# Deadlock-Free Joins in DB-Mesh, an Asynchronous Systolic Array Accelerator

Bingyi Cao
Columbia University
bingyi@cs.columbia.edu

Kenneth A. Ross
Columbia University
kar@cs.columbia.edu

Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Martha A. Kim
Columbia University
martha@cs.columbia.edu

## ABSTRACT

Previous database accelerator proposals such as the Q100 provide a fixed set of database operators, chosen to support a target query workload. Some queries may not be well-supported by a fixed accelerator, typically because they need more resources/operators of a particular kind than the accelerator provides. By Amdahl's law, these queries become relatively more expensive as they are not fully accelerated. We propose a second-level accelerator, DB-Mesh, to take up some of this workload. DB-Mesh is an asynchronous systolic array that is more generic than the Q100, and can be configured to run a variety of operators with configurable parameters such as record widths. We demonstrate DB-Mesh applied to nested loops joins, an operator that is not directly supported on the Q100. We show that a naïve implementation has the potential for deadlock, and show how to avoid deadlock with a careful design. We also demonstrate how the data flow policy used in the array influences system throughput.

**ACM Reference format:**
Bingyi Cao, Kenneth A. Ross, Stephen A. Edwards, and Martha A. Kim. 2017. Deadlock-Free Joins in DB-Mesh, an Asynchronous Systolic Array Accelerator. In *Proceedings of DaMoN'17, Chicago, IL, USA, May 15, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3076113.3076118

## 1 INTRODUCTION

Single-threaded processors hit their performance wall about a decade ago; it has become clear that application accelerators—compute hardware customized for a particular problem domain—are the way to further improve performance and reduce power consumption.

In prior work, we envisioned a data processing unit (DPU) that would do for data analytics what graphics processing units (GPUs) do for image processing. We designed the Q100 [11, 12], a DPU that targets query processing. The Chinese search giant, Baidu, announced plans for a data analytic accelerator that shares many features with the Q100 [6]. One of the most exciting results of our evaluation is that the Q100's energy efficiency gains relative to software are insensitive to the size of the database. This is a significant practical result within the context of the ever-increasing demands of big data processing.

The Q100 obtains these gains by chaining together multiple fixed-function streaming accelerators for relational operations, amortizing the expensive data read across multiple operations including aggregations, joins, and sorts. Moreover, the operations are happening in parallel, with multiple records in a single table processed in a pipeline, and multiple tables processed in separate parallel instances of the accelerators. Our design exploits the natural column-oriented structure in the workload, allowing the Q100 to more efficiently move and manipulate database content.

Engineering a single accelerator to improve a CPU-based system's performance involves a difficult tradeoff: make it too specialized and it will only work for a small fraction of the workload, which will remain dominated by the slower, power-hungry general-purpose CPU. Make the accelerator too general and it is unlikely to outperform the general-purpose processor, nullifying any advantage.

In this context, having multiple accelerators for a single workload makes sense. Amdahl's law justifies our approach: adding a modest accelerator able to cover only a small additional fraction of a workload—something fairly easy to design—can greatly improve overall system performance.

## 2 Q100 BACKGROUND

Q100 [12] is a database accelerator design that implements heterogeneous ASIC tiles to process relational operators. It achieves high performance and energy efficiency. Q100 implements various function tiles, including *sorter, partitioner, (merge-)joiner, ALU, boolean-generator, column filter, and aggregator*. For each query, Q100 maps it onto available corresponding tiles, where the output from one tile can be consumed as the input of another in a pipelined fashion. Multiple instances of a tile can be included in any candidate design. The mix of tiles used for the Q100 design was informed both by analytic target workloads and by area/power measurements for the corresponding tiles. Q100 aims to off-load a significant portion of database workloads to improve both performance and energy efficiency.

## 3 DB-MESH

The accelerator we propose, dubbed DB-Mesh, is a *configurable fabric* of processing elements specialized for database query processing (Figure 1). Unlike the Q100 [12], the processing elements in this fabric are uniform but configurable: each will be able to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*DaMoN'17, Chicago, IL, USA*
© 2017 ACM. 978-1-4503-5025-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3076113.3076118

perform any of a number of different database operations. Because the operators of the Q100 are fixed-function only, inevitably some queries will need more of some type of operator than is built in to the Q100. Addressing this is a key goal of the DB-Mesh design: enabling computational resources to be deployed in a more flexible manner than the Q100. All core Q100 operators can be mapped to DB-Mesh.

