RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt

Cagri Balkesen
cagri.balkesen@oracle.com
Oracle Labs

Nitin Kunal
nitin.kunal@oracle.com
Oracle Labs

Georgios Giannikis
georgios.giannikis@oracle.com
Oracle Labs

Pit Fender
pit.fender@oracle.com
Oracle Labs

Seema Sundara
seema.sundara@oracle.com
Oracle Labs

Felix Schmidt
felix.schmidt@oracle.com
Oracle Labs

Jarod Wen
jarod.wen@oracle.com
Oracle Labs

Sandeep Agrawal
sandeep.r.agrawal@oracle.com
Oracle Labs

Arun Raghavan
arun.r.raghavan@oracle.com
Oracle Labs

Venkatanathan Varadarajan
venkatanathan.varadarajan@oracle.com
Oracle Labs

Anand Viswanathan
anand.viswanathan@oracle.com
Oracle Labs

Balakrishnan Chandrasekaran
balakrishnan.chandrasekaran@oracle.com
Oracle Labs

Sam Idicula
sam.idicula@oracle.com
Oracle Labs

Nipun Agarwal
nipun.agarwal@oracle.com
Oracle Labs

Eric Sedlar
eric.sedlar@oracle.com
Oracle Labs

ABSTRACT

Today, an ever increasing amount of transistors are packed into processor designs with extra features to support a broad range of applications. As a consequence, processors are becoming more and more complex and power hungry. At the same time, they only sustain an average performance for a wide variety of applications while not providing the best performance for specific applications. In this paper, we demonstrate through a carefully designed modern data processing system called RAPID and a simple, low-power processor specially tailored for data processing that at least an order of magnitude performance/power improvement in SQL processing can be achieved over a modern system running on today's complex processors. RAPID is designed from the ground up with hardware/software co-design in mind to provide architecture-conscious extreme performance while consuming less power in comparison to the modern database systems. The paper presents in detail the design and implementation of RAPID, a relational, columnar, in-memory query processing engine supporting analytical query workloads.

CCS CONCEPTS

• **Information systems** → **Online analytical processing engines**; • **Computer systems organization** → *Multicore architectures*;

KEYWORDS

Databases; In-Memory Data Processing; Hardware/Software Co-Design; DPU; Low-Power; Analytic Query Processing

ACM Reference Format:
Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. 2018. RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3190655

1 INTRODUCTION

We are in an era where the data volumes are increasing tremendously. The "big data age" is dictating an evolution in both hardware and software for data processing systems. Requirements of day-to-day analytical and business intelligence workloads of modern enterprises stress the limits of today's systems in terms of performance and power consumption. In order to address the trends, modern data processing systems need a rethinking in software design and the hardware support underneath. A modern data processing system cannot be designed by isolating software from hardware.

Looking at the hardware landscape, we see an increasing number of transistors being utilized in more processing cores, large caches, new instruction sets and advanced features. The main-memory is also becoming abundant at a reduced cost. As a result, existing data processing systems can provide higher performance by exploiting the recent advances in hardware. Following a hardware-conscious design, systems must be extended with higher parallelism, cache and interconnect conscious data placement and advanced vector-based processing to achieve higher absolute performance.

The hardware-conscious design on commodity hardware might address the necessity for absolute performance on specific cases.

However, today's commodity hardware is designed to run a wide range of applications, from games to scientific applications, with the goal of providing the best performance for the average case. Consequently, the hardware comes with increased power consumption and complexity while not providing the best performance for specific applications. This only adds up to the challenge of meeting performance/power requirements of modern cloud data centers. Modern cloud data center architectures require the systems to be power-conscious in addition to providing high performance. Therefore, the architectural rethinking of modern systems have to consider hardware/software co-design from the perspective of power consumption and heat dissipation as well.

Historically, data processing system design is driven by the commodity hardware. The goal has been to take advantage of new hardware features by tweaking the design of software accordingly. However, in this paper, we take an opposite approach and transpose the question: How should the hardware be tailored for data processing? What are the features needed for efficient relational data processing? Likewise, what are the unnecessary features that we can abandon for reducing power consumption and keeping the processor design simpler? By answering these questions, we converge to a hardware design that enables tightly integrating software and hardware for data processing. As we design the architecture from the ground up with hardware/software co-design in mind, we are able to provide an extreme query execution performance and achieve at least an order of magnitude higher performance per watt for analytical query processing.

RAPID query processing engine is co-designed with a new processor called the Data Processing Unit (**DPU**). The DPU consists of 32 low-power processing cores called the *dpCores*. A *dpCore* is a simple, in-order CPU with simple branch prediction, 16 KiB of SRAM cache and 32 KiB of SRAM scratchpad memory. Each *dpCore* executes simple MIPS-like instructions with single cycle latency tailored for database processing. The DPU is not cache-coherent and transfers to/from DRAM are explicitly programmed with a data movement system called the **DMS**. The DMS is at the core of data shuffling and partitioning without the involvement of *dpCores*. Essentially, the DPU abandons many complex components such as large caches, sophisticated branch predictors, advanced instructions and cache-coherency to process data at a lower power consumption and a small form factor.

In this paper, we present RAPID, a relational query processing engine designed to support modern analytical workloads with an emphasis on **architecture-conscious performance** at **low power** consumption compared to existing database systems. RAPID provides a novel design from scratch with hardware aware data/storage model, query optimizations and data processing operators. RAPID is pluggable and can attach to an operational relational database for offloading analytical queries.

The paper explains the integration of RAPID to a commercial, widely used relational database management system. We show that RAPID performs quite well with modern workloads while consuming a fraction of the power that alternate high-end systems need. RAPID provides in average **15X** more performance per watt compared to a high-end system. Despite being tailored for the DPU, we demonstrate that RAPID software design also leads to better performance on commodity x86 when compared to a high-end, in-memory database system due to the novel architecture and algorithms exploiting modern hardware efficiently.

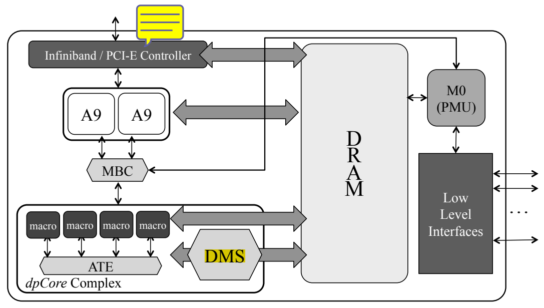

Figure 1: The block diagram of a RAPID DPU.

In the interest of this paper, we focus on the design of RAPID processing nodes that achieves the high performance per watt in analytical query execution and leave the scale-out and networking aspects as the topic of a follow-up publication.

2 RAPID DPU ARCHITECTURE

In this section, we provide a brief overview of the RAPID Data Processing Unit (**DPU**) architecture. We refer the reader to [1] by Agrawal et al. for a more detailed presentation.

RAPID DPU is a low-power, programmable processing unit especially designed for accelerating data processing at a provisioned power of 5.8 W at 40 nm fabrication process. Each processing core consumes 51 mW of dynamic power at 800 MHz, highlighting our focus on low-power design. The primary design goal of the DPU is to accelerate analytic workloads [15, 29, 32] by virtue of the hardware design.

2.1 Data Processing Cores (*dpCores*)

The DPU contains 32 data processing cores (*dpCore*) with a simple, low-power design that cooperatively exploit any data parallelism. The *dpCores* are organized into 4 macros each with 8 cores. The *dpCore* features a 64-bit MIPS-like instruction set architecture (ISA) for general purpose compute. To accelerate analytic query operators (*e.g.*, selection, join), the ISA provides single-cycle instructions such as bit-vector load (BVLD), filter (FILT) and hash value generation (CRC32).

The *dpCore* implements a simple dual-issue pipeline, one for the arithmetic logic unit (ALU) and the other for the load-store unit (LSU). The ALU supports a low-power multiplier that stalls the pipeline for multiple cycles, but has no native support for floating point arithmetic. The *dpCore* uses a simple branch predictor that predicts backward branches as taken. The memory model is relaxed, with instructions to fence-off pending loads and stores. The *dpCore* has no memory management unit and the memory is directly addressed.

2.2 Data Memory (DMEM) and Caches

Each *dpCore* contains a 32 KiB SRAM-based scratchpad memory called data memory, or DMEM in short. The DMEM is mainly utilized as a single-cycle latency, fast, temporary storage for *dpCores*. It is explicitly managed by the software.

