017.

[8] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro, "Large vector extensions inside the hmc," in *DATE*, 2016.

[9] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *POPL*, 1983.

[10] G. Graefe, "Volcano an extensible and parallel query evaluation system," *TKDE*, 1994.

[11] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data page layouts for relational databases on deep memory hierarchies," *VLDB*, 2002.

[12] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," in *SIGMOD*, 1985.

[13] P. A. Boncz and M. L. Kersten, "Mil primitives for querying a fragmented world," *VLDB*, 1999.

[14] D. Abadi, P. Boncz, and S. Harizopoulos, *The Design and Implementation of Modern Column-Oriented Database Systems*, 2013.

[15] M. A. Z. Alves, C. Villavieja, M. Diener, and t al., "Sinuca: A validated micro-architecture simulator," *HPCC*, 2015.

[16] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *DAMON*, 2015.

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

[18] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," in *MICRO*, 2005.

[19] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *PACT*, 1998.