Q100: The Architecture and Design of a Database Processing Unit

Lisa Wu Andrea Lottarini Timothy K. Paine Martha A. Kim Kenneth A. Ross
Columbia University, New York, NY
{lisa,lottarini,martha,kar}@cs.columbia.edu/tkp2108@columbia.edu

Abstract

In this paper, we propose Database Processing Units, or DPUs, a class of domain-specific database processors that can efficiently handle database applications. As a proof of concept, we present the instruction set architecture, microarchitecture, and hardware implementation of one DPU, called Q100. The Q100 has a collection of heterogeneous ASIC tiles that process relational tables and columns quickly and energy-efficiently. The architecture uses coarse grained instructions that manipulate streams of data, thereby maximizing pipeline and data parallelism, and minimizing the need to time multiplex the accelerator tiles and spill intermediate results to memory. This work explores a Q100 design space of 150 configurations, selecting three for further analysis: a small, power-conscious implementation, a high-performance implementation, and a balanced design that maximizes performance per Watt. We then demonstrate that the power-conscious Q100 handles the TPC-H queries with three orders of magnitude less energy than a state of the art software DBMS, while the performance-oriented design outperforms the same DBMS by 70X.

Categories and Subject Descriptors C.3 [*Special-purpose and application-based systems*]: Microprocessor/microcomputer applications

Keywords Accelerator; Specialized functional unit; Streaming data; Microarchitecture; Database; DPU

1. Introduction

Harvard Business Review recently published an article on Big Data that leads with a piece of artwork by Tamar Cohen titled "You can't manage what you don't measure" [28]. It goes on to describe big data analytics as not just important for business, but essential. The article emphasized that analyses must process large *volumes* of a wide *variety*, and at real-time or nearly real-time *velocity*. With the big data technology and services market forecast to grow from $3.2B in 2010 to $16.9B in 2015 [23], and 2.6 exabytes of data created each day [28], it is imperative for the research community to develop machines that can keep up with this data deluge.

For its part, the Database Management System (DBMS) software community has been exploring optimizations such as using column stores [1–3, 24, 27, 34], pipelining operations [4, 6], and vectorizing operations [40], to take advantage of commodity server hardware.

This work applies those same techniques, but in hardware, to construct a domain-specific processor for databases. Just as conventional DBMSs operate on data in logical entities of tables and columns, our processor manipulates these same data primitives. Like DBMSs use software pipelining between relational operators to reduce intermediate results, we too can exploit pipelining between relational operators implemented in hardware to increase throughput and reduce query completion time. In light of the SIMD instruction set advances in general purpose CPUs in the last decade, DBMSs also vectorize their implementations of many operators to exploit data parallelism. Our hardware does not use vectorized instructions, but exploits data parallelism by processing multiple streams of data, corresponding to tables and columns, at once.

Streams of data. Pipelines. Parallel functional units. All of these techniques have long been known to be excellent fits for hardware, creating what we believe to be an opportunity to address some very practical, real-world concerns regarding big data. Our vision is of a class of domain-specific processors called DPUs, that is analogous to GPUs. Whereas GPUs target graphics applications, DPUs target analytic database workloads. As GPUs operate on vertices, DPUs operate on tables and columns.

We design and evaluate a first DPU, called Q100. The Q100 is a performance and energy efficient data analysis accelerator. It contains a heterogeneous collection of

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541961

fixed-function ASIC tiles, each of which implements a well-known relational operator, such as a join or sort. The Q100 tiles operate on streams of data corresponding to tables and columns, over which the microarchitecture aggressively exploits pipeline and data parallelism.

This paper makes the following contributions:

- An energy-efficient instruction set architecture for processing data-analytic workloads, with instructions that both closely match standard relational primitives and are good fits for hardware acceleration.
- A high-performance, energy-efficient DPU, called Q100. Using custom processing tiles, all physically designed in 32nm standard cells, this chip provides orders of magnitude improvements in both TPC-H performance and energy consumption over state-of-the-art DBMS software.
- An in-depth tour of the Q100 design process, revealing the many opportunities, pitfalls, tradeoffs, and overheads one can expect to encounter when designing small accelerators to process big data.

In the following section, we present the design and specification of the Q100 ISA, the first DPU ISA. Then, in Section 3, we detail the step-by-step design process of the Q100, starting from physical design of the tiles and working up towards an exploration of resource scheduling algorithms. The results of this process are three Q100 designs, each optimized for a particular objective (e.g., low power, high performance, etc.). In Section 4 we compare the performance and energy consumption of TPC-H queries running on these Q100 designs to a state of the art, column store DBMS running on a Sandybridge server. Before concluding, we close with a survey of related work in Section 5.

2. Q100 Instruction Set Architecture

Q100 instructions implement standard relational operators that manipulate database primitives such as columns, tables, and constants. The producer and consumer relationship between operators are captured with dependencies specified by the instruction set architecture. Queries are represented as graphs of these instructions with the edges representing data dependencies between instructions. For execution, a query is mapped to a spatial array of specialized processing tiles, each of which carries out one of the primitive functions. When producer-consumer node pairs are mapped to the same temporal stage of the query, they operate as a pipeline with data streaming direction from producer to consumer.

The basic instruction is called a *spatial instruction* or *sinst*. These instructions implement standard SQL-esque operators, namely *select*, *join*, *aggregate*, *boolgen*, *colfilter*, *partition*, and *sort*. Figure 1 shows a simple query written in SQL to produce a summary sales quantity report per season for all items shipped as of a given date. Figure 1 bottom shows the query transformed into Q100 spatial instructions, retaining data dependencies. Together, *boolgen* and *colfil-*

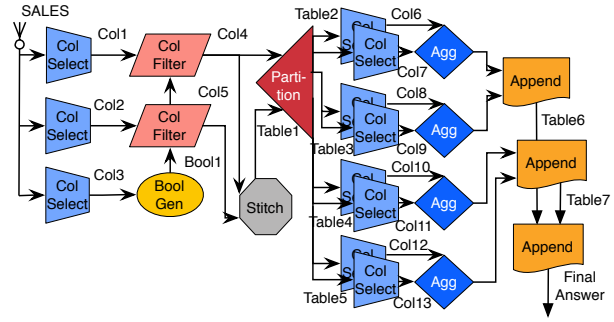
▷ *Sample query written in SQL*

```
SELECT    S_SEASON,
          SUM(S_QUANTITY) as SUM_QTY
FROM      SALES
WHERE     S_SHIPDATE <= '1998-12-01' - INTERVAL '90' DAY
GROUP BY  S_SEASON
ORDER BY  S_SEASON
```

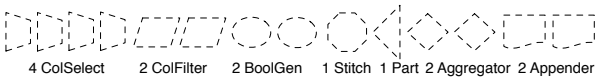
▷ *Sample query plan converted to proposed DPU spatial instructions*

```
Col1      ← ColSelect(S_SEASON from SALES);
Col2      ← ColSelect(S_QUANTITY from SALES);
Col3      ← ColSelect(S_SHIPDATE from SALES);
Bool1     ← BoolGen(Col3, '1998-09-02', LTE);
Col4      ← ColFilter(Col1 using Bool1);
Col5      ← ColFilter(Col2 using Bool1);
Table1    ← Stitch(Col4, Col5);
Table2..Table5 ← Partition(Table1 using key column Col4);
Col6..7   ← ColSelect(Col4..5 from Table2);
Col8..9   ← ColSelect(Col4..5 from Table3);
Col10..11 ← ColSelect(Col4..5 from Table4);
Col12..13 ← ColSelect(Col4..5 from Table5);
Table6 ← Append(Aggregate(SUM Col7 from Table2 group by Col6),
                Aggregate(SUM Col9 from Table3 group by Col8));
Table7 ← Append(Aggregate(SUM Col11 from Table4 group by Col10),
                Aggregate(SUM Col13 from Table5 group by Col12));
FinalAns ← Append(Table6, Table7);
```

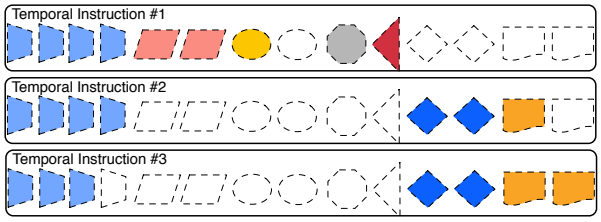
Figure 1. An example query (top) is transformed into a *spatial instruction* plan (bottom) that map onto an array of heterogeneous specialized tiles for efficient execution.