The *dpCore* also supports a general-purpose cache hierarchy with core-private 16 KiB L1-D and 8 KiB L1-I caches and a 256 KiB L2 cache shared between *dpCores* in a macro. To reduce chip complexity and power, hardware does not manage coherency between caches. Instead, the ISA provides cache flush/invalidate instructions for software-managed coherence.

2.3 Data Movement System (*DMS*)

RAPID DPU has an on-chip programmable data movement engine, called Data Movement System (**DMS**), to move data between DRAM and the *dpCores*. The majority data accesses go through the DMEM using the DMS. Compared to the traditional approach of utilizing CPU caches and hardware prefetchers, DMS is energy efficient and can deal with irregular data access patterns [15]. DMS supports complex access patterns involving horizontal hash/range partitioning and scatter/gather while transferring the data. DMS is capable of directly placing data into the DMEM to be immediately available for the *dpCores* without any overhead.

2.4 On-Chip Communication

A hardware block called the Atomic Transaction Engine (**ATE**) allows communication among the *dpCores*. The ATE block comprises of a 2-level crossbar connecting 8 *dpCores* and 4 macros and hardware to manage messaging with guaranteed point-to-point ordering over this interconnect. On each *dpCore*, the hardware ATE engine manages the DMEM pointers and delivers messages and interrupts. The ATE enables us to efficiently implement message passing and synchronization primitives (*e.g.*, mutex, barrier) with hardware support.

2.5 Overall Organization of the DPU

Putting it altogether, Figure 1 depicts the overall RAPID DPU System-on-a-Chip (SoC) schematically. The 4 *dpCore* macros and the DMS form the *dpCore* complex that does the bulk of the work. Additionally, the SoC also includes a Power Management Unit (PMU or M0), an ARM Cortex-A9 dual core processor, a MailBox Controller (MBC) and some peripherals (including PCIe and DRAM controllers).

3 RAPID SOFTWARE ARCHITECTURE

RAPID is pluggable to different relational database management systems (RDBMS). We present the integration with a widely used, commercial RDBMS denoted as System X[1]. Figure 2 shows the high level architecture of the integration.

Main goal of the integration is accelerating analytical workloads without modifying user applications or queries. The offloading decision is based on the operations supported and the estimated cost of execution in RAPID. The offloaded portion of the query is executed in RAPID and results are sent back to System X for further processing. In this model, the host database is the single source of truth. Therefore, prior to any offloading, relational tables must be loaded into RAPID and must be kept up to date by propagating changes.

[1]There is an ongoing effort for integration with an open source RDBMS.

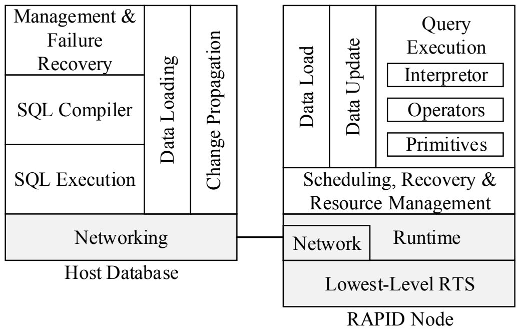

Figure 2: High-level overview of RAPID architecture.

3.1 Query Compilation and Optimization

The SQL query compiler of System X is extended with the cost model of RAPID for taking cost-based offload decisions. System X receives a SQL query, parses it, applies semantic analysis, normalization, factorization, constant folding and several rewrite phases. The plan generator of System X considers *i) full offload*: RAPID-only, *ii) partial offload*: some fragment(s) of the query offloaded or *i) no offload*.

A *fragment* of a query is a candidate for offload if, *a)* the relational operators of the fragment are supported in RAPID and *b)* the relational tables that are required by the operators in the fragment are loaded into RAPID. The cost-based optimizer works bottom-up. For each candidate fragment, it considers the cumulative costs of RAPID execution, network transfer of the results, and post-processing.

Once the plan-generator has determined the cheapest plan, the query execution plan (QEP) can be generated. As part of the code generation step, a logical operator tree is constructed. The logical operator tree typically contains one or many place holder node(s) marking the underlying subtree(s) for RAPID offload. For each subtree below a place holder node, the RAPID compiler (cf. Section 5.2) is executed. It generates, serializes and stores a RAPID QEP in the place holder node.

At System X, the place holder node is mapped to a novel physical operator, called the *RAPID operator*. The RAPID operator is responsible for triggering RAPID execution and consuming the (intermediate) results produced by RAPID.

3.2 Query Execution Offload Model

The execution paradigm in System X is pull-based, following an iterator model. Each operator implements a set of methods: `allocate()`, `start()`, `fetch()`, `close()` and `release()`. Execution proceeds top to bottom and results are propagated bottom-up. Once System X invokes `start()` on the RAPID operator, the query admissibility to RAPID is checked to ensure transactional semantics (cf. Section 3.3). On successful admission check, the stored RAPID QEP is sent to each of the RAPID execution nodes. RAPID nodes then instantiate the received QEP and schedule for execution.

The execution paradigm at the RAPID side is *push-based*. The data is pushed into the operators of the QEP in a streaming manner. As the data flows out of the root operator in RAPID, it is transmitted over the network and placed into the memory buffers of System X with a zero-copy, remote direct memory access (RDMA). Once the

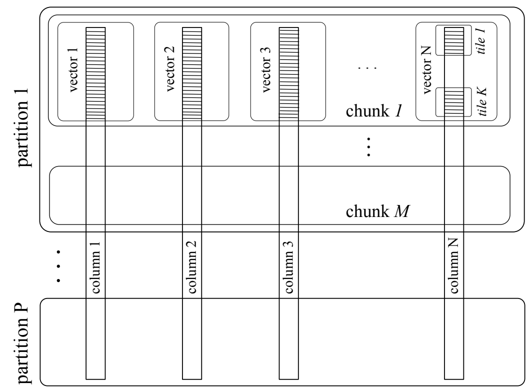

Figure 3: Storage organization of relations in RAPID.

RAPID operator of System X retrieves the data, it employs post-processing such as decoding and other transformations.

In case the admissibility criteria is not met or execution in RAPID fails, the RAPID operator can either fail or fallback to the alternative plan generated for System X-only execution.

3.3 Consistent Query Execution

Transaction semantics require pending changes that a query depends on to be available in RAPID before the query execution. To support this requirement, we rely on the transactional logs collected in in-memory journals. Query checkpointing is the activity of scanning, encoding and sending all pending changes from in-memory journals of the associated tables to RAPID. To avoid long running query checkpoints at query admission, we utilize periodic background threads for scanning and propagating the changes from the journals.

Moreover, a query is admissible to RAPID if the system change number (SCN)–a monotonically increasing timestamp– of the query is not younger than the SCN of any table it operates on. Thus, the SCN of the query at submission is compared to the SCNs of the tables accessed by the fragment.

3.4 Other Components and Networking

RAPID relies on the host database system for durability and failure recovery. If a RAPID node fails, a recovery protocol brings up a spare RAPID node and loads it with the correct partition of the dataset. During recovery, RAPID cluster cannot be used for query offloading, unless there are replicas.

The RAPID metadata holds the information about base tables loaded into RAPID, state of the system, table statistics, table partitioning information and column encodings.

RAPID nodes are connected by a high bandwidth Infiniband network. We implemented a custom networking layer on top which allows scheduling transfers that dynamically adapt to the workload. As a result of carefully using the hardware and the algorithms that avoid congestion, we achieve the peak bandwidth provided by the underlying physical layer in all-to-all communications. Due to reasons of space, we leave the networking details as a topic of a follow-up paper.

4 DATA AND STORAGE MODEL

We describe the data and storage model of RAPID which is optimized for in-memory load and query performance.

4.1 Data and Storage

RAPID stores the entire data in main-memory, while its durability and persistence are provided by the host database system. Relational tables in RAPID consist of one or more ***horizontal partitions***. Each partition contains horizontal slices of relational data called ***chunks***. The data inside a chunk is a set of rows of the table stored in ***columnar layout***. Each column of a table stored inside a chunk is called a ***vector***, which is a flat array of column's data. The sweet-spot for the vector size in the RAPID DPU is 16 KiB, due to enabling double buffering and overlap of DMS transfers with compute. Lastly, the unit of transfer for operators is called a ***tile***, and consists of 64+ rows. The overall storage organization of a relation is depicted in Figure 3.

4.2 Data Encoding

The DPU deliberately lacks native floating point arithmetics for a simple, low-power design. Moreover, there are strict alignment rules for memory addressing which complicate storage of variable width columns. Therefore, RAPID software handles all common data types using fixed width encoding. In particular, we extensively use *decimal scaled binary* number (**DSB**) encoding and *dictionary encoding*.

