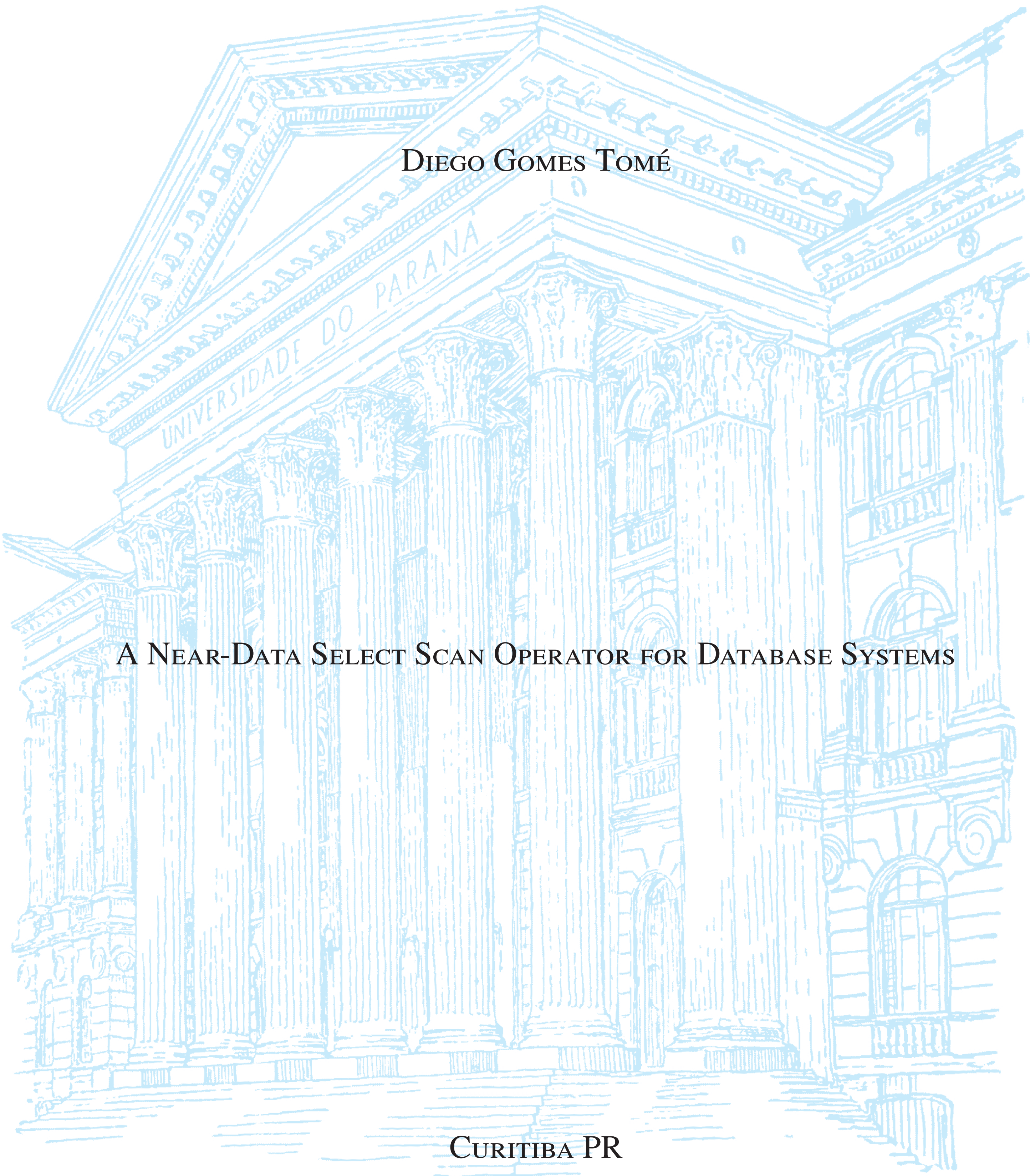
Universidade Federal do Paraná



Diego Gomes Tomé

A Near-Data Select Scan Operator for Database Systems

Curitiba PR

2017

Diego Gomes Tomé

A Near-Data Select Scan Operator for Database Systems

Dissertation presented as partial requirement to obtain the Master's Degree in Informatics in the Post-Graduate Program in Informatics, Exact Sciences Sector, Federal University of Paraná.

Concentration Field: *Ciência da Computação*.

Advisor: Eduardo Cunha de Almeida.

Co-advisor: Marco Antonio Zanata Alves.

Curitiba PR
2017

T656n Tomé, Diego Gomes
A Near-Data Select Scan Operator for Database Systems / Diego Gomes Tomé. – Curitiba, 2017.
64 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Eduardo Cunha de Almeida.

1. Ciência da computação. 2. SGBD em memória. 3. Cubo de memória híbrido. I. Universidade Federal do Paraná. II. Almeida, Eduardo Cunha de. III. Título.

CDD: 005.75

MINISTÉRIO DA EDUCAÇÃO
SETOR CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **DIEGO GOMES TOMÉ** intitulada: **A Near-Data Select Scan Operator for Database Systems**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.
A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 21 de Dezembro de 2017.

[illegible]
EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora (UFPR)

[illegible]
MARCO ANTONIO ZANATA ALVES
Co-orientador - Avaliador Interno (UFPR)

[illegible]
LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UFPR)

[illegible]
JOSÉ MARIA DA SILVA MONTEIRO FILHO
Avaliador Externo (UFC)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

If it was easy everyone was, if it was close everyone would go.

Agradecimentos

Firstly, I would like to thank my partner Jéssica de Aguiar and my family for their support and patience during the process of this dissertation. Every day that I had to exchange the time together by the laboratory to develop my research. Besides, the distance of my family that I've had to face, considering that I could see them just once a year.

I'm grateful to my advisor Prof. Dr. Eduardo Cunha de Almeida and my co-advisor Prof. Dr. Marco Antônio Zanata Alves, first for the opportunity to work with brilliant minds and for the continued support to achieve my Master's degree, second for his patience and motivation during the hard moments. Their guidance helped me in all the time that I have developed this dissertation and they made possible to achieve the results achieved with guidance during the research and the writing.

Besides my advisor and co-advisor, I would like to thank the rest of my dissertation committee Prof. Dr. José Maria da Silva and Prof. Dr. Luis C. de Bona for their insightful comments and encouragement, but also for the questions which bring encouragement to widen my research from various perspectives.

I would like also to thank my friends in the Department who have helped me understand the life in science and share the challenges during the academic ride, all the talks that we have had during the lunch and coffees were important to make the time in the Master's funnier and in the same time more valuable. Without they precious support it would not be possible to conduct this research.

Finally, I would like to thank all the people that in some way have contributed to help me in the development of this dissertation.

Resumo

Um dos grandes gargalos em sistemas de bancos de dados focados em leitura consiste em mover dados em torno da hierarquia de memória para serem processados na CPU. O movimento de dados é penalizado pela diferença de desempenho entre o processador e a memória, que é um problema bem conhecido chamado *memory wall*. O surgimento de memórias inteligentes, como o novo Hybrid Memory Cube (HMC), permitem mitigar o problema do *memory wall* executando instruções em chips de lógica integrados a uma pilha de DRAMs. Essas memórias possuem potencial para computação de operações de banco de dados direto em memória além do armazenamento de bancos de dados. O objetivo desta dissertação é justamente a execução do operador algébrico de seleção direto em memória para reduzir o movimento de dados através da memória e da hierarquia de cache. O foco na operação de seleção leva em conta o fato que a leitura de colunas a serem filtradas movem grandes quantidades de dados antes de outras operações como junções (ou seja, otimização *push-down*). Inicialmente, foi avaliada a execução da operação de seleção usando o HMC como uma DRAM comum. Posteriormente, são apresentadas extensões à arquitetura e ao conjunto de instruções do HMC, chamado HMC-Scan, para executar a operação de seleção próximo aos dados no chip lógico do HMC. Em particular, a extensão HMC-Scan tem o objetivo de resolver internamente as dependências de instruções. Contudo, nós observamos que o HMC-Scan requer muita interação entre a CPU e a memória para avaliar a execução de filtros de consultas. Portanto, numa segunda contribuição, apresentamos a extensão arquitetural HIPE-Scan para diminuir esta interação através da técnica de predicação. A predicação suporta a avaliação de predicados direto em memória sem necessidade de decisões da CPU e transforma dependências de controle em dependências de dados (isto é, execução predicada). Nós implementamos a operação de seleção próximo aos dados nas estratégias de execução de consulta orientada a linha/coluna/vetor para a arquitetura x86 e para nas duas extensões HMC-Scan e HIPE-Scan. Nossas simulações mostram uma melhora de desempenho de até 3.7× para HMC-Scan e 5.6× para HIPE-Scan quando executada a consulta 06 do benchmark TPC-H de 1 GB na estratégia de execução orientada a coluna.

Palavras-chave: SGBD em Memória, Cubo de Memória Híbrido, Processamento em Memória.

Abstract

A large burden of processing read-mostly databases consists of moving data around the memory hierarchy rather than processing data in the processor. The data movement is penalized by the performance gap between the processor and the memory, which is the well-known problem called memory wall. The emergence of smart memories, as the new Hybrid Memory Cube (HMC), allows mitigating the memory wall problem by executing instructions in logic chips integrated to a stack of DRAMs. These memories can enable not only in-memory databases but also have potential for in-memory computation of database operations. In this dissertation, we focus on the discussion of near-data query processing to reduce data movement through the memory and cache hierarchy. We focus on the select scan database operator, because the scanning of columns moves large amounts of data prior to other operations like joins (i.e., push-down optimization). Initially, we evaluate the execution of the select scan using the HMC as an ordinary DRAM. Then, we introduce extensions to the HMC Instruction Set Architecture (ISA) to execute our near-data select scan operator inside the HMC, called HMC-Scan. In particular, we extend the HMC ISA with HMC-Scan to internally solve instruction dependencies. To support branch-less evaluation of the select scan and transform control-flow dependencies into data-flow dependencies (i.e., predicated execution) we propose another HMC ISA extension called HIPE-Scan. The HIPE-Scan leads to less iteration between processor and HMC during the execution of query filters that depends on in-memory data. We implemented the near-data select scan in the row/column/vector-wise query engines for x86 and two HMC extensions, HMC-Scan and HIPE-Scan achieving performance improvements of up to 3.7× for HMC-Scan and 5.6× for HIPE-Scan when executing the Query-6 from 1 GB TPC-H database on column-wise.

Keywords: In-Memory DBMS, Hybrid Memory Cube, Processing-in-Memory.

Contents

List of Figures

List of Tables

List of Acronyms

AVX Advanced Vector Extensions.

CAS Column Address Strobe.

CISC Complex Instruction Set Computer.

CPU Central Processing Unit.

CWD Column Write Delay.

DBMS Database Management System.

DRAM Dynamic Random Access Memory.

FPGA Field-Programmable Gate Array.

GCC GNU Compiler Collection.

GPU Graphics Processing Unit.

HIPE HMC Instruction Predication Extension.

HMC Hybrid Memory Cube.

ILP Instruction Level Parallelism.

ISA Instruction Set Architecture.

MMX Multi-Media eXtension.

NDP Near-Data Processing.

NoC Network-on-Chip.

NUMA Non-Uniform Memory Access.

NVM Non-volatile Memory.

OoO Out-of-Order.

PIM Processor-in-Memory.

RAS	Row Address Strobe.
RAW	Read After Write.
RISC	Reduced Instruction Set Computer.
RP	Row Precharge.
SIMD	Single Instruction Multiple Data.
SSE	Streaming SIMD Extensions.
TLB	Translation Look-aside Buffer.
TSV	Through-Silicon Via.
VLIW	Very Long Instruction Word.
WAR	Write After Read.
WAW	Write After Write.

Chapter 1

Introduction

During the last decades the research in Database Management System (DBMS) has been driven by disk-based data processing. Recently, in-memory databases have been gaining importance as memories improved capacity and bandwidth, but also due to the decreasing price of DRAM. Storing the entire database in main memory provides faster access and real-time analytics. Besides, when we change the storage layer of the database from disk-based to main memory we also change the bottleneck to pointer-chasing, cache-unfriendly structures, memory hierarchy and concurrency control [56].

To overcome those bottlenecks, previous work have focused on the design of new algorithms and data structures to make a better usage of the cache memories and multi-core processing [21, 57, 52, 47]. Moreover, with the releasing of new hardware technologies such as Non-Uniform Memory Access (NUMA) architecture, Single Instruction Multiple Data (SIMD), Non-volatile Memory (NVM), Field-Programmable Gate Array (FPGA), on-chip Graphics Processing Unit (GPU) and HMC, new hardware-conscious alternatives were proposed to increase performance [39, 53, 8, 33, 9].

These new hardware have great impact on the design of hardware-conscious in-memory DBMS. In this dissertation, we focus on the design of the select scan database operator to process read-mostly On-Line Analytical Processing (OLAP) that requires moving lots of data across the new memory hardware HMC.

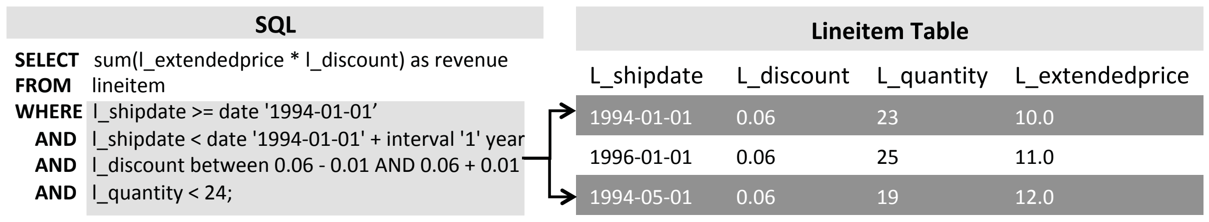
1.1 Problem

In the past decades, the disparity between processor performance and main memory latency has grown tightly, a well-known problem called "memory wall" [54]. The focus on CPU performance development and more recently on improving parallelism with multi-core processors resulted in performance improvements of 70% per year, while memory innovation focused on increasing capacity resulting in performance improvement of only 10% per year [28]. This increasing gap presents direct impact on large scale data processing, specially on in-memory databases.

In-memory databases became popular over the years due to the dropping cost per bit of DRAM 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"). Moreover, large data movement increases the cache pollution, once data is installed inside the cache line (by removing potentially useful data), it will never be used again.