(a) Unrestricted Graph of Spatial Instructions



(b) Resource Profile



(c) Resource-Aware Temporal Instructions

Figure 2. The example query from Figure 1 is mapped onto a directed graph with nodes as relational operators and edges as data dependencies. Given a set of Q100 resources, the graph is broken into three *temporal instructions* that are executed in sequence, one after another.

	Tile	Area mm^2	Area % Xeon [a]	Power mW	Power % Xeon	Critical Path ns	Design Width (bits) Record	Column	Comparator	Other Constraint
Functional	**Aggregator**	0.029	0.07%	7.1	0.14%	1.95		256	256	
	ALU	0.091	0.21%	12.0	0.24%	0.29		64	64	
	BoolGen	0.003	0.01%	0.2	<0.01%	0.41		256	256	
	ColFilter	0.001	<0.01%	0.1	<0.01%	0.23		256		
	Joiner	0.016	0.04%	2.6	0.05%	0.51	1024	256	64	
	Partitioner	0.942	2.20%	28.8	0.58%	***3.17	1024	256	64	
	Sorter	0.188	0.44%	39.4	0.79%	2.48	1024	256	64	1024 entries at a time
Auxiliary	**Append**	0.011	0.03%	5.4	0.11%	0.37	1024	256		
	ColSelect	0.049	0.11%	8.0	0.16%	0.35	1024	256		
	Concat	0.003	0.01%	1.2	0.02%	0.28		256		
	Stitch	0.011	0.03%	5.4	0.11%	0.37		256		

[a] Intel E5620 Xeon server with 2 chips. Each chip contains 4 cores 8 threads running at 2.4 GHz with 12 MB LLC, 3 channels of DDR3, providing 24 GB RAM. Comparisons are done using estimated single core area and power consumption derived from published specification.

Table 1. The physical design characteristics of Q100 tiles post place and route, and compared to a Xeon core. ***The slowest tile, the partitioner, determines the frequency of Q100 at 315 MHz.

ter for example, support the `WHERE` clauses, while *partition* and *sort* are to support the `ORDER BY` clauses found in many query languages. Generating a column of booleans using a condition specified via a `WHERE` clause then filtering the projected columns is not a new concept, and is implemented by Vectorwise [40], a commercial DBMS, and other database software vendors that use column-stores.

Other helper spatial instructions perform a variety of auxiliary functions such as (1) tuple reconstruction (i.e. *stitch* individual columns of a row back into a row, or *append* smaller tables with the same attributes into bigger tables) to transform columns into intermediate or final table outputs, and (2) `GROUP BY` and `ORDER BY` clauses to perform aggregations and sorts (i.e. *concatenate* entries in a pair of columns to create one column in order to reduce the number of sorts performed when there are multiple `ORDER BY` columns).

In situations where a query does not fit on the array of available Q100 of tiles, it must be split into multiple temporal stages. These temporal stages are called *temporal instructions*, or *tinsts*, and are executed in order. Each tinst contains a set of spatial instructions, pulling input data from the memory subsystem and pushing completed partial query results back to the memory subsystem. Figure 2 walks through how a graph representation of spatial instructions, implementing the example query from Figure 1, is mapped onto available specialized processing tiles. Figure 2 (a) shows the entire query as one graph with each shape representing a different primitive and edges representing producer-consumer relationships (i.e., data dependencies). Figure 2 (b) shows an example array of specialized hardware tiles, or a *resource profile*, for a particular Q100 configuration. Figure 2 (c) depicts how the query must to be broken into three temporal instructions, because the resource profile does not have enough column selectors, column filters, aggregators, or appenders at each stage.

This instruction set architecture is energy efficient because it closely matches building blocks of our target domain, while simultaneously encapsulating operations that can be implemented very efficiently in hardware. Spatial instructions are executed in a dataflow-esque style seen in dataflow machines in the 80's [12, 17], in the 90's [19], and more recently [13, 31, 35], eliminating complex issue and control logic, exposing parallelism, and passing data dependencies directly from producer to consumer. All of these features provide performance benefit and energy savings.

3. Q100 Microarchitecture

In this section we walk through the Q100 design process. We start with descriptions of the hardware tiles that implement the Q100 ISA including their size and delays when implemented in 32nm physical design (Section 3.1). Then, using 19 TPC-H queries as benchmarks we perform a detailed Q100 design space exploration with which we explore the tradeoffs and select three interesting Q100 designs: minimal power, peak performance, and a balanced design that offers maximal performance per Watt (Section 3.2). We then explore the impact of communication – both intra tile and with memory – on these three designs (Section 3.3) as well as the instruction scheduling algorithm (Section 3.4).

3.1 Q100 Tile Implementation and Characterization

The Q100 contains eleven types of hardware tile corresponding to the eleven operators in the ISA. As in the ISA, we break the discussion into core functional tiles and auxiliary helper tiles. The facts and figures of this section are summarized in Table 1, while the text that follows focuses on the design choices and tradeoffs. The slowest tile determines the clock cycle of the Q100. As Table 1 indicates, the partitioner limits the Q100 frequency to 315 MHz.

Methodology. Each tile has been implemented in Verilog and synthesized, placed, and routed using Synopsys 32nm Generic Libraries[1] with the Synopsys [36] Design and IC Compilers to produce timing, area, and power numbers. We report the post-place-and-route critical path of each design as logic delay plus clock network delay, adhering to the industry standard of reporting critical paths with a margin.

Q100 functional tiles. The *sorter* sorts its input table using a designated key column and a bitonic sort [26]. In general, hardware sorters operate in batches, and require all items in the batch to be buffered at the ready prior to the start of the sort. As buffers and sorting networks are costly, this limits the number of items that can be sorted at once. For the Q100 tile, this is 1024 records, so to sort larger tables, they must first be partitioned with the partitioner.

The *partitioner* splits a large table into multiple smaller tables called partitions. Each row in the input table is assigned to exactly one partition based on the value of the key field. The Q100 implements range partitioner, which splits the space of keys into contiguous ranges. We chose this because it is tolerant of irregular data distributions [39] and produces ordered partitions, making it a suitable precursor to the sorter.

The *joiner* performs an inner-equijoin of two tables, one with a primary key and the other with a foreign key. To keep the design simple, the Q100 currently supports only inner-equijoins. It is by far the most common type of join, though extending the joiner to support other types (e.g., outer-joins) would not increase its area or power substantially.

The *ALU* tile performs arithmetic and logical operations on two input columns, producing one output column. It supports all arithmetic and logical operations found in SQL (i.e., ADD, SUB, MUL, DIV, AND, OR, and NOT) as well as constant multiplication and division. We use these latter operations to work around the current lack of a floating point unit in the Q100. In its place, we multiply any SQL *decimal* data type by a large constant, apply the integer arithmetic, finally divide the result by the original scaling factor, effectively using fixed point to support single precision floating point arithmetic, as most domain-specific accelerators have done. SQL does not specify precision requirements for floating point calculations and most commercial DBMS supports either single-precision floating point and/or double-precision floating point calculations.