In decimal scaled binary number encoding, we use a common scale per vector that is selected as the minimum avoiding the decimal point in all values. The common scale is then used for decoding the DSB values back to the original decimal numbers. DSB encoding significantly increases the performance by avoiding floating point calculations. However, for corner cases (*e.g.*, values like 1/3), we store exception values and handle those separately. For fixed and variable length strings, we use *dictionary encoding* as it is the common wisdom in modern OLAP systems [26]. Our dictionary allows updates and range lookups for evaluating prefix and range queries. Additionally, we apply a stack of encodings on each column vector for lightweight compression (*e.g.*, *run length encoding*).

4.3 Updates

RAPID supports periodic updates to loaded base relations that are tracked per update unit (UU). A UU contains a set of changed rows with their system change number (SCN) and expiration SCN. On the RAPID side, a *tracker* module is responsible for fetching data vectors with valid SCN during query processing. The tracker uses indexes to efficiently access valid and latest version of a UU. Hence, accesses from both query processing and update jobs can be serviced concurrently. As time progresses, accumulated updates lead to occupied memory by outdated vectors. A periodic system-wide garbage collection job reclaims the memory of outdated vectors.

4.4 Efficient Data Loading

We employ an efficient mechanism for loading data into RAPID nodes. Load begins with the newly introduced "`LOAD`" command in the host database. Multiple scan threads cooperatively collect and buffer data records directly from the disk. The degree of parallelism is adjusted such that we reach the maximum disk bandwidth of the host database platform. Moreover, by directly reading disk blocks we avoid polluting the buffer pool of the host database.

5 RAPID QUERY EXECUTION

5.1 Query Execution Framework (QEF)

RAPID query execution framework (QEF) provides *i)* push-based execution, *ii)* an actor model for parallelism, *iii)* hardware-aware design, and *iv)* vectorized query processing.

Push-based Query Execution model is suitable for RAPID hardware since it avoids deep call stacks and saves valuable resources (*e.g.*, instruction caches and program stack memory). By controlling the scheduling of operators, we can better utilize the hardware features (*e.g.*, DMS, DMEM). Moreover, it is a natural fit for result sharing [9, 16, 20] and adaptive execution (*e.g.*, for dynamically changing query plans).

Actor Model for Parallelism. RAPID executes multiple hardware threads that communicate among each other with software control due to lack of cache coherency. Therefore, the QEF scheduling needs to be explicitly driven (by the query compiler) in an asynchronous and non-preemptive manner.

Actor model of concurrency is well suited for our requirements. Actors explicitly communicate and share data via asynchronous message passing programming model. Explicit send and receive operations ensure functional correctness when dealing with non-coherent caches. Using the actor model, the QEF hides the programming complexity of using the DPU and non-coherent caches from the data processing operators.

Relational Data Access Model. We designed a novel relational data access model that enables efficient access and hides the hardware complexity. As described earlier, DMS can be used to asynchronously load data into DMEM at memory bandwidth. We program the DMS using *descriptors*–a *descriptor* represents the data transfer with parameters like amount of data, source and destination memory locations. Typically, multiple descriptors are chained one after another to form a loop of DMS transfers. Loops allow reusing a set of descriptors for multiple iterations and overlap memory transfers with the ongoing computation in *dpCores*. Loop based programmed transfers and double buffering hide the memory transfer latency successfully.

The access patterns of data processing operators dictate the DMS programming style. The QEF provides a common interface to operators for specifying their memory access patterns and hides the complexity of the DMS. The interface is called the ***relation accessor*** (**RA**). The RA supports sequential, gather, scatter and partitioned data access patterns. The partitioned access pattern is particularly complicated and useful for many relational data operators like *join* and *group-by*. It provides automatically partitioned data to *dpCores* at memory bandwidth and facilitates parallelization.

Primitives. RAPID query operators carry out data processing via *primitives* that are type-specialized, side-effect-free, short functions operating on columns. Primitives perform one specific type of operation at a time, such as predicate or arithmetic expression evaluation over a set of column vectors.

```
SELECT sum(l_quantity * 0.5), min(l_quantity)
FROM lineitem WHERE l_extendedprice > 100;
```

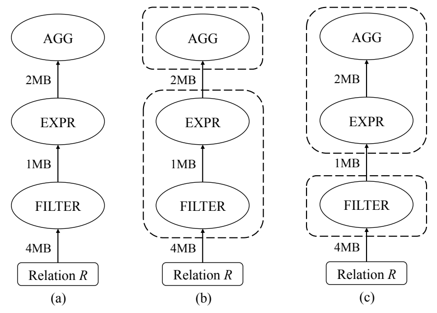

Figure 4: Example task formation for aggregation.

Primitives are defined using a C-like template that specifies the input and output data types supported. RAPID *primitive generator framework* parses the templates and generates C functions for each supported primitive and input/output type combinations at compile time. The generated functions are then compiled and linked into the RAPID binary.

5.2 RAPID Query Optimization

RAPID query compiler and optimizer (***QComp***) is a cost-based physical query optimizer working on top of the logical query optimizations by the host database. *QComp* takes logical query tree as its input and generates a physical query execution plan (**QEP**) using RAPID's cost model and database statistics. The QEP is then executed on RAPID nodes.

RAPID QEP is represented as a directed acyclic graph of physical operators, which is a natural fit for push-based execution model. Moreover, intermediate results across operators can be shared without an explicit materialization.

Physical Optimization. RAPID *QComp* finds the best physical plan among all possible plans in the search space by costing each candidate with the RAPID specific cost model. The search space is already narrowed down by the logical optimization as operators do not need to be re-ordered. RAPID *QComp* traverses the search space by considering the following factors: *i)* Physical operator options available for logical operators, *ii)* additional operators introduced by re-write, *iii)* sharing of results and operators, *iv)* primitive and encoding selection for each column, *v)* degree of parallelization, *vi)* task formation via grouping of operators, and *vii)* DMEM allocation and choice of vector sizes.

RAPID Query Cost Model. RAPID *QComp* relies on *cost model module* for estimating the cost of a query or query fragment. Running on bare-metal without an operating system, RAPID has all the resources under complete control. Hence, the cost model is quite deterministic and accurate. The cost functions take data properties, statistics and various parameters of the physical operators such as vector size, encoding type as input. The total cost of a RAPID operator is analytically modeled on top of data transfer (I/O) and compute cost functions considering the potential overlap. The analytical model is accurately calibrated with micro-benchmarks.

Task formation & DMEM sharing. Task is a group of physical operators executed together without a preemption. Operators within a task pipeline results to each other via DMEM and only results at task boundaries are materialized to DRAM. It is usually desirable to form tasks with maximum number of physical operators for increased pipelining and reduced materialization. However, DMEM is a scarce resource and including more operators in a task reduces the vector size per operator. ***Task formation*** is the process of optimally grouping physical operators and determining their vector sizes.

The task formation starts at the leaf level with tasks comprising physical operators that consume the base relation. As the first step, the minimum amount of DMEM required for input/output vectors, internal operator state and data structures are calculated. Each RAPID operator declares its internal state and data structure sizes at implementation. After selecting a leaf operator, we pick the next downstream operator. Similarly, we compute the minimum DMEM requirements and add this operator to the current task if the remaining DMEM space is sufficient. The procedure halts once there is insufficient DMEM to add another operator. In the second step, the remaining DMEM space is used for increasing the input and output vector sizes. On the other hand, instead of forming tasks with more operators, a task of a single operator with larger input/output vectors can also be formed. By considering these options, we create a set of task formation candidates for a given query and choose the one with the least overall cost of execution.

Task formation example. We illustrate task formation in Figure 4(a) with an aggregate query. Assuming 1 million input rows, column width of 4-bytes and a selectivity of 25 %, Figure 4(a) shows the amount of data transfer across operators. By forming tasks differently, we can reduce the amount of data materialized to DRAM. Thus, the option in Figure 4(c) with lowest overall cost is the choice of task formation.

5.3 Partition Optimization

Partition Operator. The partitioning operator maintains local buffers per partition in the DMEM. Tuples belonging to a given partition are cached in the corresponding local buffer. Once a local buffer fills up, its contents are flushed to the DRAM via DMS thereby significantly reducing the random accesses.

The maximum number of partitions–the ***fan-out***, that can be created in one round of a partitioning operator is typically limited by the number of local buffers, which in turn depends on the DMEM size. Thus, in order to partition data to a larger fan-out, multiple rounds are required. However, multiple rounds need scanning the data multiple times leading to higher cost. Finding the right partitioning scheme, *i.e.*, the number of rounds and the fan-out at each round, is an essential optimization in RAPID.