1.2 Motivation

The *select scan* is commonly the first database operation applied during the processing of SQL queries, because of the larger data movement compared to other database operations when evaluating a restriction predicate to tuples. The common strategy to run the select scan, also called *push-down* optimization [27], is placing the restriction predicate as far as possible down the query plan (i.e., this means the operation in the bottom of the query plan is the one executed first). Figure 1.1 presents a SQL query applying predicates (WHERE SQL clause) and filtering tuples. The processing of this SQL query eventually calls the select scan operator, which requires moving tuples from memory to the processor across the cache hierarchy to apply the predicates to these tuples. However, in OLAP databases most of the data placed in the cache hierarchy is barely used due to the ad-hoc nature. The scanning of the same column occurs only a few times caching lots of irrelevant data with potential cache latency increase when accessing unused columns. Therefore, the select scan does not benefit from cache with low data reuse.



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;
```

Lineitem Table

L_shipdate	L_discount	L_quantity	L_extendedprice
1994-01-01	0.06	23	10.0
1996-01-01	0.06	25	11.0
1994-05-01	0.06	19	12.0

Figure 1.1: The Query 06 from the TPC-H Benchmark applying the select scan filter in the Lineitem table.

The emergence of smart memories, such as the HMC[9], inverts the common data processing approach by moving computation to where data resides with many benefits, such as reducing energy consumption and providing faster response times [41]. The HMC is a 3D die-stacked technology with stacks of DRAM logically split into 32 independent vaults, interconnected using Through-Silicon Via (TSV) [11] to a logic chip at the end of the stack. The logic chip executes in-memory instructions with high level of parallelism.

The releasing of 3D-stacked technologies like the HMC and the increasing demand of processing large databases motivated the resurgence of moving the computation near data to reduce data-movement also called Near-Data Processing (NDP). Therefore, our goal is to benefit from the HMC to process NDP select scan avoiding data movement and boosting response time.

1.3 Research Question

In order to minimize the memory wall problem for database systems and benefit from NDP, the first question that we focus on, is: “What happens when database systems run the select scan over the current x86 architecture using the HMC as ordinary DRAM main memory?” We detail the execution of a microbenchmark using the column/tuple/vector-wise query engines to set the baseline.

Next, we discuss how to bypass the cache hierarchy of the x86 and process the select scan inside the HMC, focusing on the second question: “Can we use the current HMC ISA to implement near-data select scan?” Considering that the current HMC ISA is formed by read-operate and read-modify-write, all restrictive atomic update operations over a word of 16 bytes [32, 35]. We can infer that the internal DRAM 256 bytes row-buffer (also called DRAM page) is underused, and we also need to interleave x86 and HMC ISA instructions to execute the select scan. However, by interleaving instructions it often requires transferring the results of the database operations between processor and memory impacting the pipeline efficiency.

Finally, the last question that we focus on to implement the near-data select scan is: “What are the extensions required on the HMC ISA to leverage the full potential of the hardware and reduce the interleaving with x86 instructions?” From this point, we analyzed the necessary extensions that could bring improvements for database systems when execute the select scan operation right inside the HMC while also mitigating the memory wall problem.

1.4 Contributions

In order to set our baseline, we evaluated the main query execution engines using the HMC as ordinary DRAM. As result, we discuss the benefits and limitations of such execution by performing a detailed analysis of the select scan execution. Besides, we conduct an analysis by using the instructions from the current HMC ISA to evaluate the predicates in the select scan operation.

In this dissertation we proposed extensions to the current HMC ISA to perform database systems operations. First, we propose to add extra functional units to perform the select scan over the whole row-buffer available of 256 bytes. With a bigger operation size, our goal is to reduce DRAM accesses and data movement during the push-down operation. Second, we implement the loop unrolling technique to send up to 32 independent instructions at a time to the HMC vaults and achieve a higher level of parallelism.

After performing the initial evaluations we observed three major limitations in the HMC ISA: First, it can only perform update instructions (operations over a single memory address), not being able to perform operations between two distinct addresses. Second, it cannot solve data dependency between instructions internally, relying on the processor to perform such task. Third, control flow dependencies also need to be handled by the processor.

With such limitations in mind, we present extensions to support predicated execution of the select scan in the HMC, called HMC Instruction Predication Extension (HIPE). HIPE can solve internally data-flow dependencies by using internal register bank and can also transform control-flow dependencies into data-flow dependencies by supporting predicated instructions.

This leads to less iteration between Central Processing Unit (CPU) and HMC on the execution of query filters that depends on in-memory data. Cycle accurate simulations mimicking the column/tuple/vector-wise query engines show performance improvements of 3.69× for HMC-Scan and 5.44× for HIPE-Scan when executing Query-6 from 1 GB TPC-H database on column-wise. Besides, experiments varying the selectivity factor, when compared to x86 baseline showed gains of 4.84× for HIPE-Scan using predicated execution extension over low selectivity scenario, and 5.64× for HIPE-Scan without predication on high selectivity scenario. Overall, in this dissertation we provide the following main contributions:

1. We analyze the execution of the select scan with column / tuple / vector-wise query engines over the traditional x86 using the HMC as ordinary DRAM. This analysis sets the baseline to compare with our extensions to the HMC ISA.

2. We analyze the current HMC ISA by replacing the predicate evaluation instructions in order to reduce data movement

3. We extend the HMC ISA to take full advantage of the DRAM row-buffer. This extension reduces DRAM accesses at the same time reduces the amount of requests in the push-down of the select scan.

4. We implement the loop unrolling technique to better use the multiple vaults. This extension increases the parallelism of the select scan.

5. We extend the HMC ISA to support data dependencies and branch-less decisions when the select scan evaluates query filters. This extension, called HMC predicated execution (HIPE), mainly changes from control-flow, where the evaluation of the filters happens in the processor, to in-memory data-flow with less interleaving between HMC and the processor.

6. We discuss pros/cons of our extensions when the select scan is implemented on top of the main query engines: column-at-a-time, tuple-at-a-time, and vectorized. Our experiments showcase the potential of our near-data select scan running a micro-benchmark of the TPC-H of 1 GB.

The remainder of this dissertation is organized as follows: In Chapter 2 we give an overview of modern computer architectures and the HMC architecture, besides we describe databases systems and the predicate processing in row-stores and column-stores. In Chapter 3 we present the related work on Processor-in-Memory (PIM),NDP and hardware conscious DBMS. In Chapter 4 we present our proposals for architectural extensions that perform the HMC push-down and the HMC predicated execution of query filters. The experimental setup, methodology and evaluation results running a TPC-H micro-benchmark is described in Chapter 5. Finally, in Chapter 6 we discuss our conclusions and point out future work.

Chapter 2

Background

In this chapter we introduce concepts from the field of computer architecture and database systems that will be used throughout this dissertation. The preliminaries behind many of the terms and definitions used in this dissertation lies in modern computer architecture, so a briefly discussion about CPU and HMC architecture is provided in Section 2.1. Next, we present a general introduction on databases in Section 2.2, followed by a discussion about storage models and query execution engines implemented in the state of the art DBMS architecture. Implementation details that will be relevant to the task of integrating query processing with HMC processing-in-memory approach are highlighted in Section 2.32.3.

2.1 Modern Computer Architecture

In this section, we describe the main concepts of modern computer architecture that are most important for query processing in database systems. We also describe the architecture of the HMC since its an important component used in this dissertation.

2.1.1 Processor Architecture

During the processor design, one of the first decisions to be made is regarding the instructions supported, this definition establishes the ISA on the processor. This ISA also defines the instruction format, the amount of registers visible to the programmer and compiler, and also the addresses modes used on the instructions.

One ISA can be classified as Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC). CISC architectures usually have many specialized and complex instructions, with multiple instruction addresses modes and support memory accesses on many different instructions. On the other hand, RISC architectures simplifies the processor by assemble more simple and basic instructions, with fewer addresses modes and these are usually load/store architectures, which means that only specific load and store instructions may access the memory.

Each specific architecture can be implemented in different ways, called organization or micro-architectures, such as single-cycle, multi-cycle, pipelined and others. Although processors still being produced with single- and multi-cycle organizations, for high performance purposes (desktop and servers) the pipeline is mandatory for performance.

The pipeline can be implemented with as many stages as wanted by the designer. However, nowadays, Intel processors usually have between 12 and 16 stages. Basically, inside the pipeline, multiple instructions are being executed in a given instant, similar to an assembly line. Thus, each pipeline stage is responsible for an specialized part of the instruction execution.

A very widespread pipeline organization is the five-stage one from MIPS, it means that each instruction pass throw a determined stage in a total of 5 stages. Figure 2.1 presents the pipelined execution on the top. During the processing the first stage is the *Instruction Fetch* (IF) responsible to get the next instruction based in the *program counter* (PC). The second stage is the *Instruction Decode* (ID) where the instruction is translated from the ISA to an internal representation. The third stage is the *Execute* (EX) when the requested instruction is performed by a determined functional unit. In the fourth stage, a *Memory Access* (MEM) is performed when the instruction requests a store or load operation. Finally, the *Write-back* (WB) is the final stage when the execution result is saved in the register bank.