Occasionally, queries require less common operators. For example, queries with a join condition that is not an equality comparison require the system to compare all pairs of records [9]; in software this would be called a nested-loops join [8]. This is a key aspect of our design, and motivates the analysis of deadlock in this paper.

Finally, some queries may stretch the resources of the core Q100-based component. For example, a core join operator may be designed to handle up to $c$ columns; joins of tables with additional columns would need multiple passes through the data, slowing down execution. In DB-Mesh, we have the opportunity to be more flexible. Once we know the number of columns needed at query time, we can configure the fabric and make just one pass through the data.

Our proposed architecture for the configurable fabric is a two-dimensional array of processing elements (Figure 1). Each processing element contains some basic functionality, such as registers for buffering, comparators for performing condition tests, counters for keeping track of multi-packet records, and circuitry for receiving/-transmitting data to other processing elements. A small amount of configuration logic determines how the processing element manipulates the data (e.g., what part of the record is extracted, and what kind of comparison is performed with a local state register) and transmits the data (e.g., one output port for matches, all output ports for matches, etc.).

Our choice of asynchronous communication framework allows for a robust response to unpredictable downstream delays. In a synchronous design, any delay would force the whole circuit to halt for a cycle. In an asynchronous design, backpressure can specifically stall certain paths, while other paths operate at full capacity.

A high-level picture of the array is shown in Figure 1. Data flows from contiguous blocks of memory into the array from the left. The start, end, and current addresses of each of the $W$ blocks are maintained by the memory controller, and memory is read in a streaming fashion using a suitable batching granularity (e.g., 64 bytes). (In some operator configurations, not all of the $W$ input lanes will be used.) Information flows through the array from left-to-right

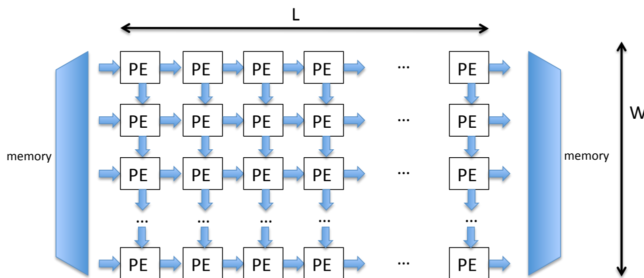


**Figure 1: Our proposed DB-Mesh accelerator: a systolic array of processing elements**

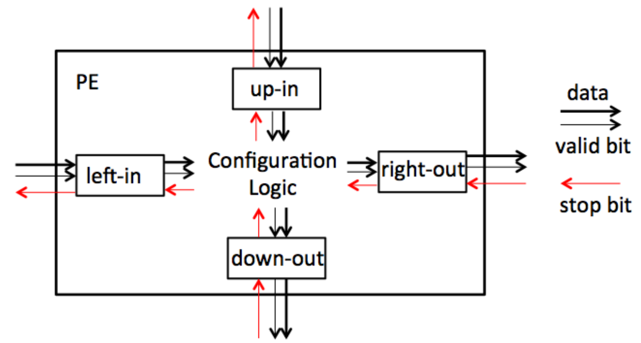


**Figure 2: Four-buffer structured processing element design**

and/or top-to-bottom, depending on the operator being executed. When data reaches the right side of the array, it is batched into 64 byte units and written to memory.

A horizontal chain of processing elements can be seen as a simple linear pipeline. Thus, without using any of the vertical transmission links, one could implement $W$ parallel pipelines. For example, the Q100 design includes a linear sorting pipeline that generates sorted runs of length equal to the length of the pipeline. The array of Figure 1 could support $W$ parallel sorters each generating runs of length $L$. Alternatively, the array could support $n$ parallel sorters, and use the other $W-n$ contiguous rows for some other operator(s).

To configure the processing elements in the DB-Mesh design, we envisage a separate control plane that may operate relatively slowly, in serial rather than in parallel. Since the vast majority of the time taken by the accelerator will be in data processing rather than configuration, we prefer to separate the two functions and heavily optimize just the data processing component.

## 3.1 Element Design

The processing element design is based on our recent MEMOCODE publication [1]. That design provides provably correct asynchronous communication on multiple input/output channels, which can be tricky in cases where one output channel is consumed by the downstream operator while another output channel is blocked. Flow control is needed because the transmitter may not have the next token ready or the receiver may not yet be willing to consume a token. Buffering also breaks long combinational paths to enable higher clock frequencies.