Required Number of Partitions. The total number of partitions is computed as the estimated total data size divided by the DMEM size. Additionally, required number of partitions is adjusted based on the degree of parallelism by taking the larger of the two. For instance, if the number of partitions is less than the number of cores (*e.g.* 32-cores on the DPU), we target a minimum of 32-ways partitioning.

Partitioning Scheme Optimization. A partitioning scheme is one of all the factorizations of the required number of partitions. For example, for a target of 64 partitions, round 1: 16-way and round 2: 4-way partitioning is one possible scheme. Once a scheme is found, we calculate the vector and buffer sizes such that data stays in DMEM. Hence, our partitioning aims minimizing *i)* rounds of partitioning and *ii)* spill from DMEM to DRAM. We utilize a heuristic based, greedy search for selecting the ideal partitioning scheme. The heuristics used for exploring the factors of a given target number of partitions are: *a)* fan-out at each round must be a power of 2, *b)* fan-out is limited by the max fan-out of a given relation, *c)* minimizing the rounds, and *d)* favoring symmetric fan-outs for different rounds (*e.g.*, favoring 8×8 over 16×4). During the search, we compute the cost of the overall partitioning with a given scheme using our calibrated cost function. Once we search through all the candidates, we end up with the optimized partitioning scheme.

Listing 1: rpdmpr_bvflt_ub4_OPT_TYPE_EQ_cval().

```
cval_reg ← load const_val;
foreach 64 rows in input do
    bvec_reg ← load next-bit-vector;          // val_reg : val in register
    reg1_reg ← bvlduw bvec_reg;               // bv-gather the next val
    repeat
        bvec_reg ← filteq reg1_reg, cval_reg;  } Dual issue.
        reg2_reg ← bvlduw bvec_reg;            }
        bvec_reg ← filteq reg2_reg, cval_reg;  } Dual issue.
        reg1_reg ← bvlduw bvec_reg;            }
    until no more values to gather in next-bit-vector;
    bvec_reg ← filteq reg1_reg, cval_reg;     // filter last val
    output ← store bvec_reg;                  // store the result bit-vector
```

5.4 Data Processing Operators

RAPID operators in QEF are defined by implementing a set of methods (op_dmem_size, op_dram_size, create, open, produce and close). The relation accessor (RA) provides the data to the operator tile by tile ($N \geq 64$ rows). All rows of a tile are consumed at once by vectorized execution. In RAPID, vectorized execution refers to multiple rows at a time processing as in [7, 38] rather than SIMD-based operations. The operator's control flow handles conditions like encoding changes, yielding, end of data and end of partition which are informed by the QEF. In a common case, control flow is a single conditional check per tile and has minimal overhead.

Filter. The filter is a fundamental operator in RAPID that typically touches lots of data in an analytical query. Hence, it must operate efficiently, at memory bandwidth. We utilize DPU features and novel algorithms to achieve this goal.

The main processing in the filter operator is carried out by vectorized primitives, one per each column. The primitives of the filter operator are implemented using *dpCore* database instructions. One of the important filter primitives is shown in Listing 1. The primitive is a tight loop of predicate evaluation over a 4-byte column. It reads the bit-vector of qualifying rows from the previous predicate evaluation and computes an equality filter on those rows using *dpCore* bvld and filteq instructions. The primitive is carefully implemented to benefit from the dual issue feature of the *dpCore*, where subsequent filteq and bvld are issued in the same clock

Listing 2: The `compute_partition_map()` primitive.

```
// 1. Radix the hash input and compute partition sizes.
for $rowid \leftarrow 1$ to $N$ do
    partid ← (hash_input[$rowid$] >> lowbitpos) & mask;
    part_sizes[partid]++;
// 2. Compute a prefix sum over partition sizes.
for sum ← 0; $partid \leftarrow 1$ to $P$ do
    offsets[$partid$] = sum;
    sum + = part_sizes[partid];
// 3. Create the partition map for each row idx.
for $rowid \leftarrow 1$ to $N$ do
    partid ← (hash_input[$rowid$] >> lowbitpos) & mask;
    partition_map[offsets[$partid$]++] = $rowid$;
```

cycle. Finally, the primitive writes out the bit-vector of qualifying rows for a subsequent predicate evaluation or materialization.

We execute the most selective predicate first by re-ordering the predicates. The RA transfers the columns for the most selective predicate into DMEM buffers while hiding the DMS complexity. Then, the primitive scans over a tile of DMEM resident data repeatedly and creates either a list of row-offset identifiers (RIDs) or a bit-vector depending on the expected number of qualifying rows. If the number of expected results are less than 1/32 (size of a RID is 32-bits) of the input, we choose the primitive with RID version; otherwise we choose the bit-vector version. Secondly, for subsequent predicates we use either RID or bit-vector based gather via DMS. The predicates after the first one typically operate on a smaller subset of rows given by the RID-list or the bit-vector from the previous predicate. The RA programs the DMS using one of these and selectively brings only the relevant rows of the columns needed by the predicate. In case the prior predicate is not selective enough, we use double-buffering based DMS transfers. Based on the access pattern, we specialize the primitive selection, *i.e.*, either RID or bit-vector based. Listing 1 shows the primitive of equality predicate with bit-vector based row representation.

The filter operator supports a form of late-materialization in which the projection of columns can be delayed until either all predicates are evaluated or the next operator is executed. Depending on the next operator, we decide to materialize the projection columns in filter or delay further. The hardware-partitioning cannot be combined with DMS gather and it requires materialization in advance. In other cases, we defer the materialization to the next operator by passing the bit-vector or the RID-list as result.

Partitioning. RAPID combines hardware and software partitioning for efficiently partitioning relations.

Hardware Partitioning with DMS. RAPID hardware supports three types of partitioning: *i)* hash-radix, *ii)* range and *iii)* round-robin. The DMS starts by buffering data from DDR in dedicated SRAM banks called column memories (CMEM). The descriptors issued to the DMS initiate further processing of data in CMEM. The hash and range engine can apply a CRC32 checksum and stage the results in another dedicated memory called CRC memory. In hash-radix partitioning, the engine inspects the radix bits of the columns generated in CRC memory and generates a target *dpCore* ID for each row.

Listing 3: The `swpart_partcol_ub4()` primitive.

```
// 1. Gather rows of a partition (p) and write to output.
for $i \leftarrow 1$ to $N_p$ do
    rowidx ← partition_map[$i$];
    output_p[$i$] = input[rowidx];            // Gather & emit seq.
```

In range partitioning, a *dpCore* ID is generated by matching each item in CMEM against one of 32 pre-programmed range values. The *dpCore* IDs generated are also stored in dedicated SRAM banks, called CID memory. In the final stage, the descriptors instruct the DMS to store each row to the specified *dpCore*'s DMEM address.

The third type of partitioning is basic round-robin which additionally provides a mechanism for dealing with skewed data. If the query optimizer knows that a certain range of values are more common, then the frequent range can be assigned to multiple *dpCores* when programming the DMS. During partitioning, the DMS distributes the rows of the frequent range to the programmed set of multiple cores in a round-robin manner and mitigates the effect of data skew.

Vectorized Software-Partitioning. In addition to 32-ways hardware-partitioning, RAPID introduces a novel, vectorized data partitioning in software by using specialized *dpCore* instructions. The essence of the idea is creating a data-partitioning pipeline composed of branch-free, simple tight loops over vectors of columns. First, a vector of CRC32 hash values computed in hardware is provided to the software-partitioning operator in a tile by tile manner. The core primitive of the operator is called `compute_partition_map` and its pseudocode is shown in Listing 2. The primitive essentially computes the partition ID of each row into a vector called *partitioning map*. As shown in the code, it consists of series of tight loops over the hash values.

Once computed, the partitioning map is used for partitioning of all projection columns. For each column, we iterate over the partitioning map and for each partition, issue a gather operation from DMEM using the RIDs. Finally, we store back gathered rows sequentially to the output buffer of the given partition. A primitive instance for a 4-byte column type is shown in Listing 3. Overall, this approach becomes several times faster than a plain, straightforward approach.

Group-by. The SQL *group-by* operator has two versions in RAPID depending on the statistics of number of distinct groups/values (*i.e.*, **NDV**). In the first case, we consider large number of distinct groups which typically go beyond the size of the DMEM. In order to ensure the locality of accesses, we introduce a partitioning phase into the group-by evaluation to distribute distinct groups to different *dpCores*. The physical partitioning operator, which we described earlier, partitions the data such that hash tables built over resulting partitions fit into the DMEM. The RAPID *QComp* calculates the number of rounds of partitioning required and adds the partition operators to the query plan before the group-by operator.