The *boolean generator* compares an input column with either a constant or a second input column, producing a column of boolean values. Using just two hardware comparators, the tile provides all six comparisons used in SQL (i.e. EQ, NEQ, LTE, LT, GT, GTE). While this tile could have been combined with the ALU, offering two tiles à la carte leaves more flexibility when allocating tile resources. The boolean generator is often paired with the column filter (described next) with no need for an ALU. It is also often used in a chain or tree to form complex predicates, again not always in 1-to-1 correspondence with ALUs.

The *column filter* takes in a column of booleans (from a boolean generator) and a second data column. It outputs the same data column but dropping all rows where the corresponding bool is false.

Finally the *aggregator* takes in the column to be aggregated and a "group by" column whose values determine which entries in the first column to aggregate. For example, if the query sums purchases by zipcode, the data column are the purchase totals while the group-by is the zipcode. The tile requires that both input columns arrive sorted on the group-by column so that the tile can simply compare consecutive group-by values to determine where to close each aggregation. This decision has tradeoffs. A hash-based implementation might not require pre-sorting, but it would require a buffer of unknown size to maintain the partial aggregation results for each group. The Q100 aggregator supports all aggregation operations in the SQL spec, namely MAX, MIN, COUNT, SUM, and AVG.

Q100 auxiliary tiles. The *column selector* extracts a column from a table, and the *column stitcher* does the inverse, taking multiple input columns (up to a maximum total width) and producing a table. This operation often precedes partitions and sorts where queries frequently require column A sorted according to the values in column B. The *column concatenator* concatenates corresponding entries in two input columns to produce one output column. This can cut down on sorts and partitions when a query requires sorting or grouping on more than one attribute (i.e., column). Finally, the *table appender* appends two tables with the same schema. This is often used to combine the results of per-partition computations.

Modifications to TPC-H due to tile limitations. The design parameters such as record, column, key, and comparator widths are generally sized conservatively. However, we encountered a small number of situations where we had to modify the layout of an underlying table or adjust the operation, though never the semantics, of a query. When a column width exceeds the 32 byte maximum column width the Q100 can support, we divide the wide column vertically into smaller ones of no more than 32 bytes and process them in parallel. Out of 8 tables and 61 columns in TPC-H, just 10 were split in this fashion. Similarly, because the Q100 does not currently support regular expression matching, as with the SQL `LIKE` keyword, the query is converted to use as many `WHERE EQ` clauses as required. These are all minor side effects of the current Q100 design and may not be required in future implementations.

[1] Normal operating conditions (1.25V supply voltage at $25°C$) with high threshold voltage to minimize leakage.

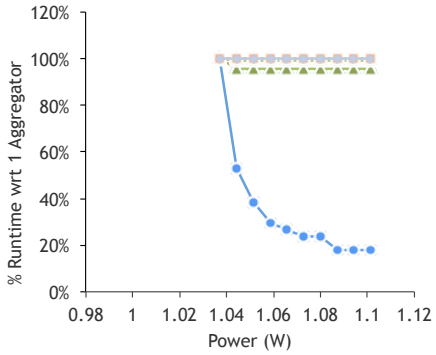


Figure 3. Aggregator sensitivity study shows that Q1 is the only query that is sensitive to number of aggregators, and its performance plateaus beyond 8 tiles.

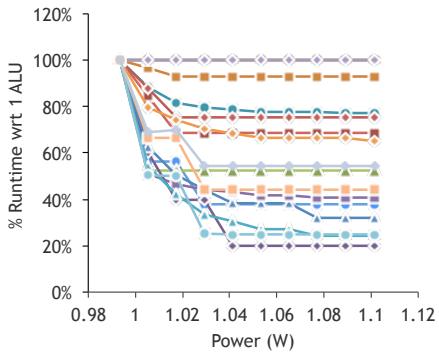


Figure 4. ALU tiles are more power hungry than aggregators, but adding more ALUs helps most query's performance. This tradeoff necessitates an exploration of the design space varying number of ALUs.

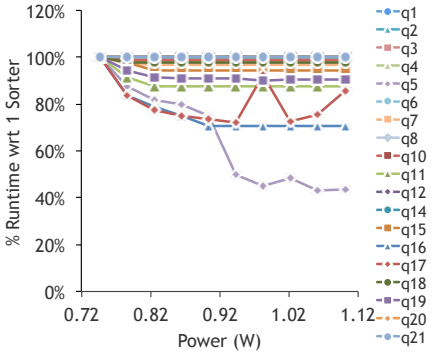


Figure 5. Sorter tiles are the most power hungry, dissipating almost 40 mW per tile. Q17 exhibits a corner case where the scheduler makes bad decisions causing performance to degrade as number of sorters increase.

3.2 Q100 Tile Mix Design Space Exploration

To understand the relative utility of each type of tile, and the tradeoffs amongst them, we explore a wide design space of different sets of Q100 tiles. We start with a sensitivity analysis of TPC-H performance, evaluating each type of tile in isolation to bound the maximum number of useful tiles of each type. We then carry out a complete design space exploration considering multiple tiles at once, from which we understand the power performance shape of the Q100 space and select three configurations (i.e., tile mixtures) for further analysis.**Methodology.** We have developed a functional and timing Q100 simulator in C++. The function and throughput of each tile have been validated against simulations of the corresponding Verilog. As we do not yet have a compiler for the Q100, we have manually implemented each TPC-H query in the Q100 ISA. Using the simulator, we have confirmed that the Q100 query implementations produce the same results as the SQL versions running on MonetDB [8]. Given a query and a Q100 configuration, a scheduling algorithm described and evaluated later in Section 3.4 schedules each query into a sequence of temporal instructions. The simulator produces cycle counts, which we convert to wall clock time using a Q100 frequency of 315 MHz.

Tile	Maximum Useful Count	"Tiny" Tile	Tile Counts Explored
Aggregator	4	X	4
ALU	5		1 ... 5
BoolGen	6	X	6
ColFilter	6	X	6
Joiner	4	X	4
Partitioner	5		1 ... 5
Sorter	6		1 ... 6
Append	8	X	8
ColSelect	7	X	7
Concat	2	X	2
Stitch	3	X	3

Table 2. "Tiny" tiles are the ones that dissipate <10 mW per tile as seen in Table 1. We eliminate configurations that will result in similar power or performance characteristics before running the design space exploration to cut down on the number of Q100 configurations under consideration.

Tile count sensitivity. To understand how sensitive Q100 is to the number of each type of tile, say aggregators, we simulate a range of Q100 configurations, sweeping the number aggregators, while holding all other types of tiles at sufficiently high counts so as not to limit performance. Figure 3 shows how the runtime of each TPC-H varies with the number of aggregators in the design. Having run this experiment for each of the eleven types of tile, we highlight three sets of results here and in Figures 3-5. Just one query, Q1, is sensitive to the number of aggregators, while the performance of the others is not affected. On the other hand, many queries benefit from more ALUs, with improvements flattening beyond 5 ALUs. Note that the aggregator and the ALU experiments are plotted with the same X-axis, while the sorter, at 39.4 mW per tile, covers a much larger power range. Across the board, these sensitivity experiments reveal that for all queries and all tiles, performance plateaus by or before ten tiles of each type.

Design space parameters. A complete design space exploration, with 1 to 10 instances of each of 11 types of tile, would result in an overwhelmingly large design space. Using the tile sizes and the results of the sensitivity study above, we are able to prune the space substantially. First, we eliminate all of the "negligible" tiles from the design space. There are the tiles that are so tiny that the difference between one or two or ten will have a negligible impact on the results

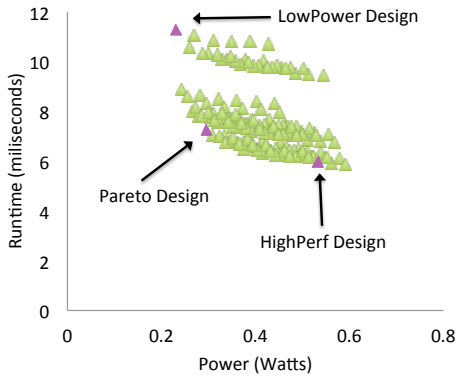


Figure 6. Out of 150 configurations, we selected three designs for further evaluation: LowPower for an energy-conscious configuration, HighPerf for a performance-conscious configuration, and Pareto for a design that maximizes performance per Watt.

of the exploration. For these tiny tiles, defined to be those eight tiles that consume less than 10 mW, we use the per-tile sensitivity analysis to identify the maximum number of useful tiles, and always allocate this many instances. For the remaining three non-tiny tiles (the ALU, partitioner, and sorter), we explore the design space only up to that count. Table 2 summarizes how the tile size and sensitivity reduce the design space from millions to 150 configurations.