As shown in Figure 2, a channel consists of data, a valid bit that indicates data is present and must not be dropped, and a stop signal indicating backpressure. A pair of buffers speak this protocol on both their inputs and outputs and input sequences are provably preserved [1]. Figure 2 shows a two-input, two-output processing element. Since the information flows through DB-Mesh from left-to-right and/or top-to-bottom, two input buffers are placed on the left and top edges of the processing element, while two output buffers are placed on the right and down edges. The configuration logic shown in the center of Figure 2 determines how to manipulate and/or route the data tokens in the input buffers. For example, the configuration logic may determine whether to advance data tokens from the input buffers to the output buffers, and if so, where and

when to advance the data tokens. The processing element can be configured in various ways to perform different database operations. Two common configuration patterns are:

**Simple Transfer.** the processing element just transfers the data token from an input buffer to one of the corresponding output buffers. There are four possible ways in which data can flow: left-to-right, top-to-botton, left-to-bottom, and top-to-right. Two data flows are allowed in the same clock cycle if neither the source buffers nor the destination buffers conflict.

**Broadcast.** the processing element broadcasts the input data token to both output buffers, which increases the number of data tokens. DB-Mesh adopts a strict firing rule that allows a broadcast to proceed only when both output buffers are available to consume data token.

Global behavior in DB-Mesh occurs in two non-overlapping phases. In the first phase, data tokens are transferred between processing elements: from upstream output buffers to downstream input buffers. In the second phase, the data flows internally within each processing element from input buffers to output buffers, based on the configuration logic for each element. Processing elements typically behave identically in the first phase but differ in the second due to different configurations in different parts of the mesh.

## 3.2 All-to-All Joins

Consider a more complex operation: a nested-loops style join between two tables $R$ and $S$. Suppose we place $L$ records of $S$ in the processing elements across the top row of the array. We then stream the rows of $R$ through the top input from memory and across the top row. In each step, a top-row processing element will compare the current $R$ record against the current $S$ record. If they match, a join record (containing a subset of the columns from $R$ and $S$) will be generated and transmitted *downwards*. This second row represents the pipeline of join results; those results flow from left to right until they reach memory. Note the importance of allowing elements to send data to two output ports: the top row both sends input records to the right and join results down. If we are happy with this design, we could replicate it at multiple levels of the array, distribute the input using the first column of the array, and process roughly $LW/2$ records from $S$ in one pass.

There is the potential for contention in the two-row design because a join result moving down may compete with a join result flowing right into the same processing element. We may choose to give priority to the vertical flow, and apply backpressure to the horizontal flow. This backpressure may propagate, and temporarily prevent upstream processing elements from making progress. In situations where we expect the join result to be much bigger than the input (i.e., a many-to-many join) the second join-result row will be the bottleneck. This observation leads to alternative designs where (based on cardinality estimates from the query optimizer) we estimate that the join result is $d$ times bigger than the input, and therefore allocate $d$ rows of the array for transporting the join results. Collisions in all "transport" rows except the last can usually be resolved by sending one join result down and the other across. This solution is an improvement on the single transport row approach because $d/(d+1)$ of the output ports, rather than $1/2$ of them, are fully utilized.

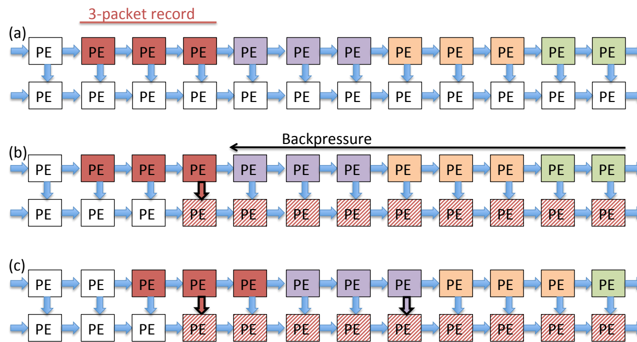


**Figure 3: Deadlock with long records.**

## 3.3 Multi-Word Data

So far, we have assumed that a packet of data contains a complete record from a table. In practice, the data links in our array will not have fixed width, and some tables will have wider records than what one packet can accommodate. In such a situation, we generalize the data flow abstraction so that a record is now a *sequence of packets* flowing through the array. To support this abstraction, processing elements need to be able to keep track of where in a record they are up to, using an internal counter. Additionally, when dealing with contention, a processing element must make sure that an entire record has made it through before packets from another record use the same path. We have built an initial simulator for this model, and can visualize the records as "worms" winding their way through the array.