At runtime, if the size of a partition is larger than the estimate, the execution can re-partition the data for that partition as needed. In the last round, if the number of partitions is less than the number of cores, only hardware-partitioning is needed; this is especially

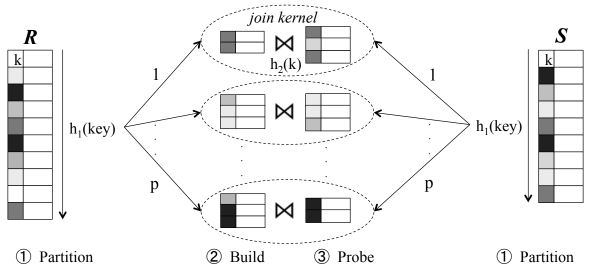

Figure 5: Overview of hash join in RAPID.

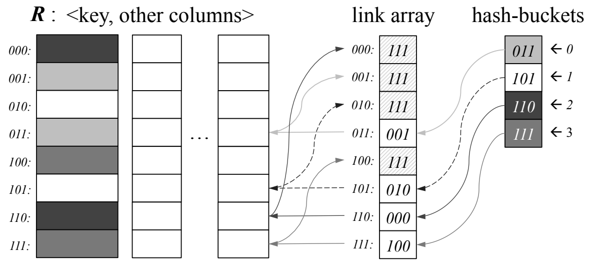

Figure 6: Hash join kernel example.

useful for moderately sized hash tables as extra round-trip through DRAM is avoided.

In the second case, if the hash table accommodating all the groups and the corresponding columns for the aggregation fits into the collective DMEM space of 32 *dpCores*, then we do not introduce a partitioning operator. Instead, we do on-the-fly hardware-partitioning at the input of the group-by operator. As the number of distinct groups is low, a *merge operator* is added to the query plan after the group-by operator. Working on aggregated data, merge operator introduces low overhead.

Other Operators. For reasons of space, we briefly list other supported operators. RAPID provides a vectorized ***Top-K*** operator. ***Window functions*** including analytic aggregates and rank with partition-by clause are supported. We provide ***sorting*** with a partitioning based algorithm. Each *dpCore* utilizes a radix-sorting algorithm. Lastly, we also support ***set operations*** such as MINUS, INTERSECT and UNION.

6 JOIN PROCESSING IN RAPID

Join is one of the bottlenecks in standard benchmarks as well as user workloads. The *equi-join* is the most prevalent join type and traditionally, hash join algorithms are the most efficient option to evaluate equi-joins [5]. Therefore, we focus on equi-joins using hash-based algorithms in RAPID.

6.1 Overview of Hash Join in RAPID

The overview of the *hash join* in RAPID is depicted in Figure 5. The hash join is divided into three stages: *i) partitioning*, *ii) build* and *iii) probe*. The reasons for using a partitioning based join are the following:

a) Providing a large degree of independent parallelism via hundreds/thousands of partitions. It enables different processors / cores to work on different data independently.
b) Partitioning reduces the size of build and probe relations. The build and probe phases (*i.e.*, the join kernel) can be performed within the DMEM with fast random accesses.
c) A non-partitioned hash join results in a large hash table in main-memory where concurrent accesses require locking and synchronization among different cores/processors.

The hash join shown in Figure 5 proceeds as follows. A partitioning stage partitions both input relations using a hash function on the join keys into a number of disjoint partition pairs. In the build stage, the driving relation is scanned, keys are hashed and stored in a hash table. In the probe stage, the outer relation is scanned and keys are hashed to probe the hash table for potential matches. Finally, an output tuple is generated on an exact match.

6.2 Partitioning

The partitioning stage splits one large join operation into a number of smaller and independent join operations that we call *join kernels*. Partitioning does not reduce the amount of data to be exchanged but facilitates efficient subsequent processing due to reduced working set size.

The partitioning operator in RAPID is described in Section 5.4. The main innovation peculiar to partitioning is the co-design of hardware and software to achieve the memory bandwidth offered by the RAPID DPU. In addition, the *QComp* optimizes the query plan by introducing the partitioning operator and its parameters as described in Section 5.3.

6.3 Hash Join Kernel

Join of each partition pair is handled by a *dpCore* using our novel *hash join kernel* optimized for the DPU and DMEM.

To evaluate the join over DMEM-resident small relations, we utilize a compact hash table representation using only bit-arrays. Our mechanism mimics the well known bucket-chained hash table, albeit without any memory pointers. In order to make the addressing of the hash table efficient, we adjust the size of the hash table as a power of two. Based on the database statistics on *NDV*, we further reduce the size of the hash table by 2-4X with respect to number of rows. We then compute a fast modulo using a bit-mask and a shift on top of the hardware computed CRC32 hash values. An intermediate bucket array keeps track of hash table buckets. The data type of the bucket array is as small as possible: If we store N items in the hash table, each element is only $\lceil log_2 N \rceil$ bits. Additionally, we use another array (called *link*) to store the chain of items that map to the same hash bucket. Similarly, the data type of the *link array* is also $\lceil log_2 N \rceil$ bits.

Figure 6 illustrates our join *build kernel* with a simple example. In this example, input relation has 8 tuples and the bucket size is selected as 4. For simplicity, color of the tuples and the buckets denote one of four hash values. The *hash-buckets* array keeps track of the last seen tuple in the input relation having the corresponding hash value. If there is an earlier tuple with the same hash value, then those tuples are chained backwards with the help of the *link* array. To illustrate, for the last tuple in R at offset 7 (*111*), the *link* array is updated at offset 7 and it now points back to the previous occurrence of the same hash value at offset 4 (*100*). Lastly, the tuple's rowid (*i.e.*, offset = *111*), is stored in the *hash-buckets* and the hash table building kernel completes.

The *hash table probe kernel* proceeds as follows. First, we find the hash table bucket by applying bit-wise modulo to the CRC32 value of the probe key. The found hash bucket contains the rowid of the last tuple in R that mapped to this bucket. The tuple from R is compared with the probe tuple and a result tuple is generated on a match. Next, the *link* array is traversed backwards to follow the chain of tuples with the same hash value. The search terminates when the offset pointed by *link* item is *111*, meaning end of the chain. To illustrate, for hash-bucket at index 1, the probe key is first compared with R tuple at offset 5 (*101*). Then by following the *link* array, the next offset to be compared to the search key is found at offset 2 (*010*).

Vectorized, primitive-based execution. Our hash join kernel is divided into number of primitive operations that carry out a single, simple task. For instance, computing the hash table bucket indices from a given array of hash values for N items at once is one such primitive. Another primitive is for join key comparisons; it computes a comparison mask by comparing a set of join key pairs in a tight, simple loop. By carefully integrating such primitives, we achieve a fast, branch-free execution tailored towards the DPU architecture. Moreover, as the primitives are compact, they are also amenable to further hardware-conscious hand tuning–*e.g.*, we create *dpCore* assembly code for careful execution.

6.4 Skew & Statistics Resilient Execution

Our join can handle data skew and inaccuracy in host database statistics where both cases might result in some join partitions larger than the estimated partition size in *QComp*.

Large Skew. If any partition size is larger than a configurable factor of the estimate, we call it *large skew*. It can either result from data skew or statistics with high inaccuracy. In this case, the runtime introduces additional partitioning operators dynamically into the query plan. The partitions with large sizes are further partitioned into smaller ones that fit into the DMEM available. In a broader sense, this technique helps us to alleviate the *large skew* problem.

Small Skew. If a partition size is larger than the estimate by a few rows up to a configurable fraction, we call it *small skew*. The solution for the *large skew* is an overkill for *small skew* as we do not want to pay the overhead of additional partitioning for a few extra rows. In order to tolerate statistics inaccuracies and address the *small skew* problem, we extended our hash join with a DMEM-resilient execution strategy. Once the statistics are not correct, we gracefully overflow the hash table in DMEM to DRAM.

Normally, we assume a partition size calculated by the RAPID *QComp* utilizing host database statistics. During the runtime, if we observe that the estimate is incorrect, then not all the tuples in a given partition will fit into the DMEM. Therefore, we start overflowing upcoming rows to the DRAM resiliently at runtime. As the build phase continues, some fraction of the hash table (*overflow*) accumulates in main-memory and most of it is stored in the DMEM. Consequently, during the probe phase, each part of the hash table must be probed. Under this approach, we observe minimal performance degradation if statistics are off by a small margin while we gracefully proceed with the correct execution.