Power-performance design space. Figure 6 plots the power-performance tradeoffs for 150 Q100 designs. Amongst these configurations we select the three designs indicated in the plot for further evaluation:

1. An energy-conscious design point (LowPower) that has just 1 partitioner, 1 sorter, and 1 ALU, and consumes the lowest power amongst all the configurations.
2. A balanced design on the Pareto-optimal frontier (Pareto), that, with 2 partitioners, 1 sorter, and 4 ALUs, provides the most performance per Watt amongst the designs.
3. A performance-optimized design (HighPerf), with 3 partitioners, 6 sorters, and 5 ALUs, that maximizes performance at the cost of a relatively higher power consumption.

3.3 Q100 Communication Needs

Having explored the Q100's computational needs, we now turn to its communication needs, both on-chip intra-tile communication and off-chip with memory. Because the target workload is large scale data, each of these channels will need to support substantial throughput.

Communication topology. In the experiments and simulations thus far, we have assumed all-to-all communication for all of the Q100 tiles and memory. However, analytic queries are not random, and we expect them to have certain tendencies. For example, one would expect that boolgen outputs are often fed into column selects. To test this hypothesis, we count the number of connections between each combination of tiles. For this analysis, we include memory as a "tile" as it is one of the communicating blocks in the system. Figures 7-9 indicate how many times a particular source (y-axis) feeds into a particular destination (x-axis) across all of TPC-H. Looking at this data we observe first that most tiles communicate to and from memory so often, that it will be important to properly understand and provision for the Q100 to/from memory bandwidth. Second, tiles do tend to communicate with a subset of each other, validating our hypothesis that the communication was not truly all-to-all. Thirdly, we note that these communication patterns do not vary across the three Q100 designs.

On-chip bandwidth constraints. We envision a NoC like the one implemented on Intel's TeraFlops chip [38]. It is a 2D mesh and can support 80 execution nodes[2]. While specific NoC design is outside the scope of this paper, we want to understand whether such a design can provide the bandwidth required by these queries. To make a conservative estimate, we scaled down TeraFlop's node-to-node 80 GB/s at 4 GHz to the frequency of the Q100, resulting in a conservative Q100 NoC bandwidth of 6.3 GB/s.

Figures 10-12 plot the peak bandwidth for each connection in the same fashion as the earlier connection counts. The cells marked with X are those for which the peak bandwidth at some point, during one or more of the TPC-H query executions, exceed our estimated limit of 6.3 GB/s. In those cases the NoC will slow down the overall query execution. We also note the following. First, that common connections (per Figures 7-9) do not require high bandwidth except for the Appender to Appender connection, which manipulates large volumes of data in a short amount of time. Second, there are a handful of very common, high-bandwidth connections that, if need be, can be fixed with point to point connections at some cost to instruction mapping flexibility, but at some potential energy and throughput savings.

To understand and quantify the performance impact of the Q100 NoC bandwidth, we perform a sensitivity study, sweeping the bandwidth from 5 GB/s to 20 GB/s as shown in Figure 13. The runtime of all queries in all three configurations are normalized to that of the HighPerf design with unlimited NoC bandwidth (IDEAL). We observe that only a handful of queries are sensitive to an imposed NoC bandwidth limit, however, the slowdown for those queries can be as much as 50X, making interconnect throughput a performance bottleneck when limited to 6.3 GB/s.

Off-chip bandwidth constraints. Memory, we have also seen, is a very frequent communicator, acting as a source or

[2] Though the more recent version of the Intel SCC [21] provides higher bandwidth and lower power, we chose TeraFlops because it connects execution units rather than CPU cores, and therefore better resembles the Q100.

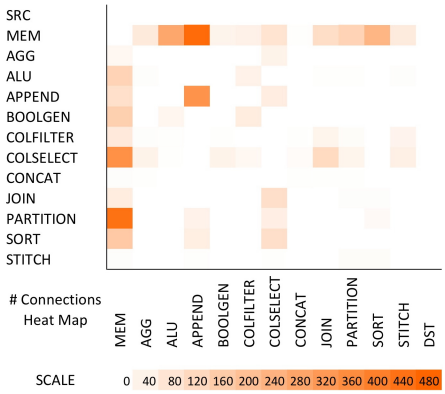


Figure 7. A heat map of tile-to-tile connection counts for the LowPower design shows that most intra-tile connections exist mostly when communicating to and from memory.

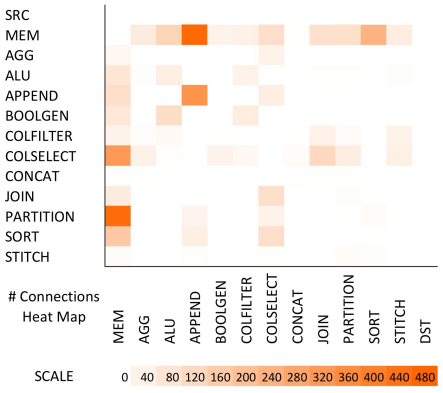


Figure 8. Our Pareto design uses slightly more connections than LowPower design when running the TPC-H suite, but memory is still the busiest communication tile.

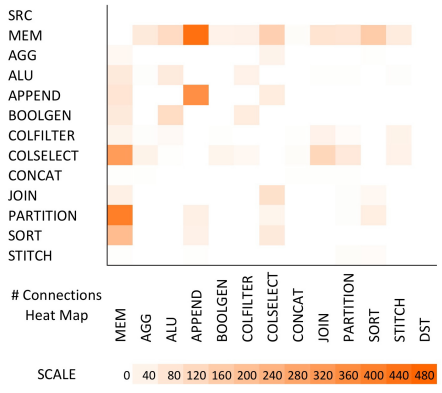


Figure 9. HighPerf design intra-tile heat map exhibits almost identical behavior as Pareto design.

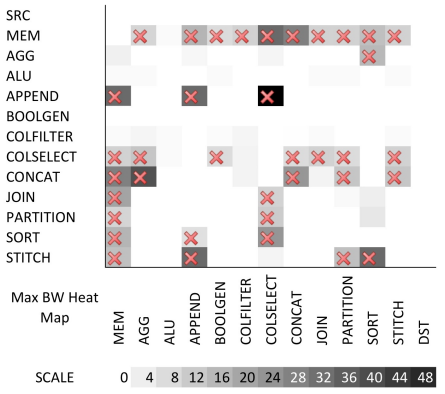


Figure 10. Even with a LowPower design, the communication bandwidth for most connections exceed the provisioned 6.3 GB/s NoC bandwidth, marked as X's in the figures.

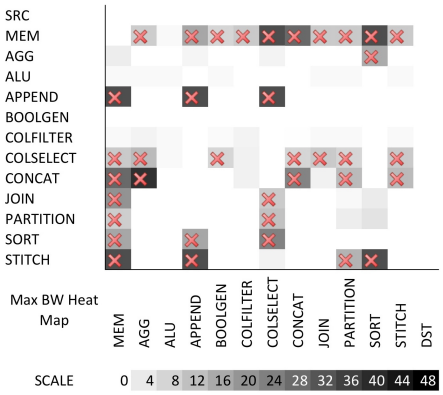


Figure 11. Similar to connection count heat map, Pareto design maximum intra-connection bandwidth exhibit almost identical behavior as HighPerf design.