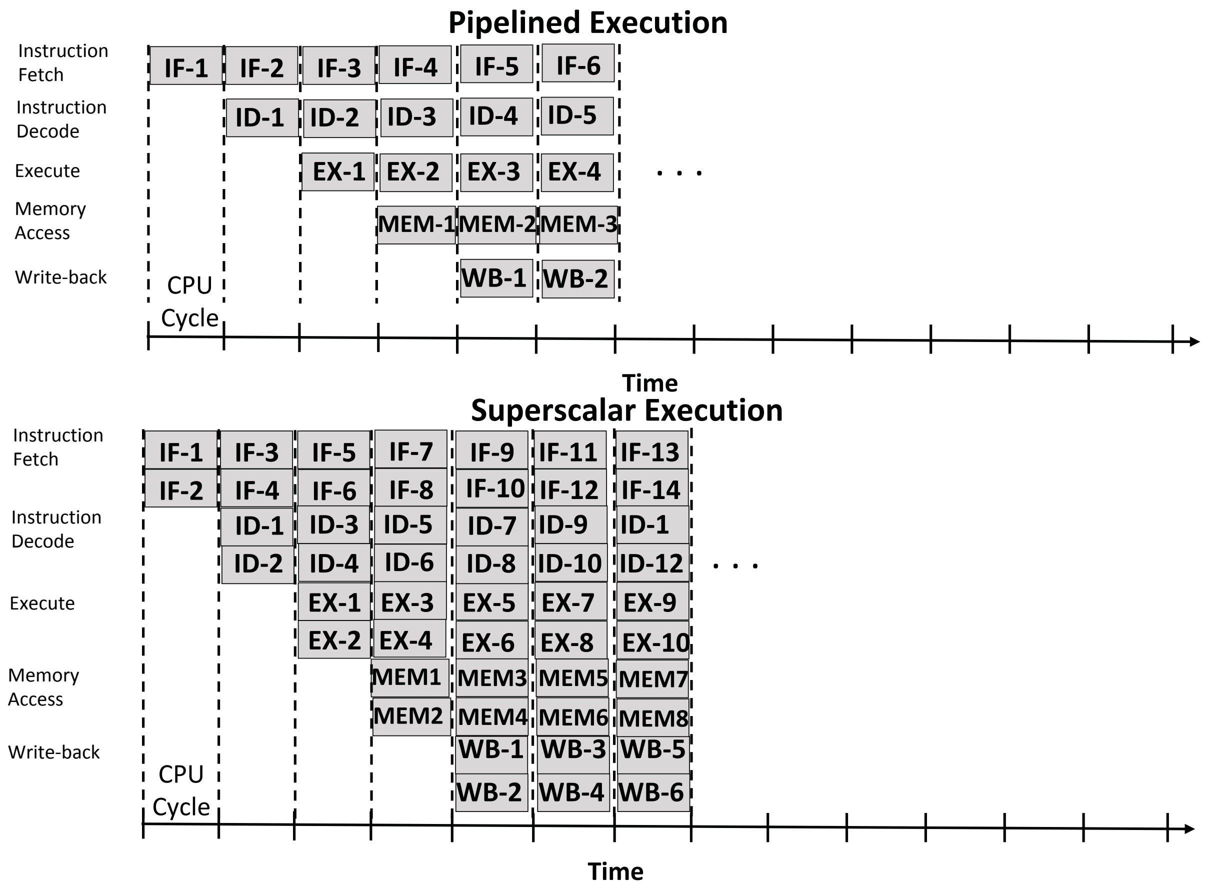


Figure 2.1: Pipelined vs Superscalar execution

In order to increase this Instruction Level Parallelism (ILP) inside the processor and to gain performance, the pipeline can be superscalar, which means that instead of one instruction be present on each stage, the circuit supports multiple instructions per stage as represented in the bottom of Figure 2.1. Considering the higher instruction parallelism inside the superscalar pipelines, we can decide if the instructions will be executed on in-order or Out-of-Order (OoO) fashion. General purposed processors with focus on high performance usually adopts OoO execution inside its superscalar pipelines. The basic OoO processor can be divided into front-end and back-end. Basically, the front-end is responsible for fetching new instructions from the instruction cache memory (fetch stage), and also responsible for decode the operations into micro-operations (μops) (decode stage). The back-end is responsible for performing μops register renaming of the operand (rename stage), dispatch the μops for the functional units (FUs) (dispatch stage), execute the μops inside the FUs (execution stage) and retire the ready μops (write-back stage). Figure 2.2 presents a diagram representing a superscalar processor adpting OoO.

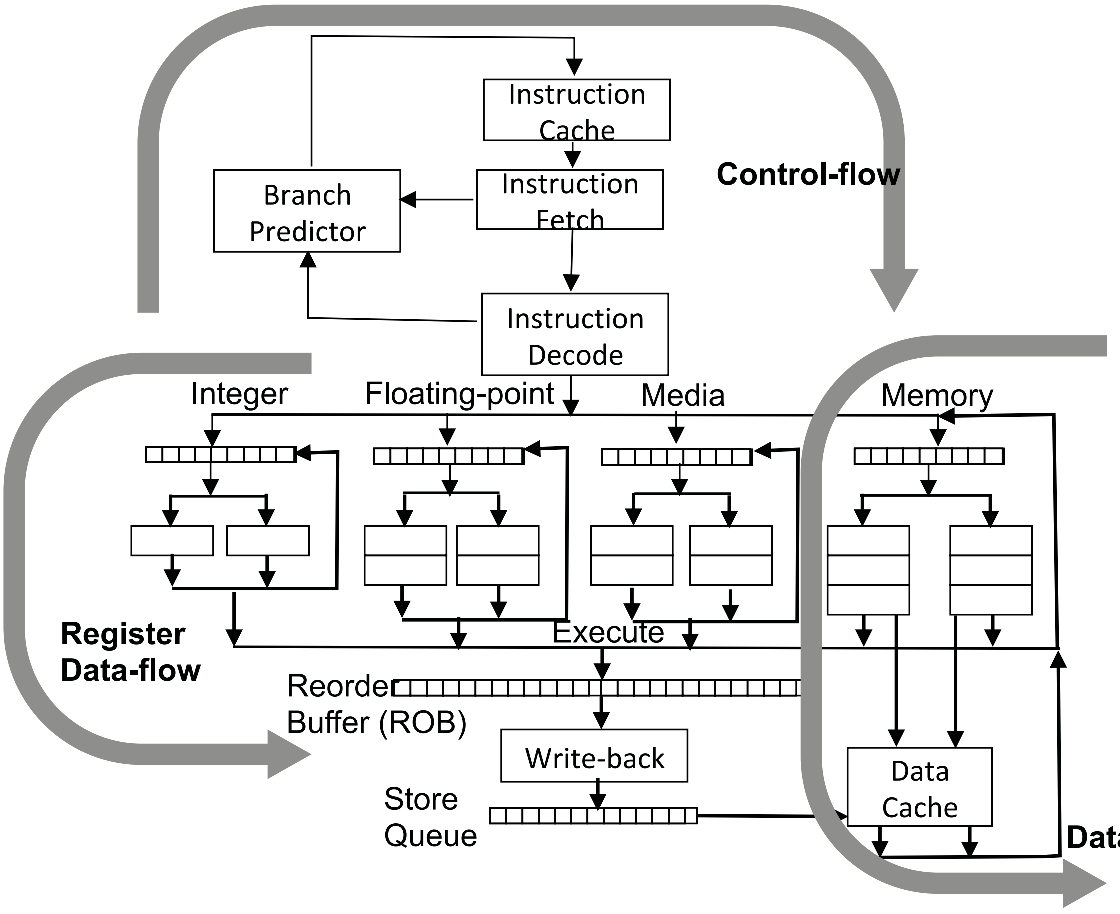


Figure 2.2: Superscalar out-of-order diagram

In order to achieve parallelism in the pipeline execution it is critical to determine whether there is a dependence between instructions. The dependencies can be classified into two types, data and control dependencies. Here we will only focus on true data dependencies know as Read After Write (RAW), as false data dependencies, know as Write After Write (WAW) and Write After Read (WAR), are easily solved by the rename stage of all modern processors.

- **Data dependence** occurs when the operands of an instruction depends on the operands of another instruction. In this way, the processor must be ready to deal with a chain of dependencies between the instructions, where, the result of one operation may be required by another.
- **Control dependence** occurs when the execution of the instruction is dependent from a branch instruction decision. To overcome pipeline stalls due to control dependencies, processors traditionally uses advanced branch predictors to predict branch target address and direction speculating the execution of further instructions, until the branch itself is executed. For instance, considering the select scan database operation over two columns in a conjunction, a determined tuple from the second column will be evaluated only if the predicate evaluation is true for the first column.

In order to also overcome control dependency, the architecture designs also employees predicated instructions. This concept was widely used on Very Long Instruction Word (VLIW) processors [22], in order to create super-blocks, which means that, two or more basic blocks could be fused into a single one. Predicated instructions are similar to regular instructions, however, the instruction will only perform write-back to the register bank depending whether a given condition is true or not. In this way, a control-flow dependency can be transformed into a data-flow one. Although such technique be present in nowadays processors, our tests in laboratory shows that GNU Compiler Collection (GCC) compilers hardly uses such instructions, even with optimizing compiler flags.

While OoO processors uses aggressive techniques to exploit possible parallelism among the instructions. In order to further exploit the data parallelism present on the applications, modern processors provides vector instructions that can apply one operation over multiple data elements, these instructions are also called SIMD instruction. The vector support on the processor is composed of vector functional units, vector registers, vector load/store unit and a set of scalar registers. These SIMD instructions were widespread by the Multi-Media eXtension (MMX) technology and more recent processors have the Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) technologies. The AVX 128-bit is supported in the Sandy-Bridge processor and its vector instruction can operate over 16 single-byte values at once [28].

2.1.2 Memory Hierarchy

In order to provide data for the increasing number of instructions being executed in parallel, the memory should provide huge storage with fast access. However, these two objectives are incompatible for the nowadays memory technologies. In order to overcome this demand, the memory hierarchy was developed, providing the notion of fast and unlimited memories by using small and fast caches nearby the processor, and huge storage capabilities with DRAM and disks far-away the processor [28]..

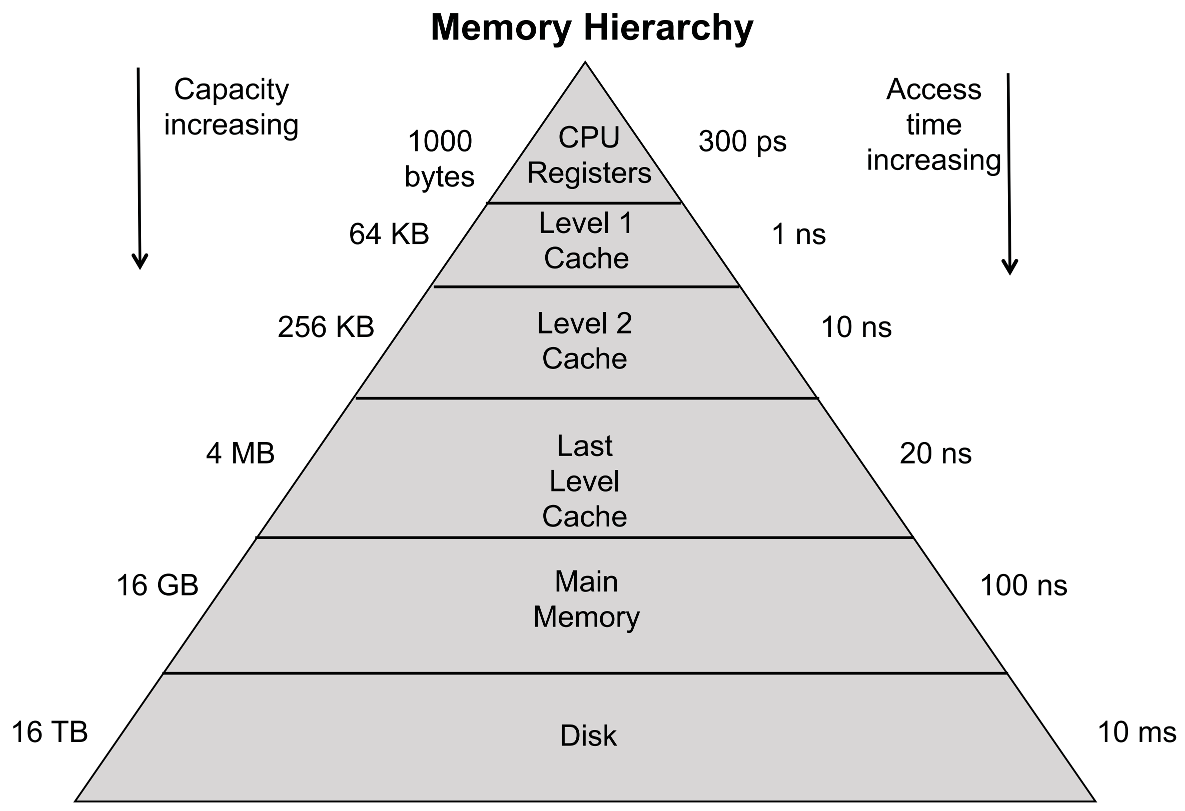


Figure 2.3: Memory hierarchy reference and latency for each level [28]

The memory hierarchy is organized into a few levels, each of them smaller, faster and more expensive than the next lower level. During a memory request, each level of the hierarchy usually is searched in a sequential way. Figure 2.3 shows the multilevel memory hierarchy and relates the latency and capacity of each level. The latency changes by a factor of 10^9 from the highest to the lowest level while the capacity grows by a factor of $10^1 2$.

The memory hierarchy was developed considering that most of the programs do not access instructions and data uniformly, but in temporal and spatial locality, meaning that after a certain memory access, both nearby addresses and recently used addresses have high probability to be accessed. Nevertheless, it is important to notice that application that do not present temporal or spatial locality for a specific hierarchy level, that level will increase the data access penalty. For instance, consider a streaming application which loops over a 128 KB vector. During the execution, a regular L1 cache of 64 KB shall not present cache hits for the requested data, because the data-set is bigger than the L1 cache. In this way, the application do not have any locality that the L1 cache can exploit. Furthermore, due to the sequential request in the cache hierarchy, the L1 cache will increase the access latency for the requested addresses by this application.

Therefore, we can see that database operations which present streaming behavior may not benefit from the cache hierarchy, instead, it will suffer higher penalties due to the traditional cache organization. In this way, processing performed near-memory architectures, such as HMC, may benefit such applications, by avoiding large data movements through the memory hierarchy.

2.1.3 HMC Architecture

Figure 2.4 illustrates the overall system architecture of the HMC. It integrates multiple DRAM dies and a single logic die, where each memory die has multiple DRAM banks. Typical DDR-3 DRAM modules are organized in 4 8-KB rows, while the HMC DRAMs uses small rows of 256 bytes providing lower energy consumption and faster accesses.

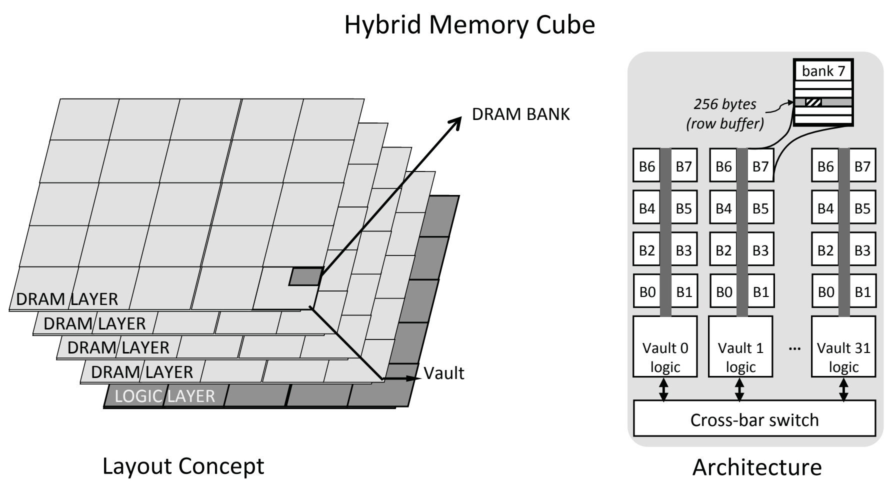


Figure 2.4: HMC overall layout and architecture.

Every set of 3D stacked banks forms a vertical group called vault interconnected by a TSV bus. In the current specification [32], the HMC provides 32 vaults that can independently access DRAM banks with a potential internal high bandwidth up to 320 GB/s.

The logic die 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 controller maintains a queue with references to the memory addresses. The vault controller may executes a memory reference based in need rather than order of arrival which makes that the result can be propagated out of order. Nevertheless, requests from the same serial link will be executed in order.

The available bandwidth of all collective vaults are accessible to I/O links connected to a crossbar switch. The communication is done by single interconnect packets similarly to those used by Network-on-Chip (NoC) systems [10].

Each vault has a memory controller and also its independent functional units enabling in-memory processing. The processor sends simple operations such as load/store instructions, while the common complex DRAM operations such as Row Precharge (RP), Row Address Strobe (RAS), Column Address Strobe (CAS), Column Write Delay (CWD) are managed by the memory controller of each vault.

The vault controller performs arithmetic and logical update atomic instructions with operands size of up 16 bytes. They are variants of read-modify-write operations supported by some memory controllers [42]. This kind of operation normally reads data from any memory location, operates over data, then writes the result back to same memory location. Once the operation finishes, the old data, modified data or operation status is returned back to the processor.

In the current set of instructions, the HMC allows in-memory processing, but the processor still needs to wait for the results in order to send this 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 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.

2.2 Relational Database Systems

A database management system (DBMS) is a set of programs that provides tools for users and applications to store and access data. It admits multiple concurrent users to query and modify data by using a high level language, such as SQL.

The most used type of a DBMS is the relational. A relation is a set of tuples, also called rows, and attributes, also called columns. A database is a collection of relations stored and managed by a DBMS. The database is defined by a schema that specifies the tables, attributes in each table and how tables are related to each other. Figure 2.5 describes a table and a database R.

The *relational algebra* can be used to define the operations over a relation, including projection (π), selection (σ), aggregation, Cartesian product ($\times$) and a variable amount of joins ($\bowtie$). For example, considering the Relation R presented in Figure 2.5 to perform a query retrieving the tuples which the language is German, one could use the following expression in relational algebra:

$$\pi_{(regionkey,language,country)} = (\sigma_{language='German'}(R))$$

The physical representation of the query from the relational algebra includes a set of correspondent physical operators combined in a *query execution plan*. The select scan is one of the physical operators and it is used to filter tuples matching a determined predicate. For instance, in Coding 2.1 the predicates are the conditions coded in the WHERE clause to filter tuples with the SQL, such as *"l_shipdate >= date'1994-01-01"'*.

2.2.1 Predicate Processing In Row-Stores

Traditionally, the storage layout implemented by most of the database management systems (DBMS) is the N-ary Storage Model (NSM) and the systems are called row-stores. Figure 2.5 despicts an example Relation R and the respective NSM page. The tuples are contiguously stored in each disk page, putting the first tuple at the beginning and placing an offset table at the end of the page to locate the starting point of each tuple.

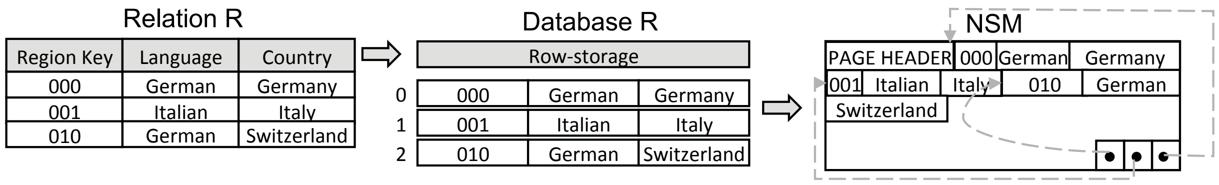


Figure 2.5: Relation R stored in a N-ary Storage Model (NSM).

In this layout the query execution layer, generally implements the "Volcano-style" iterator model[24] also referred as "tuple-at-a-time" processing model. It consists of *open()*, *next()* and *close()* methods iterating over each tuple and applying the correspondent predicate.