One appealing aspect of the design of Figure 1 is the absence of cycles in the flow path. As a result, when sending single packets containing records through the array, the system would be deadlock-free. However, this guarantee does not hold when records can span multiple packets. Our simulations show that deadlock can, in fact occur, as illustrated in Figure 3. Figure 3(a) shows records, each consisting of three packets, flowing along the top row of processing elements. In Figure 3(b), backpressure causes progress in the top row to stall. However, the join test in the fourth processing element detects a match and sends the first packet of the join result along the second row of the array. Subsequent join result packets cannot proceed immediately due to the backpressure. In Figure 3(c), the backpressure has eased, and records in the top row flow one cell to the right. Another join match is detected in the eighth processing element, downstream of the original match. The join result has no place to go, because the record that has been partially transmitted in the second row cannot be interrupted. As a result, the top row is blocked, preventing the tail of the original joining record from reaching the fourth processing element, and inducing a deadlock.

# 4 RESOLVING DEADLOCKS IN DB-MESH

There are various ways of dealing with deadlock, by imposing particular configurations or protocols on the mesh. For example, if every path from a join-result generating node to an output port is disjoint from every other, then no interference can occur. However, such a design would be inefficient, using many more processing elements and output ports than needed in the average case. In this section, we

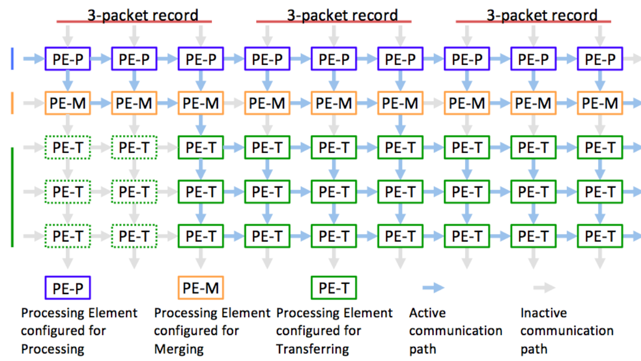


**Figure 4: DB-Mesh for Joins of Long Records**

describe a mapping of an all-to-all join operation to DB-Mesh that is provably free from deadlock, even when records contain multiple packets of data.

## 4.1 Configuring DB-Mesh for Joins of Long Records

The primary reason that deadlock occurs in Figure 3 is that the packets of a record get separated, leading to reserved regions in the mesh that block progress. Our design aims to avoid this problem by preparing complete join records using the first two rows of the mesh, before transmitting the join record though the network. Once a join result starts to flow towards an output port, it can do so without risk of fragmentation. Figure 4 illustrates the design for a join of two tables each containing 3-packet records. (When the inputs have different record lengths, we allocate space based on the longer length.)

The top row is called the processing chain, and it contains elements called "PE-Ps". Groups of three PE-Ps coordinate to store records from the build table. The leftmost of the three stores the packet from the build-table record containing the join key. The probe records flow from left to right along the processing chain after being read from memory. We assume that all packets of a record have been read from memory before any packets flow into the processing chain. The first (rightmost) packet of a probe record contains the join key, so that the first time packets from the two records meet, the PE-P can determine if they join. A bit is set in this leading probe packet so that all PE-Ps handling a build record eventually learn whether a join result should generated.

Once all three probe record packets line up with the three build table packets, a join result is generated if the earlier comparison was successful. This join result generation is achieved by transmitting first the probe and then the build packets down to the second row of the mesh in an interleaved fashion. This second row is called the merging chain, and contains elements known as "PE-Ms". The rightmost PE-M in a group sends the data down into the transfer network (note the shading of the arrows in Figure 4). The other PE-Ms transmit their local data rightwards, and then transmit incoming data from the left further to the right. PE-Ms have enough buffering capacity to hold both a build and a probe packet. Probe table packets also continue to flow along the processing chain, unless there is backpressure. Backpressure could come from the right, or from below if a join result needs to be generated but the merging chain is congested.