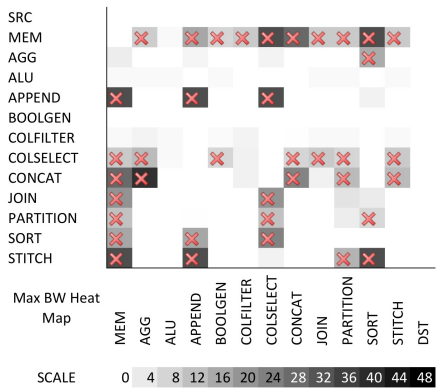


Figure 12. Heat map of HighPerf design max bandwidth per connection.

destination for all types of Q100 tiles. Half of those connections also require high throughput connections. In Figure 14 and Figure 15, we examine the high, low, and average read and write memory bandwidth for each query, sorted by average bandwidth. We first notice that queries vary substantially in their memory read bandwidths but relatively little in their write bandwidths. This is largely due to their being analytic queries, taking in large volumes of data and producing comparatively small results, matching the volcano style [16] of software relational database pipelined execution. Second, queries generally consume more bandwidth as the design becomes higher performance (i.e., going from LowPower to HighPerf), as the faster designs tend to process more data in a smaller period of time. Finally, in the same fashion that we expect the NoC will limit performance, realistic available bandwidth to and from memory is also likely to slow query processing.

Multiple instances of a streaming framework, such as the one described in recent work [39], could feed the Q100 assuming 5 GB/s per stream. At that rate, the Q100 would require 4-6 inbound stream buffers depending on the configuration and 2 outbound stream buffers, reflecting the read/write imbalance noted earlier. The provided bandwidths from these stream buffers are shown in shaded rectangles in the figure.

To quantify the performance impact of memory bandwidth, we perform a sweep of memory read bandwidth from 10 GB/s to 40 GB/s and memory write bandwidth from 5

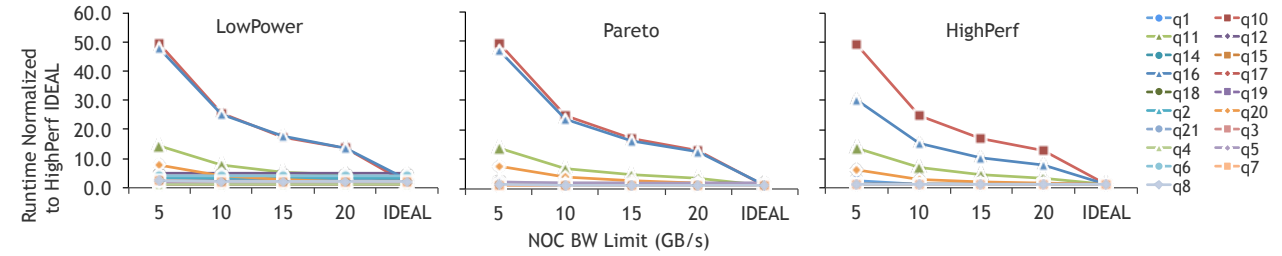


Figure 13. Most TPC-H queries are not sensitive to the Q100 intra-connection throughput, except for Q10, Q16, and Q11. These queries process large volumes of records throughout the query with little local selection conditions to "funnel" down the intermediate results. When NoC bandwidth is constrained, these queries could execute fifty times slower.

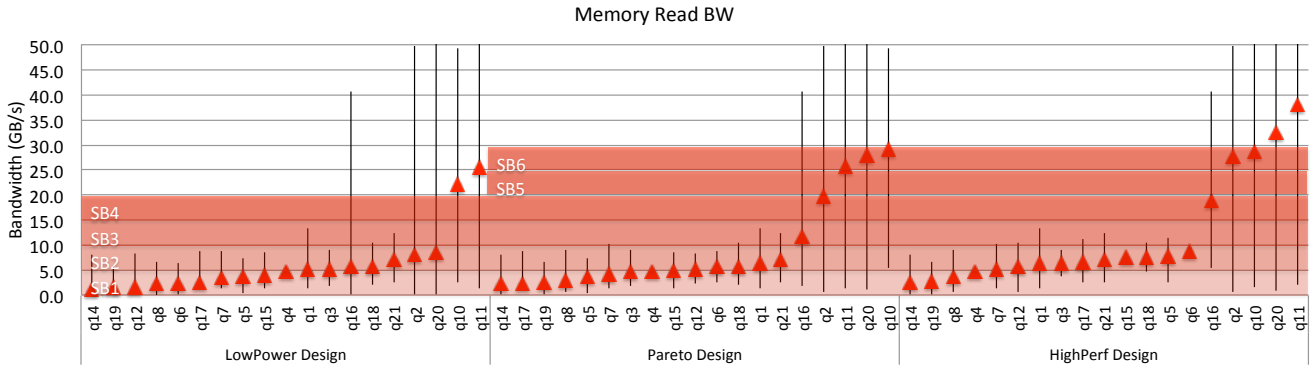


Figure 14. A plot of all TPC-H query read memory bandwidth demands (hi, lo, and avg) sorted by average. Read bandwidth varies quite a bit from query to query, having Q10 and Q11 being the most bandwidth starved. For Q100, LowPower design is provisioned with 4 stream buffers, and Pareto and HighPerf designs are provisioned with 6 stream buffers as shown in shaded gradations.

Figure 15. Write bandwidth demands are quite a bit lower than read bandwidth demands for most queries. We sized all three designs with 2 stream buffers, providing 10 GB/s write bandwidth to memory.

GB/s to 20 GB/s as shown in Figure 16 and Figure 17. As with the NoC study, only 2 or 3 queries are sensitive to memory read and write bandwidth limits, but with much more modest slowdowns.

Performance impact of communication resources. Applying the NoC and memory bandwidth limits discussed above, we simulate a NoC bandwidth cap of 6.3 GB/s, memory read limit of 20 GB/s for LowPower and 30 GB/s for Pareto and HighPerf, and memory write limit of 10 GB/s. Figure 18 shows the impact as each of these limits is applied to an unlimited-bandwidth simulation. On account of on-chip communication, queries slow down 33-61%, with only a slight additional loss on account of memory to 34-62% slowdown overall. These effects are largely due to Q10 and Q11, the two most memory hungry queries, which suffer 1.4X-1.5X slowdown and 6X to 11X slowdown respectively compared to software.

Our simulator models a uniform memory access latency of 160ns, based on a 300 cycle memory access time from a 2 GHz CPU. When the imposed interconnect and memory throughput slow the execution of a spatial and a temporal

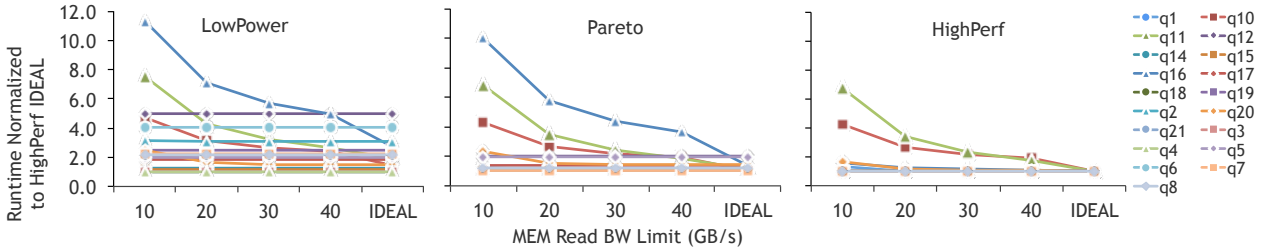


Figure 16. Similar to NoC bandwidth, most queries are not sensitive to memory read bandwidth. Q16 is particularly affected for the LowPower and Pareto designs suffering up to 12X slowdown. However, in the HighPerf design, more resources allow for a more efficient scheduling of temporal instructions, reducing high-volume communications to and from memory.