```
SELECT
  sum(l_price * l_disc) as revenue
FROM
  lineitem
WHERE
  l_date >= date '1994-01-01'
  AND l_date < date '1994-01-01' + interval '1' year
  AND l_disc between 0.05 AND 0.07
  AND l_quant < 24;
```

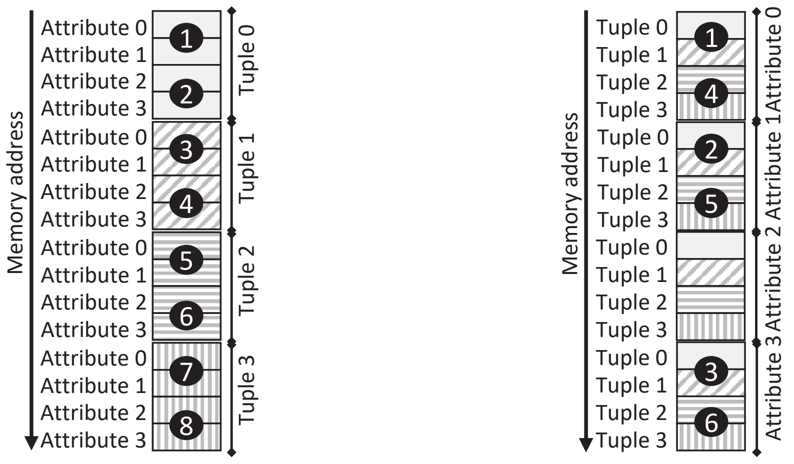
Código 2.1: Query 6 from TPC-H in SQL code.

Figure 2.6(a) 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 [24] 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 [3].

2.2.2 Predicate Processing In Column-Stores

An alternative to avoid polluting cache for read-mostly databases is the Decomposition Storage Model (DSM) [17] or column-store. Figure 2.7 describes the Relation R represented in DSM. In this layout, the data is organized in vertical partitions of attributes contiguously stored in memory pages. Different from row-stores, in the column-stores only the needed attributes traverse the memory hierarchy and the contiguous storage guarantees good cache locality. Figure 2.6(b) represents how the data is organized and processed in column-stores. Considering a database query evaluating Attributes 0, 1 and 3, the Attribute 2 is not load during the processing.

There are two strategies to execute query operations in columns-stores: "column-at-a-time"[12] and "vector-wise"[15]. In the first strategy, a query operator consumes and generates an array of values for each column. However, large volumes of intermediate results reside in memory from early stages of materialization, which may cause increase of data cache miss and I/O overhead [1].



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

Figure 2.6: Database storage and selection scan execution order.

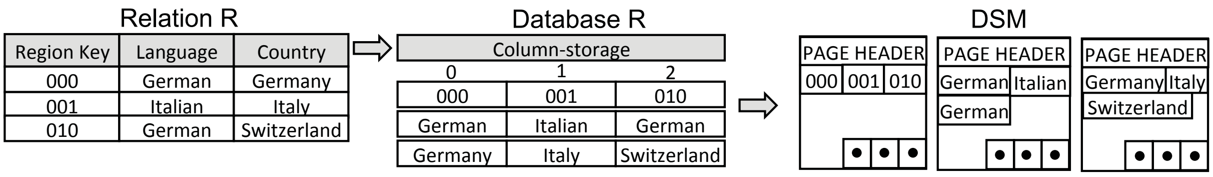


Figure 2.7: Relation R stored in a Decomposition Storage Model (DSM).

In the second strategy, the execution is focused on processing column chunks, called vectors [15] that are sized to fit the cache (normally $N = 1000$). Operations happen over the vectors for better use of the CPU cache. Modern column-stores implement the vectorized execution with late materialization [2]. In this type of data materialization, database operators generate a list of vector indices with qualified data, instead of values. For instance, a condition in the WHERE clause only returns the indices of qualified data in the vectors that are used as entries by the next condition, and so on. Tuples are materialized as late as possible by stitching together the required attributes with the remaining indices.

2.3 Select Scan Strategies on Modern CPUs

The select scan operator combines a scan with a predicate evaluation in order to filter data. There are three universal accepted strategies to implement the select scan operator, namely branching variant, branch-less variant (i.e. predicated execution) and vectorized variant. Besides, the compiler can apply loop unrolling optimization to improve performance. Considering the implementation of these optimizations, modern CPU can be better exploited by improving pipeline execution and data parallelism.

2.3.1 Branching Execution

The branching variant for the select scan operation consists on a loop iterating over a vector input and a predicate evaluation composed of an if-statement. Algorithm 1 describes the branching version. This implementation writes the result only when occurs a match in the evaluation.

Algorithm 1 Select Scan (Branching)

```
1: $j \leftarrow 0$   // output index
2: for $i \leftarrow 0$ to $colSize - 1$ do
3:     if $column[i] < predicate$ then
4:         $output[j] \leftarrow i$
5:         $j = j + 1$
6:     end if
7: end for
```

Although the implementation seems simple, it benefits only evaluations with high selectivity (i.e. only a few tuples matches), otherwise the CPU incurs a considerable amount of branch misprediction causing extra performance penalties [16].

2.3.2 Predicated Execution

The predicated execution can be used in the select scan to minimize the penalties by using branching decisions. The predicated execution strategy is a branch free alternative to make evaluations avoiding the penalties of mispredictions. The strategy shown in Algorithm 2 consists of an evaluation that always writes the result and executes the next instructions taking into account the previous evaluation in a linear way.

Algorithm 2 Select Scan (Predicated)

```
1: for $i \leftarrow 0$ to $colSize - 1$ do
2:     $m \leftarrow (column[i] < predicate?1 : 0)$
3:     $bitmap[i] \leftarrow m$
4: end for
```

2.3.3 Vectorization

Modern CPUs were designed with new vector units to allow the execution of a single instruction over multiple data (i.e. SIMD) improving data parallelism. The SIMD select scan variant applies a predicate over a vector of values generating a bitmask with 1 for matches and 0 otherwise [58]. The result extracted from the bitmask is used to perform the next operations into a database query plan. The Algorithm 3 describes the SIMD predicate evaluation and a selective store for the result.

Algorithm 3 Select Scan (SIMD)

```
1: for $i \leftarrow 0$ to $simdArray - 1$ step $i+ = simdLenght$ do
2:     $mask \leftarrow simd(column[i : i + simdLenght - 1] < predicate)$
3:     $bitmap[i : i + simdLenght - 1] \leftarrow mask[0 : simdLenght - 1]$
4: end for
```

2.3.4 Loop Unrolling

The Loop Unrolling is a well known technique used in compilers to generate an optimized code for modern processors [5]. The main motivation is to explore the CPU pipeline by replicating the body of tight loops, or loops with few instructions, with benefits in ILP, register locality and avoiding pipeline stalls [49]. The Algorithm 4 describes the implementation of the loop unrolling technique over a tight loop. In this case, the depth of each iteration is 4 and the body is replicated in order to achieve a good instruction parallelism into the pipeline and also avoid stalls.

Algorithm 4 Select Scan (Loop Unrolling)