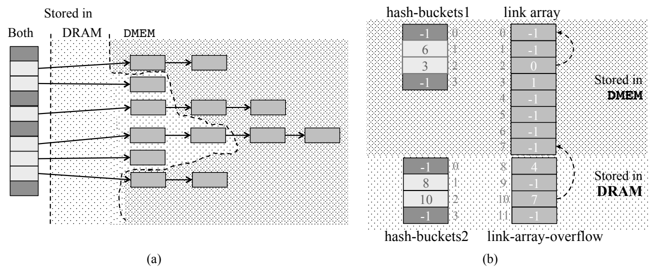

Figure 7: Illustration of DMEM & statistics resilient hash join.

The high-level idea is illustrated in Figure 7(a). For ease of visualizing, we depicted the hash table as a classic chained hash table in Figure 7(a). The dotted lines separate hash table buckets as either being resident in DRAM or in DMEM. Figure 7(b) represents our actual implementation more closely by showing the hash table storage via two integer arrays (*hash-buckets* and *link*). As shown in Figure 7(b), *link* array spans over DMEM and DRAM and chained with integer indices. Finally, *hash-buckets* has two versions, in DRAM and DMEM.

Heavy-Hitter Skew. *Heavy-hitters* are large number of repeated values mapping to the same partition even after several rounds of partitioning. To address the *heavy-hitter* skew, we developed a technique called ***flow-join*** [30]. The technique relies on detecting heavy-hitters at runtime using small approximate histograms with minimal overhead. After the detection, heavy-hitters are broadcasted and processed in a later stage. For details, we refer to [30].

6.5 Other Join Types

Based on our analysis of the customer workloads, we focus on the equi-join as the most frequent join type. In addition, we support *semi-join* and *outer joins* by extending the hash join. We support *anti-join* with a broadcast-like join. For *sort-merge join*, we apply a partitioning-based sorting and a merge-join step. For reasons of space, we omit the details.

7 EXPERIMENTAL EVALUATION

7.1 Micro-benchmarks

We start with micro-benchmarks emphasizing the performance of query processing building blocks in RAPID DPU.

Hardware-partitioning. Partitioning is the core component of many operators like join, group-by and sorting. Therefore, achieving a bandwidth close to the memory bandwidth while partitioning the data on-the-fly in hardware is crucial. In the micro-benchmark, we program the DMS to apply a 32-ways hardware-partitioning on a large relation with 4 columns of 4-bytes each. We run the micro-benchmark with all the hardware-partitioning strategies supported by the DMS: radix; hash with 1, 2 and 4 keys; and range partitioning. Radix partitioning uses the least significant 5 bits of the selected key. Hash partitioning applies hashing on 1, 2 or 4 keys (each in a different run). In range partitioning, we uniformly select the ranges over the input cardinality.

The results in Figure 8 show that our hardware-partitioning achieves around 9.3 GiB/s in all cases. To our knowledge, it outperforms the state-of-the-art hardware accelerator that demonstrated 6 GiB/s for 32-way partitioning [35]. In addition, the DMS does

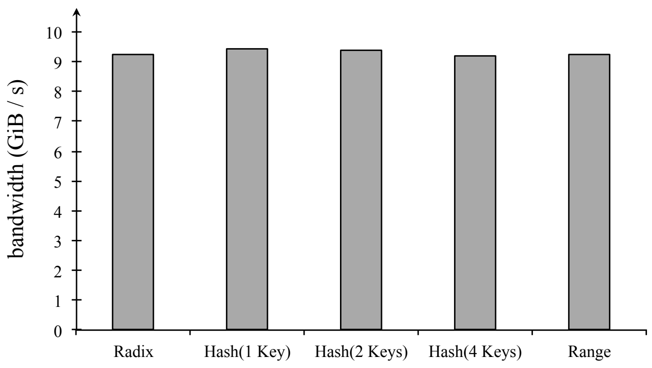

Figure 8: Hardware-partitioning performance of DMS.

all this work in isolation from the *dpCores*. The *dpCores* can do an additional 32-ways or more partitioning in software, thereby increasing our fan-out above 1024 in one pass. While there are optimized partitioning strategies (*e.g.*, software-managed buffers) for x86 architecture that can achieve similar fan-outs in a single pass [10], they are likely to require much higher power budget.

Data movement speed with DMS. As described, data transfers between the DRAM and the operators are carried out by the relation accessor (RA). The relation accessor transfers data into DMEM by using the DMA engine of the DMS and then provides the data to the operator in tiles (*i.e.* 64+ rows at-a-time). The writing of the data is also similar. In the experiment shown in Figure 9, we take a look at the data read/write speed using the DMS. We vary the number of columns from 2 to 32, the tile size from 64 to 256 rows and the access pattern as read (r) or read/write (rw). We show the results with 4-byte columns and note that we observe similar results with 1-, 2- and 8-byte columns.

Figure 9 shows the results obtained. First, there is a slight performance decrease with increasing number of columns. As DMS fetches one column at a time, it observes a small latency overhead in fetching non-contiguous DRAM pages. Second, DMS configuration overheads can be compensated more easily with a larger tile size (cf. 128_rw *vs.* 64_rw). Finally, DMS achieves a bandwidth ≥ 9 GiB/s for a total buffer size of 8 KB (*i.e.*, 128 rows/buffer, 4x4-bytes, double buffering per r&w) which is about 75 % of the peak DDR3 bandwidth, and also close to what we observe in actual queries.

7.2 Operator Performance

Filter performance. In filter experiments, a single *dpCore* achieved a bandwidth of 482 million tuples/s, which translates to 1.65 cycles / tuple, and a peak memory bandwidth of 9.6 GiB/s for 32 *dpCores*. Overall, the computation inside the filter operator is successfully hidden behind the DMS transfers and the operator executes close to the memory bandwidth achieved earlier via DMS reads. The results looked similar to Figure 9; the chart is omitted for brevity.

Software partitioning performance. Software partitioning is important for achieving higher fan-outs on top of the 32-ways hardware-partitioning. To evaluate the performance with respect to the partitioning fan-out, we conducted the experiment shown in Figure 10. In order to enable higher fan-outs, we deliberately used only 2 columns of 4-bytes each and disabled the double-buffering at

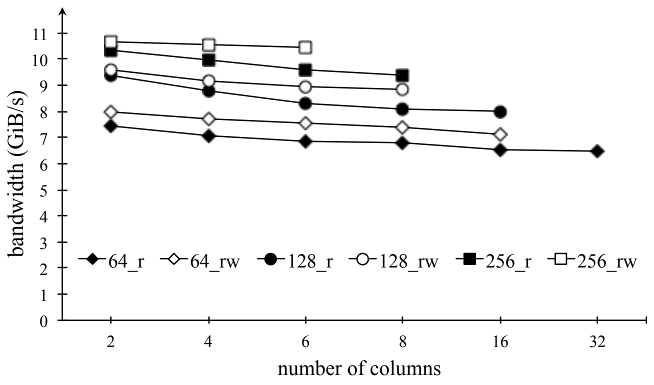

Figure 9: Read/write performance with DMS.

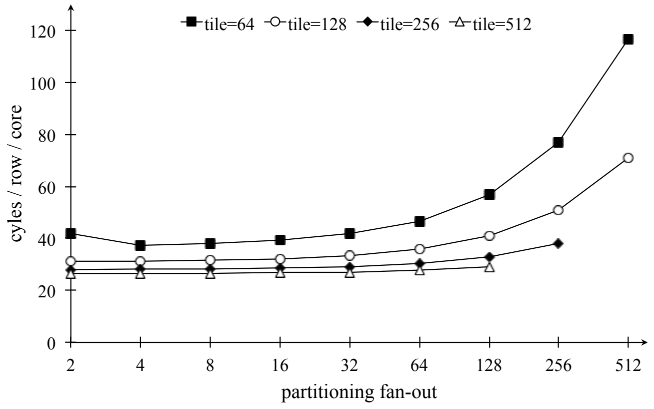

Figure 10: Software partitioning operator performance.

the output. The results in Figure 10 indicate two important things: *1)* Software partitioning performance up to 64-ways partitioning is feasible without significant performance drop, and *2)* it is better to utilize larger input tile size for the software partitioning operator when more DMEM is available. Overall, the performance at 32-ways partitioning is around 948 million rows/s using 32-cores. The bandwidth achieved reaches up to 7-7.6 GiB/s when using a tile size larger than 128. The performance of the software partitioning matches the throughput provided by the hardware-partitioning. Overall, the combined hardware and software partitioning can operate near memory-bandwidth while achieving the goal of 1024-ways fan-out in one round.