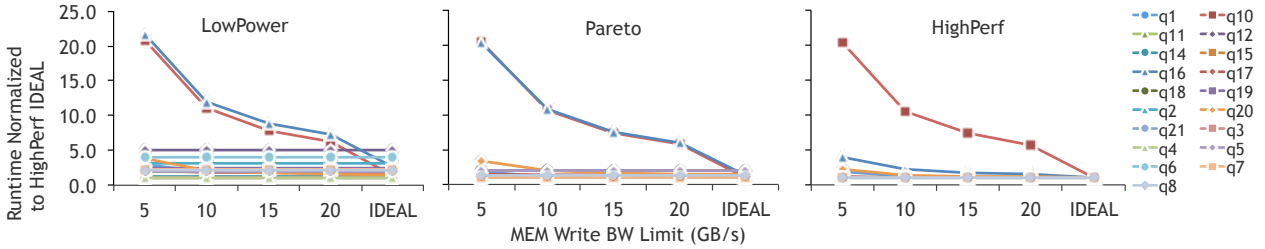


Figure 17. With 10 GB/s of memory write bandwidth, only one (LowPower and Pareto) or two (HighPerf) queries are performance- limited by memory write bandwidth.

	Area					Power				
	Tiles	NoC	SBs	Total	Total	Tiles	NoC	SBs	Total	Total
	mm^2	mm^2	mm^2	mm^2	% Xeon	W	W	W	W	% Xeon
LowPower	1.890	0.567	0.520	2.978	7.0%	0.238	0.071	0.400	0.710	14.2%
Pareto	3.107	0.932	0.780	4.819	11.3%	0.303	0.091	0.600	0.994	19.9%
HighPerf	5.080	1.524	0.780	7.384	17.3%	0.541	0.162	0.600	1.303	26.1%

Table 3. Area and power of the three Q100 configurations, broken down by tile, on chip interconnect, and stream buffers.

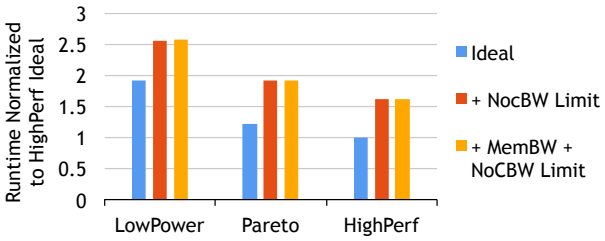


Figure 18. From the bandwidth heat maps plotted earlier, we see that Q100 was demanding a lot more NoC bandwidth than provisioned. Here, we plotted runtime with respect to no bandwidth limit penalties, and see a large slowdown at 30-60%, a caution for future implementations to design sufficient bandwidth for intra-tile connections.

instruction respectively, the simulator reflects that, although we found that throughput was primarily interconnect-limited and thus the visible slowdown beyond that due to memory was negligible. The Q100 reduces total memory accesses relative to software implementations by eliminating many reads and writes of intermediate results. For the remaining memory accesses, the Q100 is able to mask most stalls thanks to heavily parallelized computation that exploits both data and pipeline parallelism.

Area and power impact of communication resources. Starting with the area and power for the tiles in each Q100 design (based on Table 1), we add the additional area and power due to the NoC and stream buffers. Table 3 lists the area of the three design points broken down by tile, NoC, and stream buffers. We add an extra 30% area and power to the Q100 designs for the NoC, based on the characteristics of the TeraFlops implementation [38]. For the stream buffers, we add 0.13 mm^2 and 0.1 $Watts$ for each stream buffer [39]. In sum, the Q100 remains quite small, with the large, High-Perf configuration including NoC and stream buffers taking 17.3% area and 26.1% power of a single Xeon core.

3.4 Query Scheduling Algorithms

The final component of the Q100 design to explore is the scheduling algorithm, by which instructions are mapped to processing tiles.

Problem formulation and experimental algorithms. In the general case, the Q100 has fewer tiles than there are instructions in a query, so the algorithm must schedule them into multiple temporal instructions, subject to the following constraints. An instruction can be scheduled only (1) on a tile that performs its operation, and (2) if and only if all of

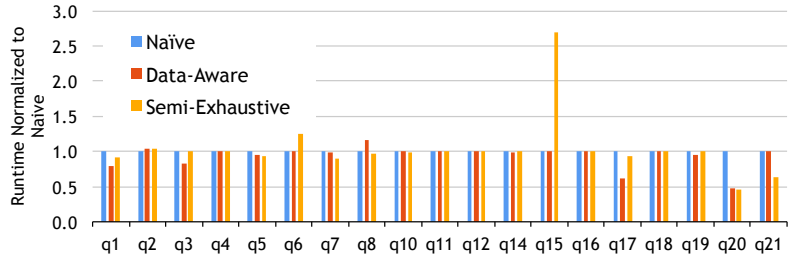


Figure 19. Completion time normalized to naive.

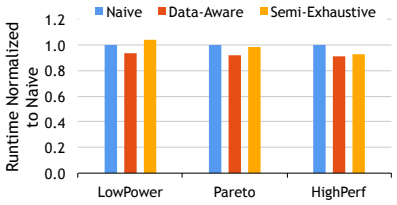


Figure 20. Average completion time normalized to completion time of naive.

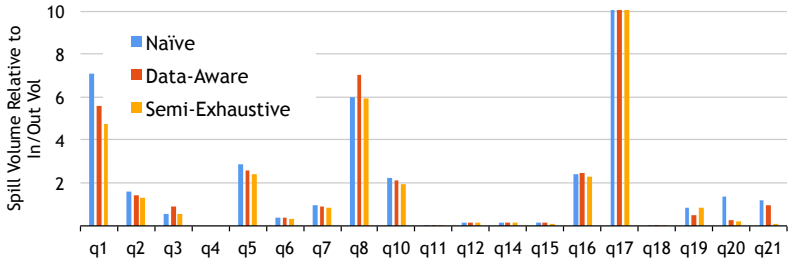


Figure 21. Data transfer size normalized to the volume of input and output data of the query.

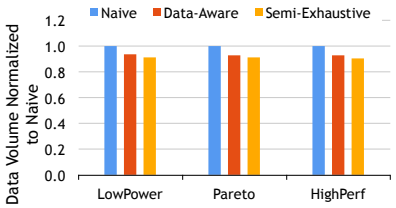


Figure 22. Average data transfer size normalized to size of naive.

its inputs producers have been scheduled in the same or in preceding temporal instructions.

We implemented and evaluated three scheduling algorithms:

- **Naive** greedily packs instructions into the Q100, subject to the constraints listed above, advancing to the next temporal instructions when no more instructions can fit, and stopping when all instructions have been scheduled. In doing this it presumes no knowledge of the volume of data flowing between instructions and therefore makes no effort to minimize data transfer between temporal instructions.
- In contrast, **data-aware** considers the volume of data passed between instructions, not as unrealistic as it may seem, as DBMSs estimate sample or otherwise estimate the rough size of various queries. The information we assume in this study is routinely available at query parse and planning time. It proceeds from largest to smallest data value, greedily attempting to pack all producers and consumers into the same temporal instruction to reduce spills to memory.
- Finally, because both naive and data-aware are greedy and subject to local minima, **semi-exhaustive** searches all legal schedules.[3] Because truly exhaustive search is infeasible – the longer runs are probably still going! – we use a heuristic to prune the search space, making it terminate, but only semi-exhaustive. While not feasible in practice, this algorithm gives us an approximate upper bound for schedule quality.

[3] Because the space of legal Q100 schedules is such a small portion of all Q100 schedules, algorithms that consider non-legal schedules, such as genetic algorithms, are not likely to be efficient for this use.