```
1: $j \leftarrow 0$   // output index    // i += 4
2: for $i \leftarrow 0$ to $colSize - 1$ do
3:     if $column[i] < predicate$ then
4:         $output[j] \leftarrow i$
5:         $j = j + 1$
6:     end if
7:     if $column[i+1] < predicate$ then
8:         $output[j] \leftarrow i$
9:         $j = j + 1$
10:    end if
11:    if $column[i+2] < predicate$ then
12:        $output[j] \leftarrow i$
13:        $j = j + 1$
14:    end if
15:    if $column[i+3] < predicate$ then
16:        $output[j] \leftarrow i$
17:        $j = j + 1$
18:    end if
19: end for
```

2.4 Conclusion

In this chapter we introduced the overall concepts used in this dissertation. We describe the construction of DBMS and predicate processing in the x86 hardware architecture. The reasoning behind many of the algorithms and methods used in this dissertation lies in the architecture of HMC and modern processors, thus we described the main hardware optimization. In the next chapter, we discuss previous work related and the their limitations covered in this dissertation.

Chapter 3

Related Work

In this chapter, we present related work alternatives to minimizing the memory wall problem. We briefly discuss the ideas behind processing-in-memory (PIM), then we focus on 3D stacked technologies and near-data processing approaches using the HMC as example. Finally, we present some work in hardware-conscious and near-data processing for database systems.

3.1 From Processing-In-Memory to Near-Data Processing

The concept behind mixing memory and logic motivated works since the 60s [34] when it was first investigated the ideas behind PIM. The use of processing units placed near memory in previous works [44, 23] brought opportunities to achieve massive parallel SIMD processing while also alternatives to position ALUs with memory arrays.

Previous work in PIM [26] proposes a PIM architecture with an external host processor to overcome the problem of data-movement throw the memory hierarchy in data-centric applications. The architecture aimed to execute selected instructions in memory reducing the amount of data-movement across the processor-memory interface. However, for many years the PIM ideas did not come out as a real alternative for solving the memory wall problem due to the technology limitation. Recently, the releasing of 3D stacked technologies made processing right inside memory became tangible by placing logic and memory in different chips but in same memory package [9].

3.2 3D Stacked Technology and Hybrid Memory Cube

The use of 3D stacked memories placed inside GPUs was also considered in previous work. A transparent instruction offloading was proposed in order to enable near-data processing in GPUs [31]. The goal was identifying candidate blocks in the compiled code and dynamically offload the blocks to memory using run-time information. However, this design requires control over the program execution, which may lead to unnecessary resource use when there is no instructions to be offloaded. Besides, it benefits only GPU workloads and the data-movement in the main memory and processor hierarchy still occurs.

There are also work considering the use of smart-SSD devices for processing database systems operations [18], such approach only benefits when the database cannot fit in main memory. Besides, there is also research for mitigating the performance gap between DRAM and FLASH devices.

With focus on the usage of as main-memory and HMC co-processor, an architecture extension was proposed to process MapReduce workloads [46]. The proposal incorporates simple processing cores at the logic layer in order to perform efficient map operations with a good memory access.

We also considered the usage of re-configurable huge functional units to process large amounts of data with register banks inside the HMC [6, 48]. 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.

Despite some work [59, 20] considered that the thermal limit of 85ºC for the HMC must be respected by architecture extensions, in this work we have not evaluated such impact since this evaluation is out of the scope of this dissertation. In order to allow our architecture extension operation we considered a cooling system similar to the proposed in previous work on HMC characterization [25].

3.3 Hardware Conscious and Near-Data Processing

The impact of the memory-wall problem motivated several work in the field of database systems over the past decades focusing on algorithms to exploit the cache benefits [54, 4, 14, 13]. There were also efforts to build hardware-conscious database systems in many directions, including, cache-conscious adaptive indexes [30] and NUMA-aware algorithms [38, 45, 19]. Nevertheless, none of them explored the data-movement mitigation, since they focus on avoiding unnecessary DRAM accesses by making use of relevant data in cache.

In the context of near-data processing, the work called JAFAR [55] was presented as an external DRAM accelerator to push down select scan operations near-data in DDR-3. The architecture presented in 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 which provides instructions that can be triggered by the processor to execute the select scan without the necessity of coordination to access external hardware. This design choice maintains the common out-of-order execution and the address translation from the Translation Look-aside Buffer (TLB) while also allows different ranges of word sizes up to 256 bytes to better usage the HMC row-buffer.

The use of the HMC as main memory was also evaluated in DBMSs by placing an accelerator inside the logic layer of the HMC, but to support join algorithms [40]. The proposal redesigns the hash and merge join algorithms in order to minimize the single word access (e.g., 16 bytes) avoiding 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.

The use of predicated execution was investigated in several work [36, 37] over the past decades, but none of these previous work implemented predication on smart memories.

3.4 Conclusion

Considering the discussions in this section, none of the past work analyze the current processing support of the HMC over read-mostly workload in row/column/vector-wise engines, neither they analyze further modifications in the logic layer of the HMC to fully execute database instructions in order to reduce data-movement.

Our approach using HMC instructions explores the logic layer of HMC to process data with native HMC operations. It adds architectural modifications in the operation size and supports for predicated execution to diminish accesses to DRAM while also provides high levels of parallel processing. Furthermore, we evaluated the execution of the main query execution engines in HMC demonstrating its potential for both row-stores and column-stores systems. In the next Chapter we present our proposals and discuss how each research question is answered.

Chapter 4

Near-Data Scan for Database Systems

In this chapter, we focus on the questions raised in Chapter 1 to present our proposal and define the extensions for the HMC logic layer. First, we discuss the select scan execution with the x86 implementation and the HMC as ordinary DRAM. Then, we propose an extension to the HMC ISA to leverage the full potential of the hardware by adding a comparison instruction and vector processing units. Then, we propose another extension to allow predicated execution and reduce the interleaving with x86 instructions.

4.1 Query Processing in HMC as DRAM

With the HMC at hand, the first question we focus on is: "What happens when database systems run the select scan over the current x86 architecture using the HMC as ordinary DRAM?" Let us consider the "column-at-a-time" [12] execution model to evaluate the select scan in the query plan. Figure 4.1 depicts a select scan in 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.

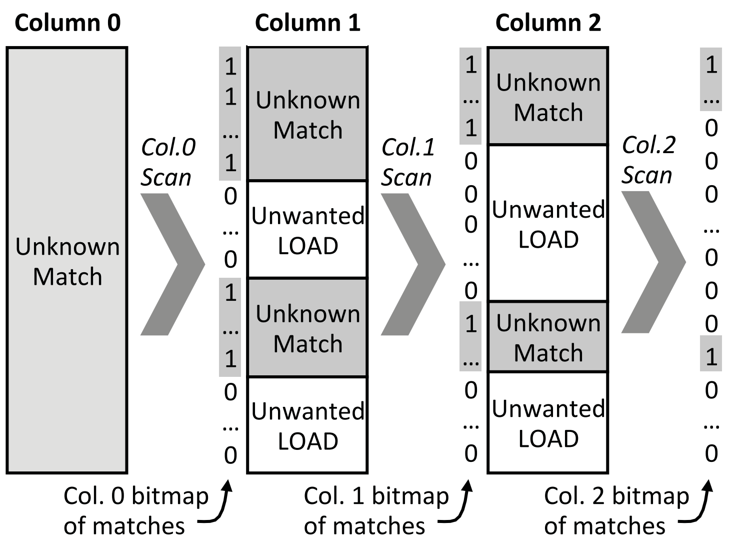


Figure 4.1: Select scans in the "column-at-a-time" model.

Figure 4.2 illustrates the data movement to process the select scan operation with the current x86 architecture instructions and the HMC placed as main-memory. In this scenario, the assembly instructions to process select scans keep up unmodified.

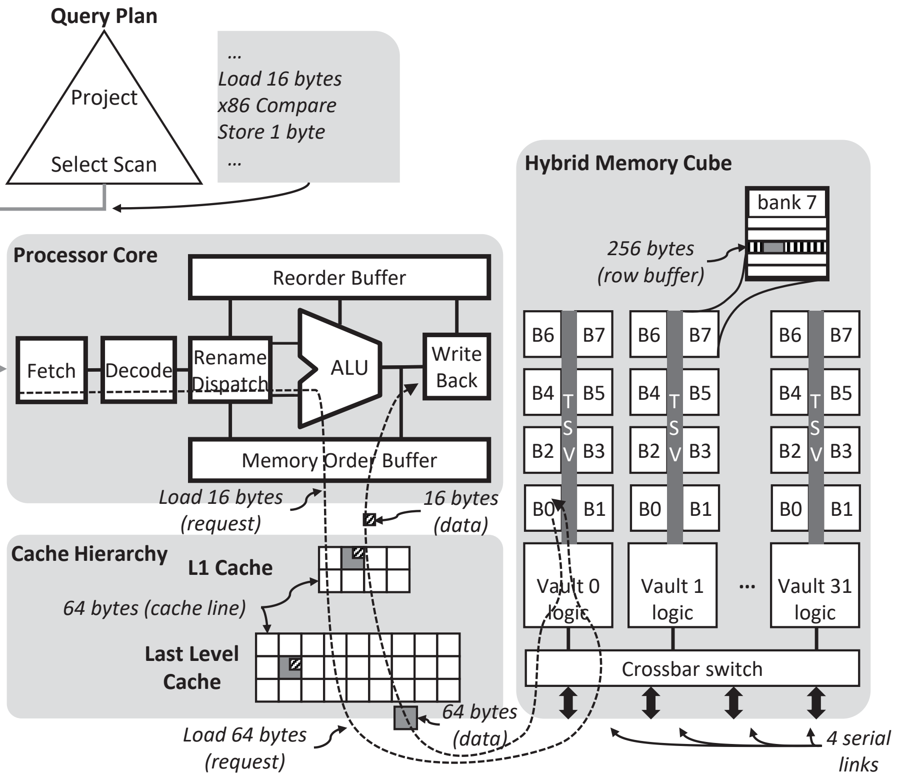


Figure 4.2: Traditional x86 processing using the HMC as DRAM.

Initially, the x86 assembly instructions are allocated in the processor pipeline. In current x86 architecture with AVX-128 approach, 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 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, but the HMC placed as main-memory provides 256 bytes per access in the buffer. Afterwards, only 64 bytes are returned back to cache and, when data is positioned in cache, the processor only operates over 16 bytes to finally evaluate query filters for the column chunk.

We run a micro-benchmark to understand the execution of the select scan using the DDR-3 and HMC in two ways: (1) the x86 processor executing the select scan with the HMC as DRAM and (2) the HMC executing the select scan replacing the x86 assembly instructions for those of the current HMC ISA specification. We discuss the changes in the assembly code of the query and how they are executed later on in this paper, but basically we swap x86 compare instructions to HMC compare instructions and the execution pipeline sends the instruction to execute in the HMC. In our motivation experiment, we execute the TPC-H Query 06 with the traditional strategies: “tuple-at-a-time” [24], “column-at-a-time” [12] and “vector-wise” [15].

Figure 4.3 shows the response time of Query 06 in 1 GB database in the x86 and in the HMC architectures. We provide further details of the execution environment in Section 5. When we put the HMC as ordinary DRAM the select scan achieves up to 65% of better performance compared with DDR-3. This occurs due to high bandwidth provided by the HMC and the Dynamic Random Access Memory (DRAM) operations managed by the vault controller. Comparing x86 with HMC as DRAM and HMC executing comparison instructions, we observe that the x86

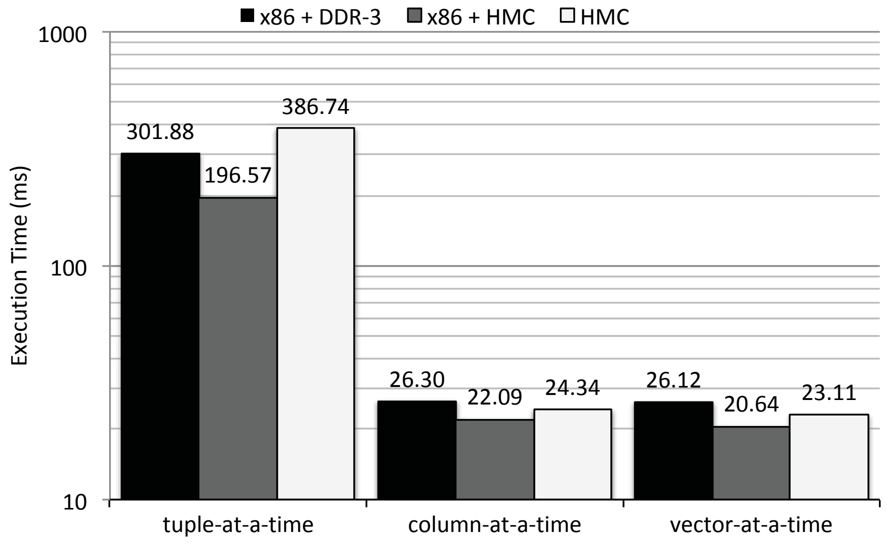


Figure 4.3: TPC-H Query 06 micro-benchmark: the x86 instructions over the DDR-3 and HMC as ordinary DRAM and the HMC instructions.

processor still presents the best response times when running the select scan rather than in the HMC no matter the query engine. Although the HMC ISA was proposed to optimize in-memory operations, the problem when simply running the select scan in the HMC is related to the current HMC ISA. First, there is only single memory address operations (update instruction), not being able to perform operations between two distinct addresses, only between address and immediate. Second only compare-and-swap instructions are available, making it costly to operate over data, since data may be modified after the compare operation to evaluate query filters. Third, the instructions operate over 16 bytes of data wasting the potential of the hardware. We observe that the column and vector-wise engines suffer less impact with the small 16 bytes load request, because only the requested columns in the query statement move data around (or the bit-vector in the vector-wise). Fourth, the processor only triggers HMC instructions enough to use only few parallelism between the vaults (i.e., only a small portion of parallelism is explored).

4.2 Select Scan Push-Down to HMC

In this section, we expose what happens to the select scan when extrapolating some architectural limitations Therefore, we present two extensions to the logic layer of the HMC to execute the select scan (HMC-Scan): (1) a comparison instruction between address and immediate (i.e., HMC compare instruction to evaluate query filters) and (2) vector functional units to utilize the DRAM row-buffer available of 256 bytes. Our goal with the HMC-Scan is to bypass the cache hierarchy reducing DRAM accesses and data movement in the push-down [50].

Figure 4.4 depicts our extension. In this proposal, the x86 processor instructions interleave with HMC instructions. This means the x86 processor continues to trigger all the instructions, but the execution and data access are up to the logic layer of the HMC, bypassing the cache hierarchy. However, we require modifications in the HMC to execute the HMC-Scan as the current ISA only supports compare-and-swap instruction (HMC_SWP) to evaluate values.

HMC-Scan Implementation We present the HMC compare instruction to operate over a memory address and an immediate. This instruction works similarly to the reciprocal x86

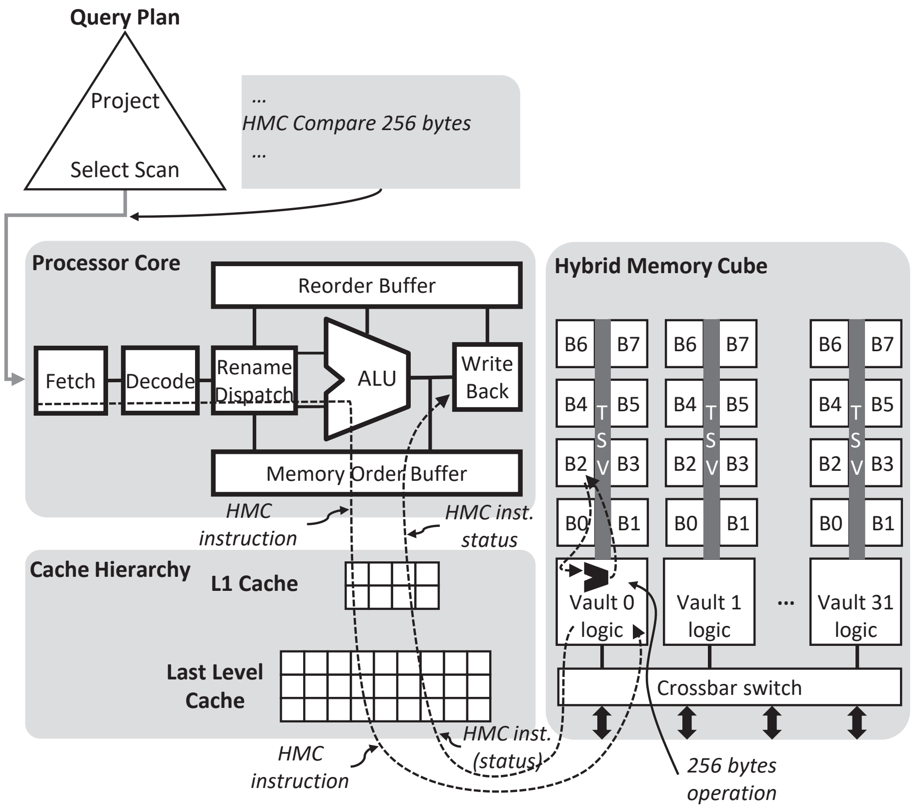


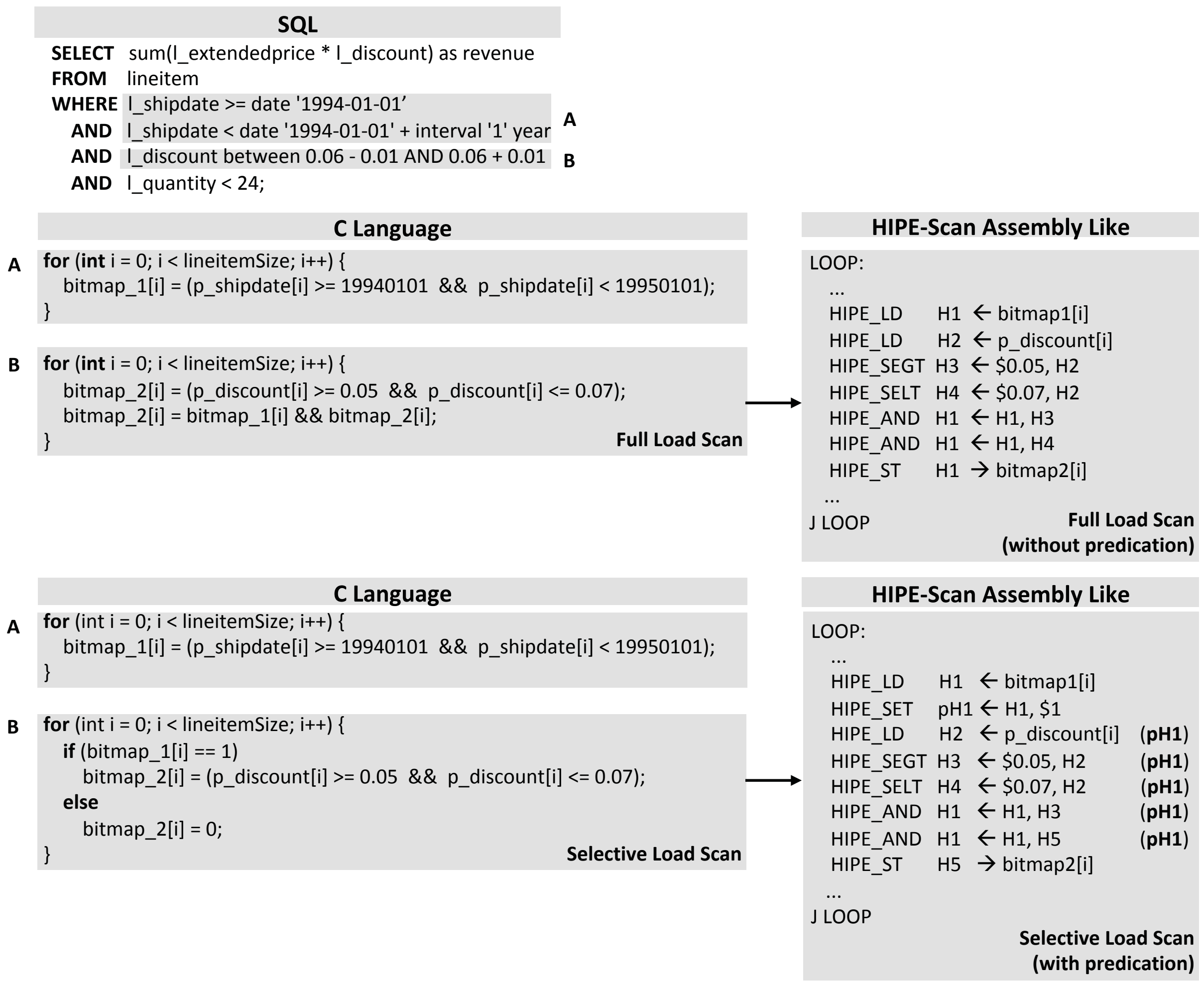
Figure 4.4: HMC-Scan bypassing the cache hierarchy to operate over 256 bytes of data in the logic layer.

compare instruction (CMP). With this new instruction, the x86 processor no longer compares the values in the registers to evaluate the query filters, but only receives the operation status bits and hands over to the logic layer of the HMC the evaluation of the query filters. Figure 4.5 presents two versions of the TPC-H Query 06 written in *C language* and simplified assembly. For now, we only refer to the “Selective Load Scan” *C language* version to understand the execution of the HMC-Scan. The assembly code is discussed in the next section.

The “selective load scan” consists of an “if-else-statement” that evaluates the matching bitmaps to perform or not load instructions. Therefore, the first scan evaluates the whole column and the resulting bitmap is used to avoid unnecessary loads. At the end of the processing, the final bitmap associates the column entries that received matches in all the query filters. For instance, initially the “Selective Load Scan” evaluates the entire *l_shipdate* column to build the bitmap of matches. This bitmap is used as a condition when scanning the *l_discount* column to avoid unwanted loads.

Hardware extension We refer again to the “selective load scan” code to understand our hardware extension. The execution of the HMC-Scan requires a new vector functional unit to operate over 256 bytes per vault instead of the current 16 bytes. This requires implementing 32 vaults × 64 scalar (i.e., 4 bytes) functional units.

With these functional units the logic layer requests 256 bytes of data from the DRAM bank to execute the compare instruction from the processor to scan the *l_shipdate* column. The vault controller responds the request and evaluates the query filter with 256 bytes of data at once. At the end of each compare instruction, the result is sent back to the x86 processor to be written in the respective bitmap to that column. The processor continues with the remaining