The third and subsequent rows are the transfer network, made up of elements called "PE-Ts". The flow logic depends on a transfer protocol to decide whether to send a join record right or down. Any protocol must ensure that once a join record starts flowing in one direction, the remainder of the record flows in the same direction and is not interrupted. Figure 4 includes one small optimization in which the rightmost PE-M is connected directly to an output port rather than sending its data down.

All PE types have counters that are configured to wrap after the appropriate number of packets (input or output table lengths), so that each PE can keep track of which packet of a record it is up to, avoid interleaving packets from different output records, and make sure that the join result is generated in a consistent packet order.

## 4.2 Freedom from Deadlock

In Appendix A we provide the proof that DB-Mesh is deadlock-free for joins of multi-word data when configured as described in Section 4.1. The outline of the proof is as follows. First, we show that probe table records stay together in the processing chain. This is important so that join results can be generated "all at once" without waiting for a straggler packet. We then show that the merging chain and transfer chain are able to generate and transmit join records without blocking indefinitely. Together with the acyclicity of DB-Mesh, freedom from deadlock follows.

# 5 TRANSFER PROTOCOLS

Once join records are generated, they flow through the lower layers of DB-Mesh towards the output ports at the right end of the mesh. The protocol governing this flow can impact performance, as we will demonstrate in this section. Raw performance can be measured as the output bandwidth, i.e., the number of output packets generated per cycle. Efficiency is measured by dividing the output bandwidth by the number of rows of DB-Mesh used, including rows devoted to processing, merge and transfer (Figure 4). Efficiency is an important measure because DB-Mesh can be split into multiple contiguous subregions to be used for different operators, meaning that over-provisioning for one operator reduces resources available for other concurrent operators.

## 5.1 Join Model

We subdivide the $L \times W$ mesh (Figure 1) into multiple separate join pipelines of length $L$ as outlined in Section 3.2. The overall capacity of DB-Mesh for build-table records is proportional to $L \times W$. Increasing the number of output ports $d$ per pipeline will yield more output bandwidth for an individual pipeline, but will decrease the number of pipelines that can be run simultaneously. If a build table is larger than the DB-Mesh capacity, then fragments of the build table would be processed in turn, analogously to a block nested loops join. Probe table records are read and sent independently in parallel along each of the join pipelines.

In what follows, we simulate a single join pipeline, varying $d$, and focus on the interesting case where there are many matches for each probe tuple.