7.3 Join Performance Analysis

Build operator performance. The join operator is divided into two micro-operators that are executed in the same task in RAPID. We evaluate the *build operator* performance with respect to *tile* and *hash-buckets* sizes. The results of the experiment is shown in Figure 11. One clear message of the experiment is that *hash-buckets* size does not have any direct impact on the build operator performance. This is mainly due to keeping the *hash-buckets* array entirely in DMEM and the DMEM provides single-cycle latency access even for random accesses. On the other hand, the tile size has an impact on performance similar to other operators. As we increase

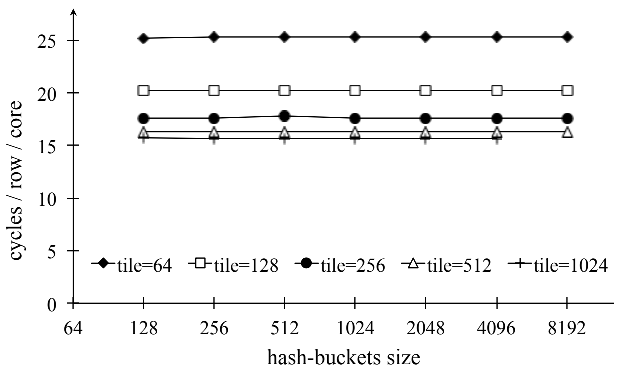

Figure 11: Join build operator performance wrt. tile & hash-buckets sizes.

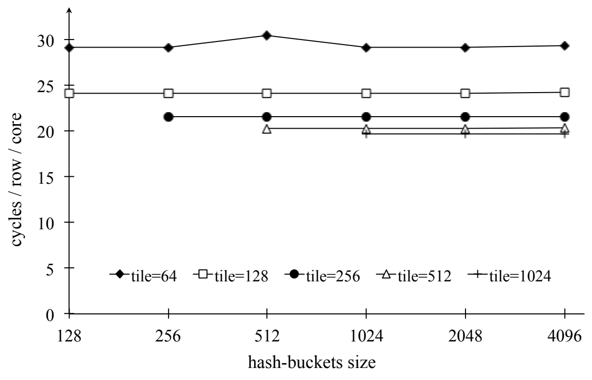

Figure 12: Join probe operator performance wrt. tile & hash-buckets sizes (hit: 50%).

the tile size from 64 to 1024 rows, we see an improvement of 39 %. For the modest tile size of 256 rows, the performance is around 46 million rows/s per *dpCore*. Since completely independent build operations on 32-cores scale linearly, we see a total throughput of ≈ 1.5 billion rows/s per DPU.

Probe operator performance. In the *probe operator* experiment, we fix the hit ratio in the hash table to 50 % and vary the *tile size* and *hash-buckets* parameters. The results in Figure 12 indicate that as long as the hash-buckets is entirely in DMEM, there is no performance impact of using a larger hash-buckets array. In addition, we see a similar effect due to the larger tile size: Performance increases up to 30 % from 64 to 1024 rows per tile. As a consequence, we see an overall probe operator performance that varies between 880 million to 1.35 billion rows/s per DPU.

Performance gain with vectorization. We improve the efficiency and reduce the setup overhead of query execution primitives by executing them on a batch of rows instead of one row at-a-time. To illustrate, we isolated and ran the join operator of TPC-H Q3 with and without vectorization. As shown in Figure 13, we gain around 46 % improvement with vectorized execution enabled. Moreover, the percentage of branch miss predictions reduced significantly. Overall, it demonstrates the importance of vectorization in RAPID.

7.4 Overall System Performance Analysis

We integrated RAPID with a widely used, commercial database management system supporting in-memory query execution. RAPID and the host database together provide end-to-end analytical query processing capabilities. System X is evaluated using in-memory execution on a dual-socket Intel E5-2699 processor with 16 cores and 32 hyper-threads. We summarize important performance aspects in this section.

Performance/power gains. The fundamental design goal for RAPID is providing better performance per watt than other systems running on commodity x86 hardware. We report performance per watt based on the CPU power alone and not the other components. In the experiment shown in Figure 14, we demonstrate the key aspect of RAPID where we achieve in average **15X** more performance per watt compared to in-memory execution in System X on x86. In the experiment, we ran half of TPC-H queries on both

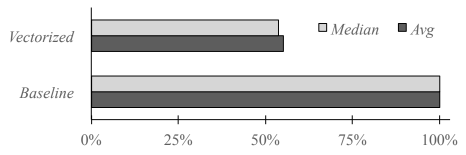

Figure 13: Performance gain in join with vectorization.

System X and RAPID. Performance per watt ratios for queries vary from **10X** to **25X**. Overall, results highlight that RAPID achieves its important performance/power goal.

Execution time distribution. As RAPID is an offload engine for executing analytical queries, it is important to have a reasonable amount of offload in order to get the most benefit from RAPID execution. In order to assess this aspect, we ran a representative half of the TPC-H queries to get an idea of the offload percentage to RAPID. The results are shown in Figure 15. As full table scans, filter, group-by, top-k and joins are offloaded to RAPID entirely, majority of the execution occurs in RAPID. Essentially, we observe that in average **97.57 %** of the elapsed time is spent in RAPID.

Software-only performance of RAPID. Despite the tailoring for the DPU, RAPID software design is also amenable to better performance on x86 due to the novel architecture and algorithms exploiting modern hardware efficiently. In order to demonstrate the software-only merit of RAPID, we compare it with a high-end, in-memory database on commodity x86. We simply ignore the power aspect and compare the two systems on 8 Intel X5-2699 servers with 1 TB of TPC-H data (scale factor 1000). The results are shown in Figure 16. While the speedup of RAPID software varies from **1.2X** to **8.5X**, the average speedup over the ran queries is **2.5X**. It must be noted that RAPID software is not particularly tuned for the x86 architecture. Overall, the important result shows that besides the power advantages, an essential part of performance in RAPID comes from the software design. Finally, RAPID on RAPID hardware runs in average **8.5X** faster than System X and the average speedup that can be attributed to RAPID hardware is in average **3.4X** (8.5/2.5).

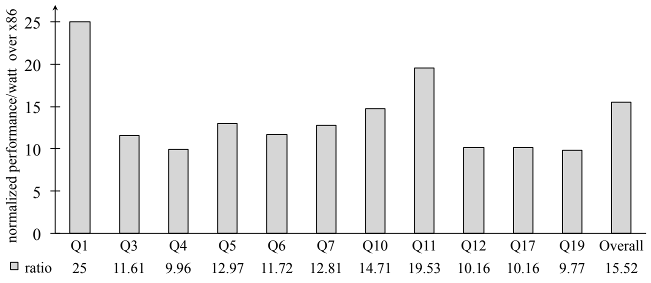

Figure 14: Performance per watt: RAPID vs. x86.

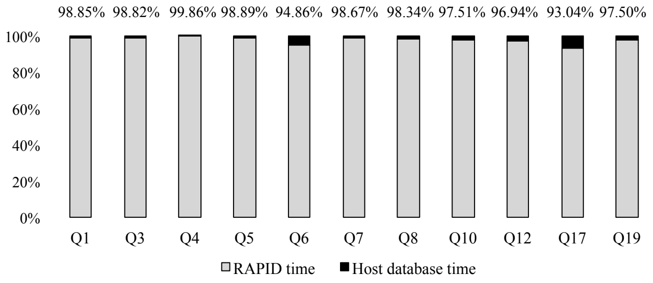

Figure 15: Elapsed time percentage in RAPID and host database. RAPID time average is 97.57 %.

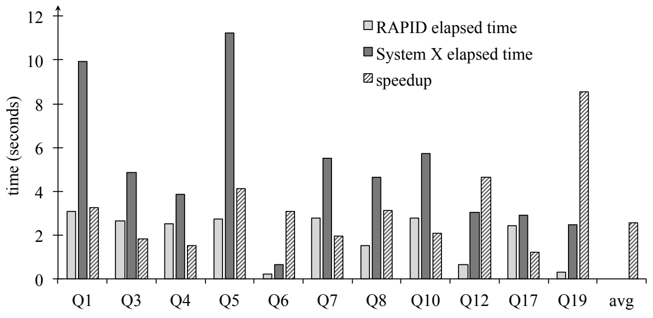

Figure 16: RAPID software vs. System X on x86.