Analysis of results. We start our analysis on the LowPower Q100 configuration as it has the fewest tiles and is thus most likely to be sensitive to scheduling. We can see in Figures 19 and 21 that, relative to naive, data-aware usually succeeds in reducing the size of memory spills, and that this correlates with a decrease in the completion time.

We also see that, for most queries, semi-exhaustive succeeds in finding schedules with the smallest amount of data transferred amongst these algorithms. The exceptions are Q1, Q17, and Q19, which are so large that the semi-exhaustive approach can only cover a small portion of the search space. Most interestingly, we observe the following pattern. For queries that must spill large volumes of data during execution, the strategy of minimizing data spills is a good one. However, for queries with negligible spills in the first place, attempting to minimize these negligible transfers will cause the scheduler to sub-optimally allocate spatial instructions to temporal instructions. The specific problem we see in Q15, for instance, is the scheduler spreading multiple slow operations across several steps (in an attempt to minimize spills) which ends up increasing the overall completion time of the query. While the data-aware scheduler is sufficient for our current analyses, this indicates a multi-objective heuristic may produce higher quality schedules.

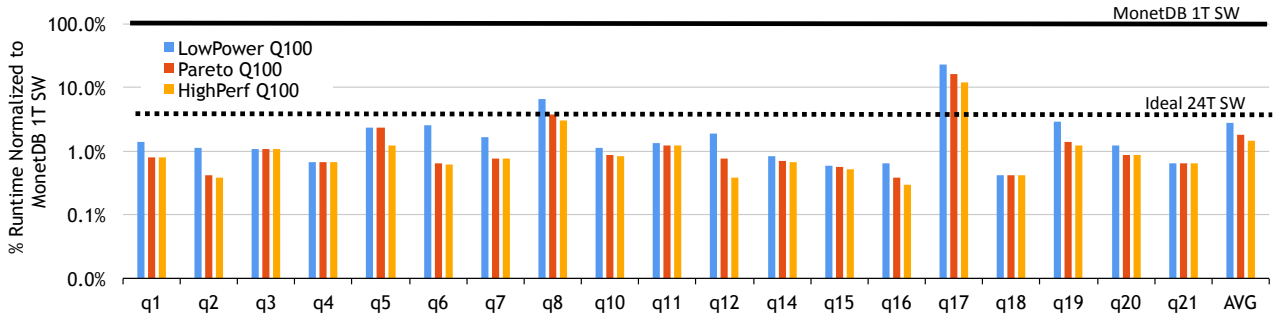


Figure 23. TPC-H query runtime normalized to MonetDB single-thread SW shows a 37X–70X performance improvement on average across all queries.

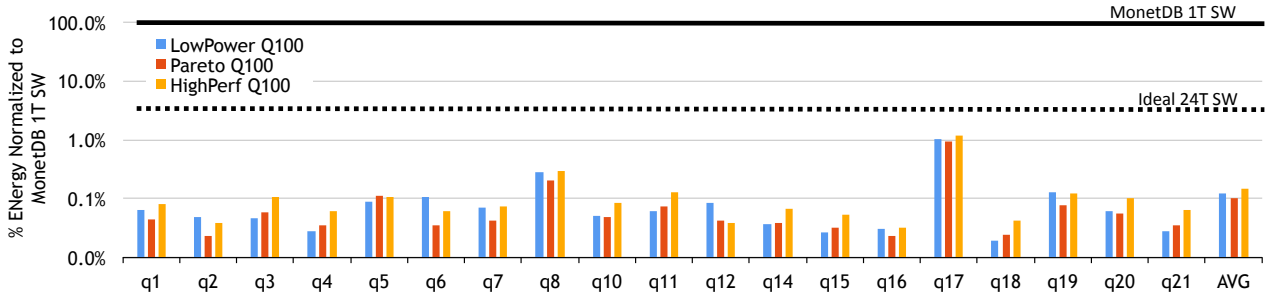


Figure 24. TPC-H query energy consumption normalized to MonetDB single-thread running on cores consuming non-idle power shows 691X–983X energy efficiency on average across all queries.

System Configuration	
Chip	2X Intel E5-2430 6C/12T, 2.2 GHz, 15 MB LLC
Memory	32 GB per chip, 3 Channels, DDR3
Max Memory BW	32 GB/sec per chip
Max TDP	95 $Watts$ per chip
Lithography	32 nm

Table 4. Hardware platform used in software measurements (Section 4. Source: Intel [25].)

4. Q100 Evaluation

Taking what we have learned about the Q100 system, its ISA, its implementation, and its communication both internal and external, we now compare our three configurations, LowPower, Pareto, and HighPerf, with a conventional software DBMS. This evaluation takes on three parts: initial power and performance benchmarking for the TPC-H queries as executed on a conventional DBMS+CPU system, comparison of Q100's execution of TPC-H to that system's, and finally an evaluation of how a Q100 designed for one scale of database handles the same queries over a database 100 times larger.

Methodology. We measure the performance and energy consumption of MonetDB 11.11.5 running on the Xeon server described in Table 4 and executing the set of TPC-H queries. Each reported result is the average of five runs during which we measured the elapsed time and the energy consumption. For the latter we used Intel's Running Average Power Limit (RAPL) energy meters [10, 20] which exposes energy usage estimates to software via model-specific registers. We sample the core energy counters at 10 ms intervals throughout the execution of each TPC-H query. We further deduct any idle "background" power as measured by the same methods on a completely idle machine. The MonetDB energy measurements we report here include only the *additional energy consumption above idle*.

Although MonetDB supports multiple threads, our measurements of power and speedups indicate that individual TPC-H queries do not parallelize well, even for large databases (i.e., 40 GB). Here we will compare the Q100's performance and energy to the measured single threaded values, as well as to an optimistic estimate of a 24-way parallelized software query, one that runs 24 times faster than the single threaded at the same average power as a single software thread. In the upcoming comparisons, we will provide both the MonetDB single-thread SW and MonetDB 24-thread SW (Idealized) as reference points.

For the Q100, we use the full timing and power model, that incorporates the runtime, area, and energy of the on-chip NoC and off-chip memory communication as described in Section 3.3.

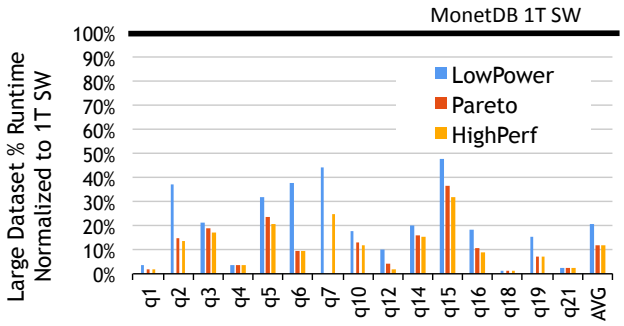


Figure 25. With a dataset that is 100X the size of our previous input tables, TPC-H still shows a 10X performance improvement relative to software on average.

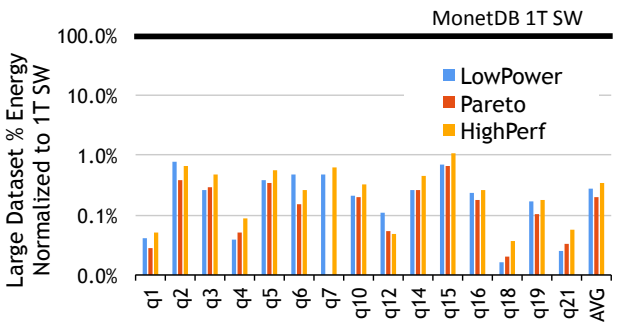


Figure 26. With a 100X larger dataset, the Q100 still consumes $1/100^{th}$ of the energy that software consumes.