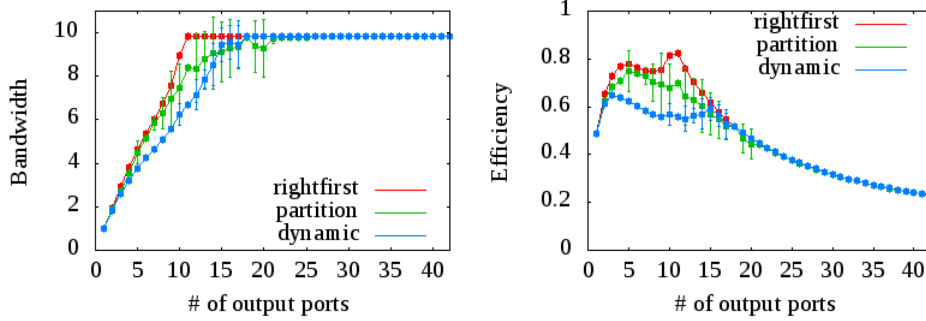


**Figure 5: Throughput/Efficiency evaluation for single-packet all-to-all join with uniform distribution**

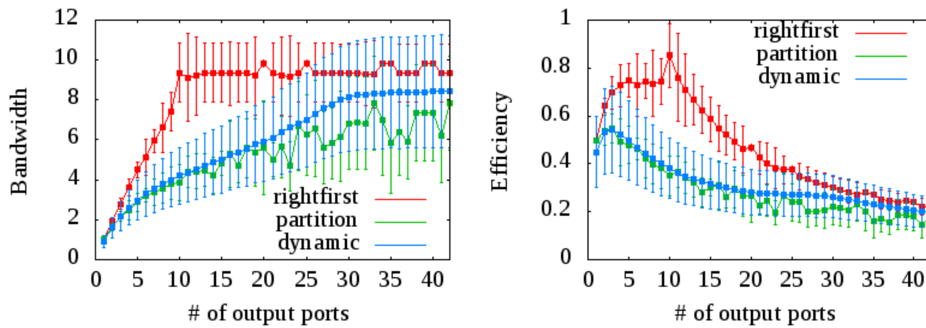


**Figure 6: Throughput/Efficiency evaluation for single-packet all-to-all join with non-uniform distribution**

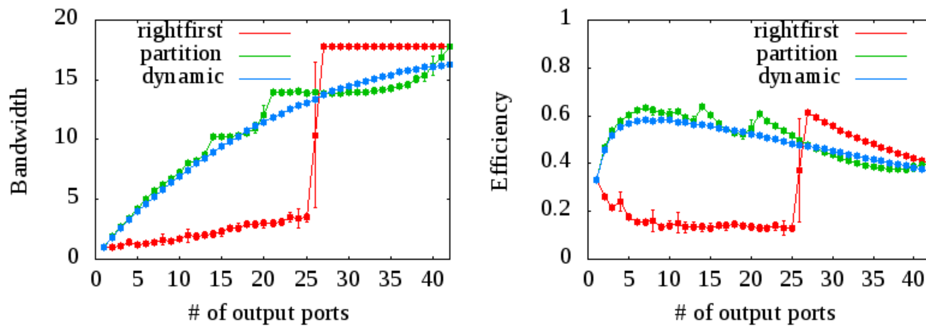


**Figure 7: Throughput/Efficiency evaluation for 3-packets all-to-all join with uniform distribution**

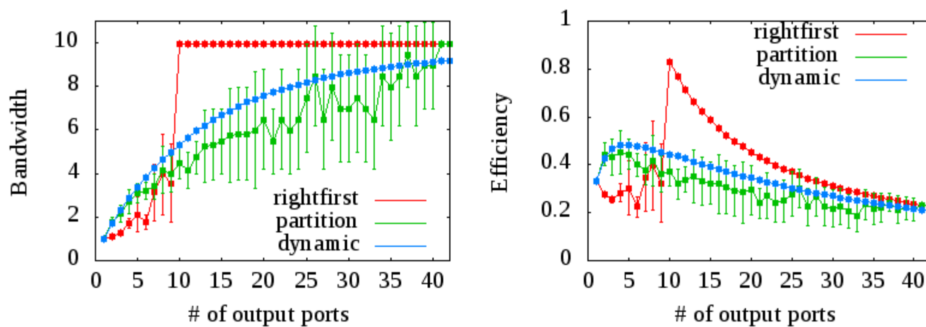


**Figure 8: Throughput/Efficiency evaluation for 3-packets all-to-all join with non-uniform distribution**

## 5.2 Performance

We investigate the performance by simulating a join consisting of a small single-pipeline "build" relation and a large "probe" relation. The build relation is loaded into the processing elements of the top row of DB-Mesh. We execute a join corresponding to a range predicate, and manipulate the range endpoints to generate a particular output cardinality. For example, if a probe record matches four build records, on average, then the join output from this join pipeline will have four times as many records as the probe input. For these experiments, we assume $L = 128$ (Figure 1) and vary the number of rows dedicated to the join pipeline. Some experiments use random assignments of keys to nodes, which may influence the results. We therefore perform ten runs of each configuration with different randomizations, and show both the mean and standard deviation in the graphs.

We consider three transfer protocols here:

- Right-First: Send data right if the right buffer is available; otherwise send data down (unless this is the bottom row).
- Partition: Statically map each top-row processing element to an output row in a round-robin fashion. Each transfer row is then responsible for a roughly equal proportion of the join results.
- Dynamic: Processing elements in each column of the transfer unit are independently and randomly numbered from 1 to $r$, where there are $r$ transfer rows. The top row processing element sends output records to successive "logical" rows 1 to $r$ in sequence. Because each column is independently permuted, the physical rows are randomized and uncorrelated across columns.

Our first experiment considers a join in which both input and output records contain one data packet. The join is configured so that there are ten matches in the pipeline, on average, for each probe record, so that the output result is ten times larger than the input. Figure 5 shows the results when the matches occur uniformly among the 128 top-row processing elements. Figure 6 shows the results when the matches occur in a highly biased fashion, where ten of the elements in the top row generate all the matches, and the remaining elements generate no matches. For uniform data, all transfer protocols perform reasonably well. For nonuniform matching, the Right-First protocol does best because it is somewhat responsive to the current congestion conditions, and can fully saturate the transfer network as soon as it is provisioned with enough output ports (10) to match the output and input bandwidth bottlenecks. Partitioning and randomization do not do so well here because at any given moment of time, some output rows are over-utilized and some are underutilized. For partitioning this is due to one partition having more that its share of data generating elements. For the dynamic approach, this is due to the fact that randomized data is not perfectly uniform. The variance is also higher for the non-uniform case.