```
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   A
  AND   l_discount between 0.06 - 0.01 AND 0.06 + 0.01       B
  AND   l_quantity < 24;
```

```
C Language
A  for (int i = 0; i < lineitemSize; i++) {
     bitmap_1[i] = (p_shipdate[i] >= 19940101  &&  p_shipdate[i] < 19950101);
   }

B  for (int i = 0; i < lineitemSize; i++) {
     bitmap_2[i] = (p_discount[i] >= 0.05  &&  p_discount[i] <= 0.07);
     bitmap_2[i] = bitmap_1[i] && bitmap_2[i];
   }                                                         Full Load Scan
```

```
HIPE-Scan Assembly Like
LOOP:
  ...
  HIPE_LD     H1 ← bitmap1[i]
  HIPE_LD     H2 ← p_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 Scan
                        (without predication)
```

```
C Language
A  for (int i = 0; i < lineitemSize; i++) {
     bitmap_1[i] = (p_shipdate[i] >= 19940101  &&  p_shipdate[i] < 19950101);
   }

B  for (int i = 0; i < lineitemSize; i++) {
     if (bitmap_1[i] == 1)
       bitmap_2[i] = (p_discount[i] >= 0.05  &&  p_discount[i] <= 0.07);
     else
       bitmap_2[i] = 0;
   }                                                    Selective Load Scan
```