8 RELATED WORK

The idea of using custom hardware for databases is not new and reaches back to the seventies [12]. As one of the early systems, the *Gamma Database Machine* advocates a shared-nothing architecture and parallelism on 32 processors [13]. However, since commodity hardware improved rapidly at the time, the hardware design and implementation for database machines could not keep up and became too expensive [8].

More recently, the *Q100 DPU* introduces a collection of heterogeneous accelerator tiles with a coarse-grained instruction set to express key database processing primitives [37]. In contrast, our ISA is more general-purpose and provides generic acceleration features, asynchronous data movement and partitioning.

HARP [36], a hardware accelerator for range partitioning, is different from our architecture, particularly with respect to the DMS. First, we decouple the high-functionality engines from the CPU cores for independent operation. Second, the usage of our DMS is asynchronous. In contrast, HARP requires the core to execute instructions for every 64 bytes of data. Third, in HARP, the core cannot immediately consume the partitioned data, as data has to go to DRAM first.

SPARC M7 contains a customized *Database Analytics Acceleration* (*DAX*) engine that is introduced for optimizing the performance of the Oracle In-Memory Database [2]. The 32-core M7 processor can utilize eight DAX engines which directly access user memory space. The accelerators can decompress the input data and perform filtering of values, range comparisons, and set membership lookups.

The *Tomahawk MPSoC* [19] consists of four database accelerators, a core manager, global memory and a FPGA connected by a packet-switched network-on-chip. The database accelerators are based on a Tensilica RISC processor that is extended by a specialized instruction set for hashing, bitmap compression and sorted set operations [3].

Hayes et al. claim that SSE/AVX SIMD extensions in x86 are insufficient to express data-level parallelism for database processing [21]. As a solution, they propose new vector ISA extensions for x86 hash table probes. Widx [25] also targets hash lookups, but as an on-chip accelerator. It decouples key hashing from the list traversal and processes multiple keys in parallel on a set of programmable walker units.

Do et al. proposes using smart SSDs for improving performance and energy efficiency [14]. They compiled simple filter and aggregation operators into the SSD firmware. However, the processor in current generation of SSDs quickly becomes a bottleneck. Gold et al. experiments with network processors offloading sequential/index range scans and hash joins [17]. Many researchers investigate the possibilities of offloading database processing to off-the-shelf GPUs [4, 18, 22, 24, 34].

As an alternative to custom hardware, many researchers consider FPGAs for data processing [11, 23, 27, 28, 31, 33]. LINQits introduce pre-designed templates to offload filter, hash group-by/aggregation and hash join operations to FPGAs [11]. Woods et al. implement an intelligent storage engine using FPGAs and SSDs called *Ibex* that supports push-down of projections, complex filters and group-by aggregation [33]. *Netezza* [6], IBM's data analytics appliance pushes selections and projections to multiple FPGAs in parallel.

In their seminal work, Boncz et al. [7, 38] lay the foundations of vectorized query processing, which also influence the design of the vectorized processing in RAPID.

9 CONCLUSIONS

As database engines need more and more performance, they are also becoming power hungry. In order to address the performance/power efficiency, we need to re-design the query processing engine by looking at the software design together with the hardware architecture. We present the hardware/software co-design of the RAPID in-memory query processing engine. In the paper, we describe the RAPID DPU architecture and the software design of the RAPID analytic query processing engine tailored for each other. Through an experimental analysis, we show that RAPID achieves at least an order of magnitude higher performance per watt in comparison to an existing high-end system running on commodity hardware.

REFERENCES

[1] S. R Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar. 2017. A Many-core Architecture for In-memory Data Processing. In *MICRO (MICRO '17)*. 245–258.

[2] K. Aingaran, S. Jairath, G. K. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki. 2015. M7: Oracle's Next-Generation Sparc Processor. *MICRO* 35, 2 (2015), 36–45.

[3] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner. 2014. An Application-specific Instruction Set for Accelerating Set-oriented Database Primitives. In *SIGMOD*.

[4] P. Bakkum and K. Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*. 94–103.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96.

[6] F. Beier and K. Stolze. 2017. Architecture of a data analytics service in hybrid cloud environments. *it - Information Technology* 59, 3 (2017), 151–158.

[7] P. A. Boncz, M. Zukowski, and N. Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, Vol. 5. 225–237.

[8] H. Boral and D. J. DeWitt. 1983. Database Machines: An Idea Whose Time Passed? A Critique of the Future of Database Machines. In *Database Machines, International Workshop*. 166–187.

[9] G. Candea, N. Polyzotis, and R. Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *PVLDB* 2, 1 (2009), 277–288.

[10] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. on Knowl. and Data Eng.* 27, 7 (2015), 1754–1766.

[11] E. S. Chung, J. D. Davis, and J. Lee. 2013. LINQits: Big Data on Little Clients. In *ISCA (ISCA '13)*. 261–272.

[12] David J. DeWitt. 1978. DIRECT - a Multiprocessor Organization for Supporting Relational Data Base Management Systems. In *Proceedings of the 5th Annual Symposium on Computer Architecture (ISCA '78)*. 182–189.

[13] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. 1990. The Gamma Database Machine Project. *IEEE Trans. on Knowl. and Data Eng.* 2, 1 (1990), 44–62.

[14] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *SIGMOD (SIGMOD '13)*. ACM, 1221–1230.

[15] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D.l Popescu, A. Ailamaki, and B. Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* (2012), 37–48.

[16] G. Giannikis, G. Alonso, and D. Kossmann. 2012. SharedDB: Killing One Thousand Queries with One Stone. *PVLDB* 5, 6 (Feb. 2012), 526–537.

[17] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. 2005. Accelerating Database Operators Using a Network Processor. In *DaMoN*.

[18] N. K. Govindaraju and D. Manocha. 2005. Efficient Relational Database Management using Graphics Processors. In *DaMoN*.

[19] S. Haas, O. Arnold, B. Nöthen, S. Scholze, G. Ellguth, A. Dixius, S. Höppner, S. Schiefer, S. Hartmann, S. Henker, T. Hocker, J. Schreiter, H. Eisenreich, J. Schlüßler, D. Walter, T. Seifert, F. Pauls, M. Hasler, Y. Chen, H. Hensel, S. Moriam, E. Matús, C. Mayr, R. Schüffny, and G. P. Fettweis. 2016. An MPSoC for energy-efficient database query processing. In *DAC*. 112:1–112:6.

[20] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD (SIGMOD '05)*. 383–394.

[21] T. Hayes, O. Palomar, O. S. Unsal, A. Cristal, and M. Valero. 2012. Vector Extensions for Decision Support DBMS Acceleration. In *MICRO '12*. IEEE Computer Society, 166–176.

[22] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4 (2009), 21:1–21:39.

[23] Z. István, G. Alonso, M. Blott, and K. A. Vissers. 2013. A flexible hash table design for 10GBPS key-value stores on FPGAS. In *FPL*. 1–8.

[24] T. Kaldewey, G. M. Lohman, R. Müller, and P. B. Volk. 2012. GPU join processing revisited. In *DaMoN*. 55–62.

[25] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO (MICRO '13)*.

[26] I. Müller, C. Ratsch, and F. Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. 283–294.

[27] R. Müller, J. Teubner, and G. Alonso. 2009. Streams on Wires - A Query Compiler for FPGAs. *PVLDB* 2, 1 (2009), 229–240.

[28] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. 2013. Flexible Query Processor on FPGAs. *PVLDB* 6, 12 (2013), 1310–1313.

[29] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. 2009. A Comparison of Approaches to Large-scale Data Analysis. In *SIGMOD*. 165–178.

[30] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*. 1194–1205.

[31] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. W. Asaad. 2012. Database analytics acceleration using FPGAs. In *PACT*. 411–420.

[32] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. *HPCA* (2014), 488–499.

[33] L. Woods, Z. István, and G. Alonso. 2014. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *PVLDB* 7, 11 (2014), 963–974.

[34] H. Wu, G. . Diamos, S. Cadambi, and S. Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *MICRO*. 107–118.

[35] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. 2013. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. In *ISCA (ISCA '13)*. 249–260.

[36] L. Wu, R. J Barker, M. A Kim, and K. A. Ross. 2013. Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning. In *ISCA*.

[37] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. 2015. The Q100 Database Processing Unit. *MICRO* 35 (2015).

[38] M. Zukowski, P. Boncz, N. Nes, and S. Héman. 2005. MonetDB/X100 - A DBMS in the CPU cache. In *IEEE Data Eng. Bull.*, Vol. 28. 17–22.