Q100 performance comparison. Figure 23 plots the query execution time on the Q100 designs relative to the execution time on single threaded MonetDB. We see that Q100 performance exceeds a single software thread by 37X–70X, and exceeds a perfectly-scaled 24-thread software by 1.5X–2.9X. This is primarily due to Q100's reduced instruction control costs which are a byproduct of the large instruction granularity, where each Q100 instruction does the work of billions (or more, depending on the data size) software instructions. In addition, the Q100 processes many instructions at once, in pipelines and in parallel, generating further speedups. Finally the Q100, having brought some data onto the chip, exploits on-chip communication tile parallelism to perform multiple operations on the data before returning the results to memory, thereby maximizing the work per memory access and hiding the memory latency with computation.

Q100 energy comparison. Fixed function ASICs, which comprise the Q100, are inherently more energy efficient than general purpose processors. Both industry and academia, for example, state that GPUs are 10X-1000X more efficient than multi-core CPUs for well-suited graphics kernels; Similarly, the Q100 is 1400X-2300X more energy efficient than MonetDB when executing the analytic queries for which it was designed. We note that the energy efficiency of our Pareto design is 1.1X better than our LowPower design and 1.6X better than our HighPerf design.

Scaling up data. Finally, as big data continues to grow, we wish to evaluate how the Q100 handles databases that are orders of magnitude larger than the ones for which it was initially developed, we performed the same Q100-MonetDB comparison using 100X larger data. Figure 25 and Figure 26 show the results. With the input data having grown by 100X, Q100 speedup over software drops from 100X to 10X. The total energy remains 100X lower regardless of data size.

5. Related Work

Hardware acceleration of databases. Database machines were developed by the database community in the early 1980s as specialized hardware for database workloads. These efforts largely failed, primarily because commodity CPUs were improving so rapidly at the time, and hardware design was slow and expensive [7]. The architectures proposed at that time targeted a different set of challenges than those we face today, namely dark silicon, the utilization wall, the power wall, etc.

Much more recently, a flurry of projects accelerates queries by compiling them down to FPGAs, such as LINQits [9], Teradata [11], and [29]. Industry appliances using Xeon servers combined with FPGAs such as the IBM Netezza [22] also show promising performance and energy efficiency. Whereas we have designed a *domain specific circuit*, these projects produce *query-specific circuits*, a different point in the specialization space. Other have investigated using existing accelerators, such as network processors [14] or GPUs [15] to speed relational operators. Our work is similar in that we too accelerate database queries and relational operators, but differs in the overall strategy and specific hardware platform.

Streaming computation. StreamIt [37] is a programming language and a compilation infrastructure supporting parallel execution of stream-based applications. It relates to our work in as much as Q100 processes database queries in streams of data, forming direct communication between producing and consuming kernels. However, whereas we target database-specific, hardware kernels, StreamIt supports user-defined software kernels, from other domains including digital signal processing, multimedia, and cellular communications. It is worth noting that the Q100 does not process streams in the most general sense of the term. Our ISA and implementation operate on relational streams (i.e., where

multiple streams have corresponding elements) which is a specific type of stream.

Domain-specific accelerators. The Q100 is a domain specific processor, of which there are many others targeting different domains. GPUs are perhaps the most visible and among the most successful such processors targeting graphics applications [5, 30]. There is a large body of research around other domain-specific acceleration: Convolution Engine [32] targets image processing kernels and stencil computations, [33] uses specialization to speed up regular expression matching in queries, and [18] accelerates H.264 video encoders. In spirit these projects share similarities with the Q100, but in their design and target particulars they are quite different.

6. Conclusion

As data quantities continue to explode, technology must keep pace. To mitigate commensurate increases in time and energy required to process this data with conventional DBMSs running on general purpose CPUs, this paper has presented the Q100, a DPU for analytic query workloads.

With the Q100, we have presented an instruction set architecture which closely resembles common SQL operators, together with a set of specialized hardware modules implementing these operators. The Q100 demonstrates a significant performance gain over optimistically scaled multithreaded software, and an order of magnitude gain over single threaded software, for less than 15% the area and power of a Xeon core at the evaluated configurations. Importantly, as inputs scale by 100X, the Q100 sees only a single order of magnitude drop in performance and negligible decrease in energy efficiency. Given the current gap between the Q100 and standard software query processing, as well as the growth rate in data volumes, it is clear that specialized hardware like the Q100 is the only way systems will be able to keep pace with increases in data without sacrificing energy efficiency.

7. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1065338. The infrastructure was partly supported by resources of Sethumadhavan's Comp Arch and Security Technology Lab (CASTL) which is funded through grants CNS/TC 1054844, FA 99500910389, FA 865011C7190, FA 87501020253 and gifts from Microsoft Research, WindRiver Corp, Xilinx Inc. and Synopsys Inc.. The authors also wish to thank Yunsung Kim, Stephen Edwards, and the anonymous reviewers for their time and feedback.

References

[1] Kx systems. `http://kx.com/_papers/Kx_White_Paper-2013-02c.pdf`.

[2] Sybase IQ. `http://www.sybase.com/products/archivedproducts/sybaseiq`.

[3] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *VLDB*, August 2009.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, 2007.

[5] AMD/ATI. `http://www.amd.com`.

[6] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[7] Haran Boral and David J. DeWitt. Database machines: an idea whose time has passed? In *IWDM*, 1983.

[8] Centrum Wiskunde and Informatica. `http://www.monetdb.org`.

[9] E. S. Chung, J. D. Davis, and J. Lee. Linqits: Big data on little clients. In *ISCA*, 2013.

[10] Intel Corporation. Intel 64®and IA-32 architectures software developer's manual. `http://download.intel.com/products/processor/manual/253669.pdf`.

[11] Teradata Corporation. `http://www.teradata.com`.

[12] J. B. Dennis. *Advanced topics in data-flow computing*. Prentice-Hall, 1991.

[13] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the TRIPS computer system. In *ASPLOS*, 2009.

[14] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *DaMoN*, 2005.

[15] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGGRAPH*, 2005.

[16] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensivility and efficient search. In *ICDE*, 1993.

[17] J.R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 1985.

[18] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[19] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance studies of ld on the monsoon dataflow system. 1993.

[20] D. Howard, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: memory power estimateion and capping. In *ISLPED*, 2010.

[21] J. Howard, S. Dighe, Y. Hoskote, S. R. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. F. Van der Wijngaart, and T. G. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.

[22] IBM. IBM Netezza Data Warehouse Appliance. http://www-01.ibm.com/software/data/netezza/.

[23] IDC Research. IDC's most recent worldwide Big Data technology and services market forecast. http://www.idc.com/getdoc.jsp?containerId=prUS23355112.

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

[25] Intel Corporation. Intel Xeon Processor E5-2430, 2012. http://ark.intel.com/products/64616/Intel-Xeon-Processor-E5-2430--(15M-Cache-2_20-GHz-7_20-GTs-Intel-QPI).

[26] M. F. Ionescu and K. E. Schauser. Optimizing parallel bitonic sort. In *IPDPS*, 1997.

[27] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. In *VLDB*, 2012.

[28] A. McAfee and E. Brynjolfsson. Big Data: The management revolution. *Harvard Business Review*, October 2012.

[29] R. Muller and J. Teubner. FPGAs: A new point in the database design space, 2010. EDBT Tutorial.

[30] NVIDIA. http://www.nvidia.com.

[31] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, and J. Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. In *ISCA*, 2013.

[32] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. In *ISCA*, 2013.

[33] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *HPCA*, 2012.

[34] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*, 2005.

[35] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Trans. Comp. Syst.*, 2007.

[36] Synopsys, Inc. 32/28nm Generic Library for IC Design, Design Compiler, IC Compiler. http://www.synopsys.com.

[37] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, 2010.

[38] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *ISSCC*, February 2007.

[39] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, 2013.

[40] M. Zukowski and P. Boncz. Vectorwise: Beyond column stores. *Data Engineering Bulletin*, 2012.