```
HIPE-Scan Assembly Like
LOOP:
  ...
  HIPE_LD     H1  ← bitmap1[i]
  HIPE_SET    pH1 ← H1, $1
  HIPE_LD     H2  ← p_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 Scan
                      (with predication)
```

Figure 4.5: The translation of the TPC-H Query 06 to C and simplified assembly version. Our two versions consider the HIPE-Scan instruction set performing *full load scan* or performing *selective load scan* using predicated instructions.

database operators moving on in the query plan evaluating the *l_discount* column. The following adaptations are required to implement the HMC-Scan:

- **Database:** We require no change in the source code of the DBMS to implement our proposal, but we need to recompile it to insert the HMC instructions, similarly to the SSE and AVX approaches.

- **Processor:** The processor needs an extension to its ISA to provide the execution of HMC instructions. The instructions pass the pipeline in the same way as a memory load operation. The HMC instructions 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 to HMC in [6] to extend the HMC instruction size to 256 bytes. Our goal is to take advantage of the full row buffer size that makes available 256 bytes of contiguous data for each bank access during each access, reducing thus the number of DRAM accesses.

4.3 HIPE: HMC Instruction Predication Extension

The HMC-Scan evaluates near-data query filters for in-memory databases. However, the HMC ISA is limited to perform update-instructions between a single memory address and an immediate. For instance, data resulted from a query filter must be returned to the processor in case it needs to be written to a second memory address. Although the comparison and store instructions are both executed inside the HMC, the interaction with the x86 processor creates data-flow dependency, which cannot be solved straight inside the HMC. Besides, the x86 processor is also required to solved control-flow dependency in order to decide over the execution of each instruction in the query plan, as discussed in Section 2. With such limitations in mind, in this section we describe our HMC extension to support data-flow inside the HMC and then replace control-flow by data-flow dependency to reduce the interaction with the x86 processor [51].

We refer again to Figure 4.5 to exemplify the interaction between the x86 processor and the HMC while running the "Full Load Scan" version of the TPC-H Query 06. When the database opts for this version of the query, the evaluation of each column does not take into account matching entries between columns and therefore does not worry about unwanted load operations. The evaluation of filters occurs independently in each column and creates branches in the execution of the code.

Let us consider the branched execution of a source code depicted by Figure 4.6. 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 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. Previous work could only solve such problem by inserting a full processor inside the memory, which has a huge area overhead.

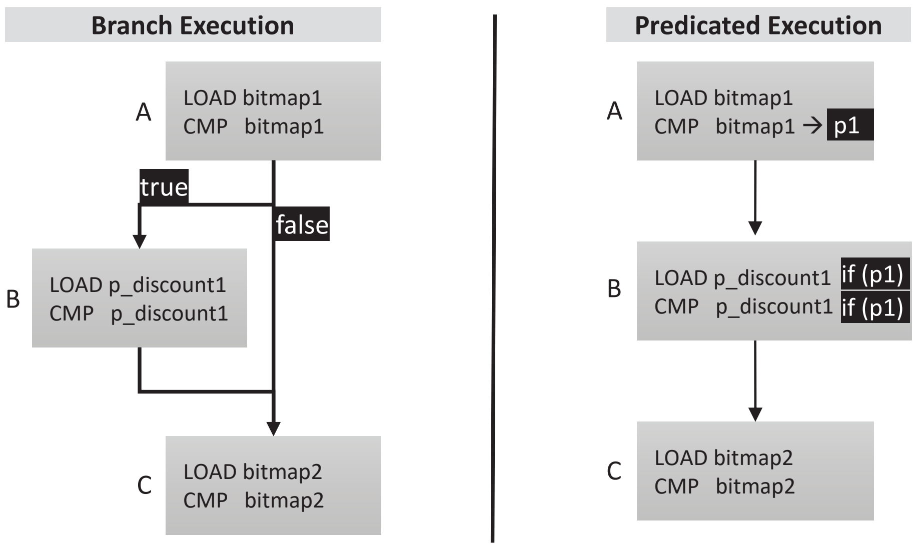


Figure 4.6: Branched vs. predicated execution of query filters.

Instead, we propose to use predicated execution of the select scan code, called HIPE-Scan. Using predicated instructions we can merge multiple basic-blocks into a single super-block. It means that, the branch is removed and the instructions annotated. The annotations mean that the code may only be executed considering a certain condition. Figure 4.6 also shows the predicated execution of a code. This sequence of instructions presents only data-flow dependencies and no longer require the x86 processor to solve any control-flow. During the execution, in the case the predicate is false, the predicated instructions will be squashed by the logic layer, i.e. these instructions are converted to *NOP* operations and simply discarded.

Hardware extension Figure 4.7 depicts the 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. The load/store instructions perform data transfers between the DRAM and the register bank. The ALU operations perform computations inside the ALU. The load/store and ALU instructions evaluate predicates. This means they execute if some register matches the wanted value.

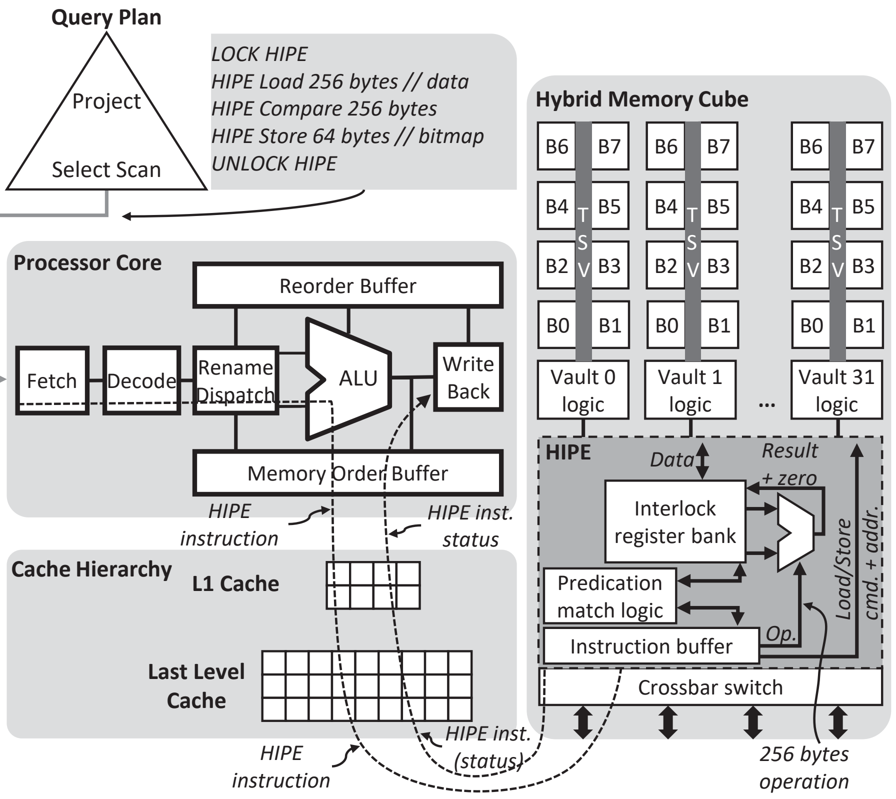


Figure 4.7: The architecture of HIPE.

The register bank stores not only the result value, but also the zero flag from each operation used during the predicated execution. The register bank implements a 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.

The predication match logic is responsible for checking before the instruction execution if the predicate is true or false. 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 in position to discuss the assembly code of both query versions and replace control-flow to data-flow dependency in the HMC. In the assembly code of the “Full Load Scan” 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 Scan" version presents the predicated execution, where no instructions annotated with $pH1$ are executed if the query filter evaluates false.

Similarly to the HMC-Scan support, the following modifications are required to implement the HIPE-Scan:

- **Database:** we require no changes in the source code of the database system to implement the HIPE-Scan, 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.
- **HMC:** We based our extension to execute predicated execution inside the HMC on the state-of-the-art of HMC processing, called HIVE [6]. Our goal is to take advantage of the data-flow support already provided by this previous work and extend it to support predicated instructions.

Chapter 5

Experimental Evaluation

In this chapter we present the simulation environment, the experimental methodology and the results with the implementation of the *select scan* on top of the tuple/column/vector-wise query engines implemented as HMC-Scan and HIPE-Scan.

5.1 Methodology and Setup

We used the SiNUCA cycle-accurate in-house simulator [7] 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 *select scan*, considering an aggressive out-of-order processor, advanced multi-banked and non-blocking caches together with the HMC. Table 5.1 shows the major system parameters used in our study.

In order to generate the simulation inputs we have implemented the tuple/column/vector-wise query engines running the query 06 from TPC-H benchmark. First, we analyzed the generated assembly code from the compilation to build the basic blocks that belongs to the select scan operation. Next, we identified the execution flow and the order of each basic block execution. Finally, we traced the memory footprint during the execution to identify load and store operations.

The baseline architecture was inspired by the Intel Sandy-Bridge processor micro-architecture referred to as x86. The Sandy-Bridge was configured with AVX-128 instruction set capabilities, and in all cases, the main memory used was the HMC version 2.1 [29]. For this baseline (x86), all the instructions are executed in the x86 processor.

The approach called HMC-Scan 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 case we are executing the *compare* instructions inside the HMC interleaving it with x86 instructions.

We call HIPE-Scan the approach using the extended HMC ISA that supports data-flow inside the HMC, and also supports predicated instructions also inside the HMC

5.2 Evaluation Plan

We focused our experiments on TPC-H, a decision support database benchmark widely adopted to assess the performance of DBMSs by representing a read-mostly workload. TPC-H consists of a suite of twenty-two business oriented ad-hoc queries that have broad industry-wise relevance and examine large volumes of data that can be configured in three dataset size: 1GB,

Table 5.1: 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;

10GB and 100GB. In our experiments, we generated a database of 1GB and we stick to a micro-benchmark running the TPC-H Query 06. This query implements complex boolean expressions, what results in a possibility of push-down the most selective predicates. It also consists of conjunctions to the *lineitem* table without join operations. We let join operations for future work as it requires understanding the impact of each one of the many different join algorithms on the HMC.

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 bitmap with “1” for match and “0” for no match to be used ahead by the further predicates. In the vectorized execution, we generated the bitmap for each vector, such as the proposed design principles for SIMD vectorization of main-memory database operators [43]. We considered that each tuple from the *lineitem* table occupies 64-bytes, which is equal to the cache line size and this assumption is beneficial for x86. During our experiments, we run the tuple-at-a-time engine considering a row-store storage model and the other two query engines (column-at-a-time and vectorized execution) considering the column-store storage model.

In order to perform our experiments, we divided the *select scan* implementation for multiple predicates into the two variations, referred as: *selective load scan* and *full load scan*.

Table 5.2 describes the evaluation plan and the scan implementation used for each architecture. In the first and second experiments, we evaluated the best operand size and loop

Table 5.2: Experiments evaluation plan

Evaluation/Architecture	x86	HMC-Scan	HIPE-Scan
Operand Size	Selective	Selective	Full
Loop Unroll Depth	Selective	Selective	Full
Selectivity	Selective	Selective	Full/Selective

unrolling depth for each architecture. The x86 and HMC-Scan architectures implement the *selective load scan*, while the HIPE-Scan implements the *full load scan*.

When we performed the third experiment analyzing the impact of selectivity when filtering different amounts of data, we evaluated HIPE-Scan with both the *selective load scan* and *full load scan*. When performing the *selective load scan*, the HIPE-Scan make use of the predicated execution architecture provided by the HMC ISA extension proposed in this work.

5.3 Evaluation Results

5.3.1 Impact of Operand Sizes

We evaluated the TPC-H Query 06 using HMC-Scan and HIPE-Scan using five different data operand sizes from 16 to 256 bytes (i.e. limited by the row buffer size), while we set the x86 to work with up to 64 bytes (i.e. the biggest x86 supported instruction size AVX-512).

Tuple-at-a-time: Figure 5.1 shows that the time to process the select scan increased in 97% compared to x86 when performing 16 bytes width operations in the HMC-Scan. We observed the same result when performing with 32 and 64 bytes width operations. They increased the response time by 1.02× and 1.19× respectively compared to x86. As expected, the increase of the response time was due to the amount of DRAM access to open and close the row-buffer during each access (i.e., due to the closed-page policy) with a 64 bytes cache line to fetch by each x86 load instruction. The best scenario happens when the operand size is set to 256 bytes. The execution time dropped in 18% compared to the x86 baseline with the best time, because the select scan is allowed to process 4 contiguous tuples per operation without suffering from the latency of the cache hierarchy.

The execution with HIPE-Scan processing 16 bytes at a time resulted in an increase of 3× in the execution time when compared to x86 baseline. When extended to 256 bytes, the execution time was only 11% bigger than x86. The increasing in execution time occurs due to the control-dependency of each isolated lock/unlock block when loading large amounts of data with HIPE-Scan.

Column-at-a-time: Figure 5.2 presents that the execution time using the HMC-Scan operating over 16 bytes of data increased in 10% compared to the x86 due to effect of the closed page policy. To mitigate this effect and also reduce the amount of DRAM accesses, we increased the amount of data processed by each instruction to the entire row-buffer size of 256 bytes, reducing the execution time by 4.38×.

The execution with HIPE-Scan over 16 bytes per instruction increased the execution time in 9.45× compared to x86 (AVX-128) execution. When the instruction is extended to operate over 256 bytes, the execution time dropped in 10% against the AVX-128, but it increases 2× compared with the best case of x86 execution (AVX-512). Notice that after processing the first

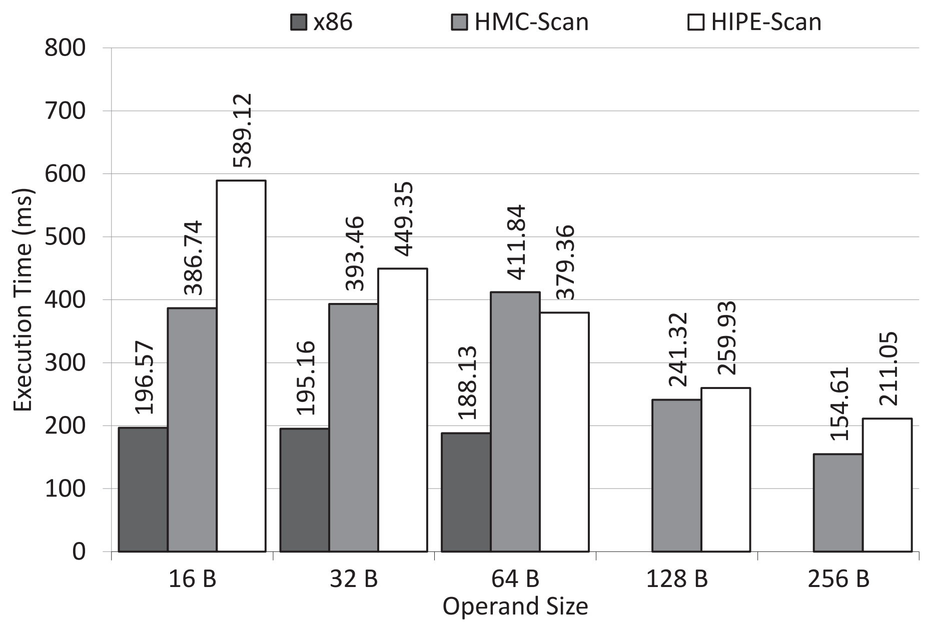


Figure 5.1: Tuple-at-a-time execution varying operation size.

column, the processor needs to fetch the previous generated bitmap to decide the portions of the second column it needs to process. This generates data dependency and delays the execution of HIPE-Scan instructions as more DRAM accesses need to be performed, in contrast to cache access for x86 and HMC-Scan.

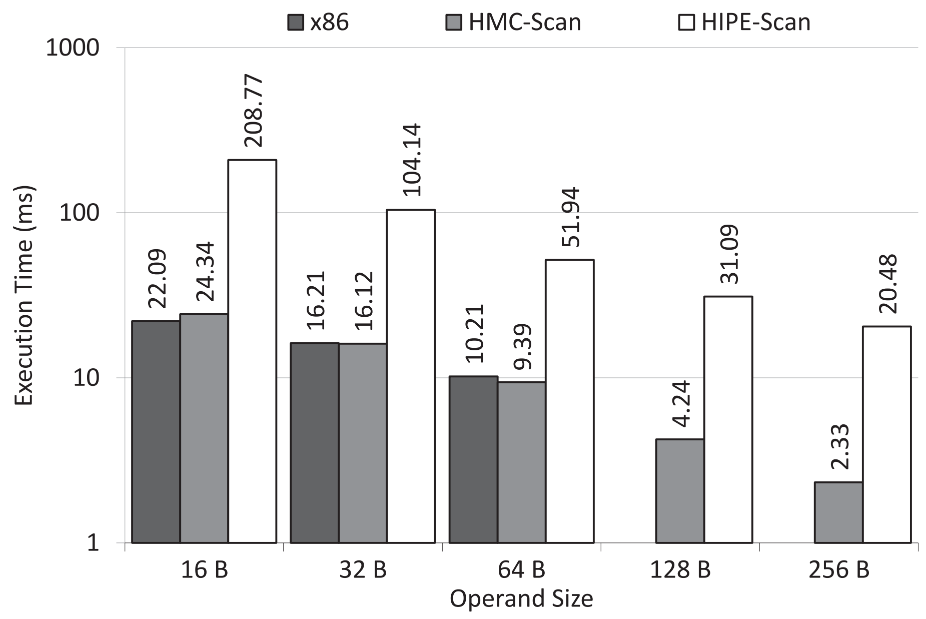


Figure 5.2: Column-at-a-time execution varying operation size.

5.3.2 Impact of Unrolling Depths

In this section, we choose the best case observed in the previous experiment with 64 bytes for x86 and 256 bytes for HMC-Scan and HIPE-Scan operand size using the column-store storage model to analyze the impact of loop unrolling technique. In this case, we evaluated two query execution strategies, column-at-a-time and vectorized execution.

We evaluated five different unrolling depths, ranging from 1x to 32x, while we set the x86 to up to 8x (i.e. the deepest unroll implemented by compilers due to the reduced number of general purpose registers).

Column-at-a-time: Figure 5.3 shows that the HMC-Scan reduces the execution time by 5.05× with the 32× depth when compared to x86 (AVX-512). As expected the unrolling loop technique allows more parallelism, although the x86 execution takes advantage until the depth of 4x. A slight increase happens in the x86 depth of 8×, because the amount of code in the loop body results in a higher miss ratio in the instruction cache. Besides, another factor limitation that occurs in aggressive unrolling loop is the shortfall in registers. The pressure made by the amount of instructions forces that several registers be allocated. Thus, the code may lose the performance advantages creating shortfall in registers.

The evaluation with HIPE-Scan presented in Figure 5.4 shows the performance improvement provided by the HMC parallelism. When we performed the unroll depth of 32× the execution time reduces in 7.6× compared with x86 (AVX-512) without unrolling, and 5.4× compared with x86 (AVX-512) with depth of 4×. The HIPE-Scan implementation allows the processor to send 32 independent instructions per loop iteration, which implicates in a full parallelism usage between the 32 HMC vaults. Besides, the *full load scan* algorithm reduces the necessity of control from the processor by sending all the load instructions sequentially. By increasing the amount of code in the loop body the HMC doesn't suffer like the CPU, since the CPU has a limitation fo loop unrolling due to the increasing in the cache instruction miss rate.

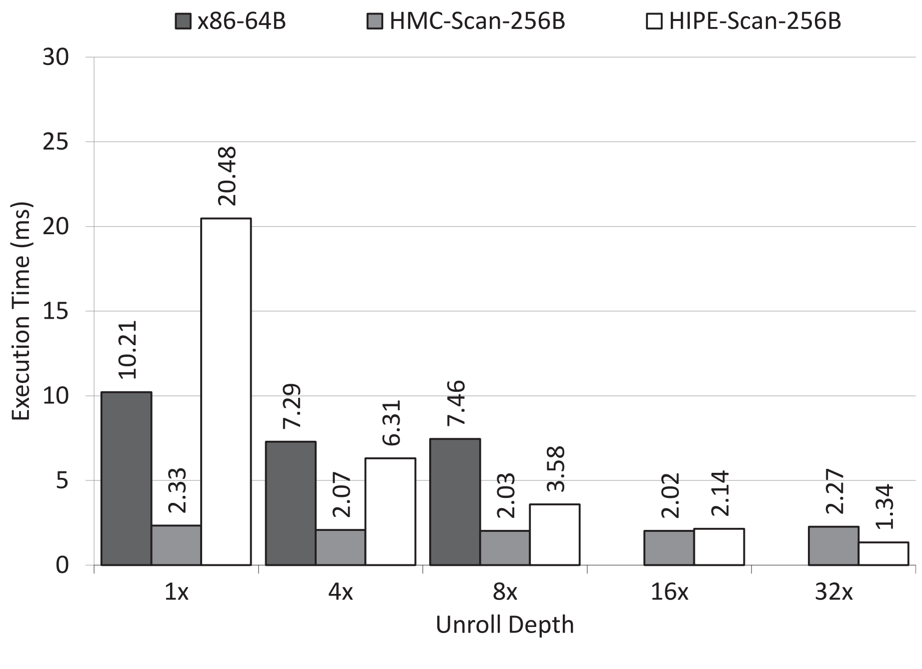


Figure 5.3: Column-at-a-time execution varying loop unrolling depth for x86, HMC-Scan and HIPE-Scan.

Vectorized execution: Figure 5.4 presents the results for the vectorized execution. In this query execution strategy, the processing occurs in a granularity of vectors by each column.

The HMC-Scan presents the best result with the depth of 8× reducing in 3.36× the execution time compared to x86 (AVX-512) in the best case with the depth of 4×. When

HMC-Scan executes the depth of 16× and 32× the execution time starts get worst due to the increasing number of x86 instructions to perform the bitmap evaluation between predicates.

The evaluation with HIPE-Scan demonstrates the high potential of the HMC bandwidth. By comparing to x86 (AVX-512), the execution time dropped in 7.66× in the best case (32× depth), performing 32 independent instructions at time inside the HIPE-Scan. When the loop is unrolled, HIPE-Scan overlaps DRAM latency with consistent parallel requests, due to the interlock register bank. With this depth, each loop iteration is capable of sending requests over 4 KBytes (4 vectors of 1024 KBytes) of data to be processed inside the HMC.

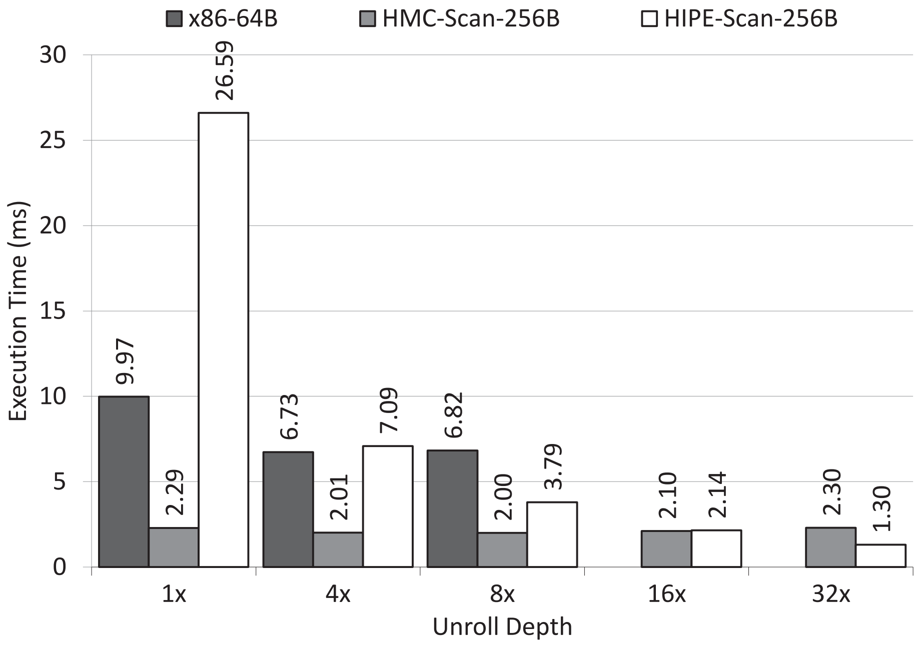


Figure 5.4: Vector-at-a-time execution varying loop unrolling depth for x86, HMC-Scan and HIPE-Scan.

5.3.3 Impact of Selectivity

In this section, we choose the best loop unrolling depth for each architecture to observe the impact of evaluating different amounts of data when varying the selectivity. Figure 5.5 describes the execution time for the column-at-a-time engine varying the selectivity of the *shipdate* attribute.

The selectivity factor has important impact in the x86 architecture, increasing the execution time by almost 90% comparing the 0.001 and 100 selectivity factors. When compared to the HMC-Scan the execution time difference between x86 and HMC-Scan varies from 2.98× in the selectivity factor of 0.001 to up to 2.88× in the selectivity factor of 100.

For the HIPE-Scan performing the *full load scan* algorithm, the selectivity factor has no impact in execution time, since in this case both wanted and unwanted data are requested and checked independent of the selectivity. The cache bypassing also brings benefits since it is not necessary to check the cache in order to perform the predicates. When compared to x86 architecture it shows an improvement of 3.00× for 0.001 selectivity factor and 5.64× for the 100 factor.

The evaluation of the HIPE-Scan performing the *selective load scan* algorithm achieves a better result when operates with selectivity smaller than 1%. HIPE-Scan using predication

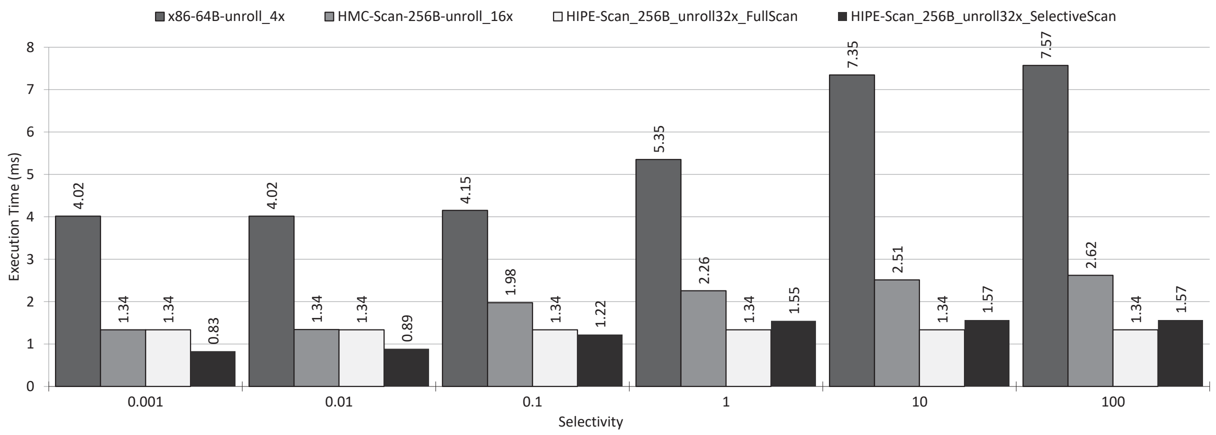


Figure 5.5: Evaluating execution time of TPC-H Q6 varying the selectivity factor in the different hardware architectures in the column-at-a-time engine.

could improve by 4.84× for the 0.001 selectivity factor, and 4.82× for 100 factor. By using the predicated execution, the algorithm on HIPE-Scan can avoid many DRAM accesses considering the previous bitmap evaluation and for the 0.001 factor it could translate in high performance improvements.

Thus, we can notice a clear trade-off between the *full load scan* and the *selective load scan* considering different selective factors. Such trade-off shows us that the database scheduler needs adaptations to maximize the gains from the architecture.

5.3.4 Understanding the Impact of Predication

In this section, we present the execution flow diagram to explain the trade-off between HIPE-Scan with and without predication. In this example, we consider that every instruction inside HIPE takes 1 cycle and load/store latency varies between 80 cycle when there is few operations in parallel (low contention) and 100 cycles when multiple operations are happening in parallel (high contention). It is important to notice that the simulator considers different latency and more component details, in such manner that the simulation results differ from the results in this simplified example.

Figure 5.6 describes the HIPE-Scan instructions when executing the *full load scan* on the *l_discount* column with the bitmap generated by the scan on the *l_shipdate* column. This example illustrates a select scan unrolled 4×. After a lock instruction during each clock cycle, a 256 bytes-wide load is requested, then the bitmap of the *l_shipdate* column scan is loaded. Considering the DRAM access latency with high contention, the query 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* 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. Besides, since the previous bitmap result is used only after the instructions of load and compare operations, a full scan will be performed in each column.

Figure 5.7 and Figure 5.8 illustrate what happens when using the *selective load scan* supported by the predicated execution for high and low selectivity, respectively. Figure 5.7 presents the worst-case scenario, where we observe an increase of 36% in the execution time compared to the *full load scan*. We observe an extra DRAM read latency included in the critical

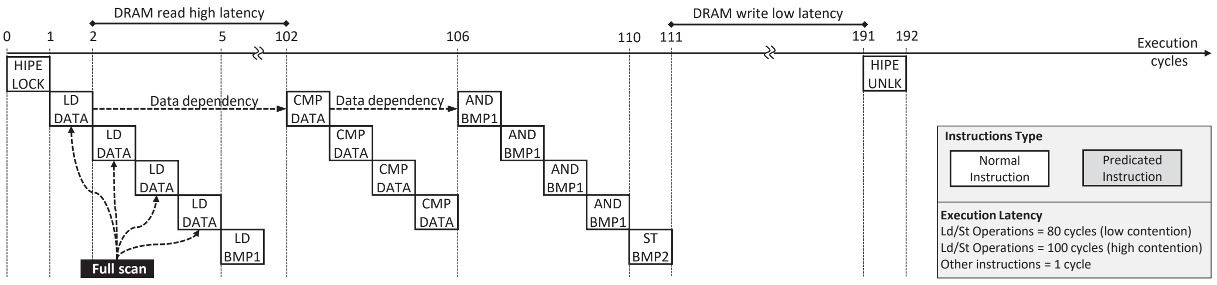


Figure 5.6: HIPE *full load scan* execution (no predication/loop optimization).

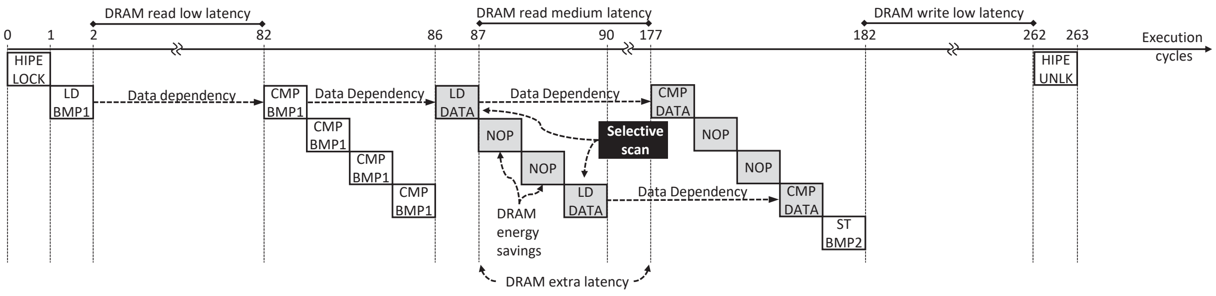


Figure 5.7: HIPE *selective load scan* (predication/loop optimization) worst/average case.

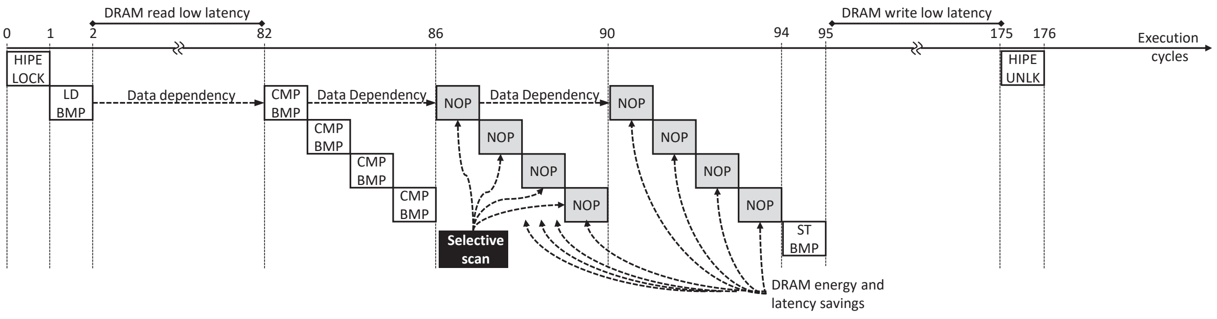


Figure 5.8: HIPE *selective load scan* (predication/loop optimization) best case.

path, because the *selective load scan* first loads and compare the bitmap from the *l_shipdate* column before issuing the load for the *l_discount* column. Figure 5.8) presents the best-case scenario. We observe 9% of reduction in the execution time compared to the *full load scan*. This scenario turns out to be a common case, where no load is issued, due to the low number of matches in the *l_shipdate* column. Preliminary results show that such the reduction in the number of loads also reduces 5% in energy consumption of the DRAM.

In summary, the *full load scan* increases the DRAM contention and also wastes energy loading non-required data in the low selectivity scenario. On the other hand, the *full load scan* performs better due to the lack of extra data-dependency for the high selective scenario.

Chapter 6

Conclusions and Future Work

In this dissertation we presented a new near-data approach to process select scan database operations inside the Hybrid Memory Cube (HMC) to address the performance gap between memory and processor. To this end, we presented two extensions to the HMC Instruction Set Architecture (ISA) called HMC-Scan and HIPE-Scan.

The research questions that motivated our work, were:

1. What happens when database systems run the select scan over the current x86 architecture using the HMC as ordinary DRAM?
2. Can we use the current HMC Instruction Set Architecture (ISA) to implement the near-data select scan?
3. What are the extensions to the HMC ISA to leverage the full potential of the hardware and reduce the interleaving with x86 instructions?

Motivated by the first question, we implemented the main query execution engines to execute the Query 06 from the TPC-H benchmark. We evaluated these engines against the execution with the current HMC ISA. When using the HMC as ordinary DRAM the processor presented important overhead in data-movement. Even worse, the data provided by the HMC in each access remained underused.

Next, to answer the second question we used the current HMC ISA to execute the select scan with no noticeable performance improvements. The result motivated the hardware extensions proposed in this dissertation.

To answer the third question, we formulated some architecture extensions and we evaluated them against the main query execution engines in a x86 (AVX-128 and AVX-512) architecture. The execution of the select scan inside the HMC outperformed the tuple-at-a-time in 30%, the column-at-a-time in 5.64× and the vectorized execution in 5.17× when compared to baseline that used x86 execution with HMC as ordinary DRAM.

We conclude that our results support the development of future in-memory DBMS to benefit from near-data processing architectures, like the HMC. However, much work needs to be done to explore all the DRAM parallelism present on the memory devices, including understanding the impact of the many algorithms implemented by the relational operators. We observed that choosing the correct algorithm on different selectivity factors may have great impact on the performance, which may occur to other metrics inside the database.

6.1 Future Work

An important next step includes understanding other database operations inside the HMC such as: joins, projections and aggregations. When the evaluation and adaptations finish it could be possible allocate each operation from a query plan to be executed in the best hardware to achieve good performance while also saving energy.

To sum up, we set the following ideas for future work:

1. Develop a many-core scheduler to analyze the query plan and choose the best hardware to each query operator. This requires a cost model to decide between x86 and HMC instructions during run-time, since not always the near-data processing is the right choice for DBMSs. Besides, a general cost model could also help to adapt Database Systems for other co-processor such as GPUs, FPGAs and SSDs.

2. A deep energy consumption analysis to provide the possible gains in reducing DRAM access and data-movement with the extensions proposed in this dissertation.

6.2 Published Papers

The resulting publications of our work are presented below:

- Diego G. Tomé, Marco A. Z. Alves, and Eduardo C. Almeida. Uma abordagem para processamento em memória de operacões de seleção em sistemas de bancos de dados. In Simpósio Brasileiro de banco de Dados (SBBD), 2017.

- Diego G. Tomé, Paulo C. Santos, Luigi Carro, and et al. HIPE: HMC instruction predication extension applied on database processing. In Desing Automation and Test in Europe (DATE), pages 710–715, 2018.

The following papers were produced as part of the development of this dissertation:

- Paulo C. Santos, Geraldo F. Oliveira, Diego G. Tome, and et al. Operand size reconfigurationfor big data processing in memory. In Desing Automation and Test in Europe (DATE), pages 710–715, 2017.

- Aline S. Cordeiro e Tiago R. Kepe e Diego G. Tomé e Eduardo C. de Almeida e Marco A. Z. Alves. Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions. XVIII Simpósio em Sistemas Computacionais de Alto Desempenho-WSCAD, 2017.

Bibliography

[1] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., 2013.

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, 2007.

[3] A. Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB*, pages 198–215, 2002.

[4] Anastassia Ailamaki, David J. DeWitt, and Mark D. et al. Hill. Dbmss on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[5] FE Allen and J Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30, 1972.

[6] Marco A. Z. Alves, Matthias Diener, Paulo C. Santos, and Luigi Carro. Large vector extensions inside the hmc. In *DATE*, pages 1249–1254, 2016.

[7] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, and t al. Sinuca: A validated micro-architecture simulator. *HPCC*, pages 605–610, 2015.

[8] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *SIGMOD*, pages 1753–1758, 2017.

[9] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, and et al. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro*, pages 36–42, 2014.

[10] L. Benini. A network on chip architecture and design methodology. In *VLSI*, pages 117–, 2002.

[11] Eric Beyne, Piet De Moor, Wouter Ruythooren, and et al. Through-silicon via and die stacking technologies for microsystems-integration. *IEDM*, 2008.

[12] Peter A. Boncz and Martin L. Kersten. Mil primitives for querying a fragmented world. *VLDB*, pages 101–119, 1999.

[13] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, pages 77–85, 2008.

[14] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[15] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[16] David Broneske, Sebastian Breß, and Gunter Saake. Database scan variants on modern cpus: A performance study. In *IMDM@VLDB*, 2014.

[17] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[18] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, and et al. Query processing on smart ssds: Opportunities and challenges. In *SIGMOD*, pages 1221–1230, 2013.

[19] Simone Dominico, Eduardo Cunha de Almeida, and Jorge Augusto Meira. A petrinet mechanism for OLAP in NUMA. In *DaMoN*, pages 7:1–7:4, 2017.

[20] Yasuko Eckert, Nuwan Jayasena, and Gabriel H. Loh. Thermal feasibility of die-stacked processing in memory. 2014.

[21] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, pages 31–46, 2015.

[22] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA*, pages 140–150, 1983.

[23] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 1995.

[24] G. Graefe. Volcano an extensible and parallel query evaluation system. *TKDE*, pages 120–135, 1994.

[25] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, and et al. Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube. *CoRR*, 2017.

[26] Mary Hall, Peter Kogge, Jeff Koller, and et al. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *SC99*, 1999.

[27] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, pages 267–276, 1993.

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

[29] HMC-Consortium. Hmc specification 2.1. `http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf`. Accessed: 2017-05-31.

[30] Pedro Holanda and Eduardo Cunha de Almeida. Spst-index: A self-pruning splay tree index for caching database cracking. In *EDBT*, pages 458–461, 2017.

[31] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, and et al. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *ISCA*, 2016.

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

[33] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proc. VLDB Endow.*, pages 642–653, 2015.

[34] W. H. Kautz. Cellular logic-in-memory arrays. *IEEE Transaction on Computers*, pages 719–727, 1969.

[35] Soroosh Khoram, Yue Zha, Jialiang Zhang, and Jing Li. Challenges and opportunities: From near-memory computing to in-memory computing. In *ISPD*, pages 43–46, 2017.

[36] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *MICRO*, 2005.

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

[38] Yinan Li, Ippokratis Pandis, René Müller, and et al. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[39] Lukas M. Maas, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Buzzard: A numa-aware in-memory indexing system. In *IGMOD*, pages 1285–1286, 2013.

[40] Nooshin S. Mirzadeh, Onur Kocberber, Babak Falsafi, and et. al. Sort vs. hash join revisited for near-memory execution. In *ASBD*, 2015.

[41] Onur Mutlu. Memory scaling: A systems architecture perspective. In *IMW*, pages 21–25, 2013.

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

[43] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[44] J. L. Potter and W. C. Meilander. Array processor supercomputers. *Proceedings of the IEEE*, pages 1896–1914, 1989.

[45] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and et al. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *VLDB*, pages 37–48, 2016.

[46] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, and al. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. *ISPASS*, 2014.

[47] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *FAST*, pages 1–16, 2014.

[48] Paulo C. Santos, Geraldo F. Oliveira, Diego G. Tome, and et al. Operand size reconfiguration for big data processing in memory. In *DATE*, pages 710–715, 2017.

[49] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th International Conference on Supercomputing*. ACM, 2000.

[50] Diego G. Tomé, Marco A. Z. Alves, and Eduardo C. Almeida. Uma abordagem para processamento em memória de operacões de seleção em sistemas de bancos de dados. In *SBBD*, 2017.

[51] Diego G. Tomé, Paulo C. Santos, Luigi Carro, and et al. Hipe: Hmc instruction predication extension applied on database processing. In *DATE*, pages 710–715, 2018.

[52] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[53] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, pages 385–394, 2009.

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

[55] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *DAMON*, pages 2:1–2:10, 2015.

[56] Hao Zhang, Bogdan Marius Tudor, Gang Chen, and Beng Chin Ooi. Efficient in-memory data management: An analysis. *Proc. VLDB Endow.*, pages 833–836, 2014.

[57] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *SIGMOD*, pages 1567–1581, 2016.

[58] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, 2002.

[59] Yuxiong Zhu, Borui Wang, Dong Li, and Jishen Zhao. Integrated thermal analysis for processing in die-stacking memory. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, 2016.