Our second experiment considers a join in which both input table records contain three data packets, and join results contain six packets. The join is configured so that there are ten matches in the pipeline, on average, for each probe record, so that the output result is twenty times larger than the input (measured in packets). The top-row processing elements have capacity to hold 42 build-table records. Figure 7 shows the results when the matches occur uniformly among the build table rows. Figure 8 shows the results when the matches occur in a highly biased fashion, where ten of the elements in the top row generate all the matches, and the remaining elements generate no matches.

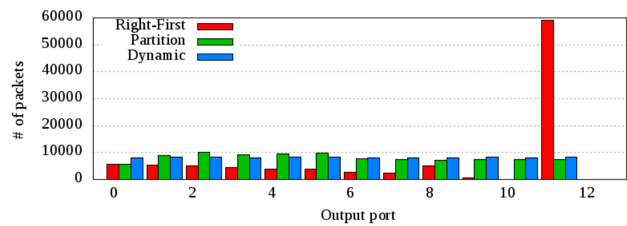


**Figure 9: Join record distribution to output ports**

Unlike the single-packet case, the Right-First protocol performs poorly when the output is under-provisioned. The reason for this behavior is that records in a Right-First protocol can shift between rows, with two problematic consequences. First, stalled records can block other records in multiple rows; the other two protocols choose a single row and can therefore only hold up one row. Second, changing lanes happens in just one direction (down), meaning that there is a concentration of traffic on the lowest row. Figure 9 illustrates the bias in the distribution of packets to the various output ports for the configuration of Figure 7 when the number of output ports is 12.

The dynamic method slightly outperforms the partitioned approach in the non-uniform case because all output rows are used, unlike the partitioned approach that uses at most ten rows. If the partition approach maps several active processing elements to one output row, that row may become the bottleneck. Note also that the non-uniform example has a bottleneck in the transfer of data from the merge chain to the transfer network. This bottleneck potentially limits the output bandwidth below the capacity of the transfer network.

For a single-packet workload, or for a well-provisioned multi-packet workload, the Right-First protocol is the clear winner. The level of provisioning can be estimated at query optimization time based on join selectivities. If an under-provisioned multi-packet workload is needed, the transfer network should be configured with the dynamic protocol.

For these workloads, where the output bandwidth dominates, efficiency is best with a non-minimal number of output ports. In other words, it may better for the aggregate output bandwidth to provision fewer pipelines each having a larger $d$.

# 6 RELATED WORK

The future of computing lies in parallel processing since power dissipation and energy consumption, instead of integration levels, are now the limiting factor. The industry has responded in many ways including arrays of general-purpose processors such as Intel's experimental 48-core Single-Chip Cloud Computer and Tilera's announced 72-core TILE-Gx. Like DB-Mesh, these chips also sport high-bandwidth mesh networks, but their approach remains rooted in general-purpose computing. By contrast, the "cores" proposed in DB-Mesh are much simpler, more specialized, and should consume far less area and power. While we do not yet have physical numbers for DB-Mesh, we are confident that when a database workload

can run on DB-Mesh, it will consume far less power than general-purpose alternatives.

At the other extreme of parallel processor complexity, Kung and Lohman [2] propose systolic arrays—regular two-dimensional arrays of very simple, specialized processors—for database operations. Superficially, such arrays bear a strong resemblance to DB-Mesh, but their details differ substantially. Systolic arrays are synchronous: every piece of data in the array moves one step per clock cycle. Orchestrating these uniform, global flows is the key challenge in programming a systolic array. By contrast, DB-Mesh is conceptually asynchronous to accommodate varying rates, say, at the memory ports, and can handle dynamically routed data to provide a throughput improvement by taking advantage of data-dependent behavior. Kung and Lohman were forced to take such a rigid approach by integrations levels in 1980, where they estimated a single chip with extremely simple processors could only operate on at most 1000 bits. Today, that number would be roughly 100 million, allowing us to put far more storage throughout the array and consider devoting only a small fraction of a high-end chip to a custom database processor.

While FPGAs offer the ultimate in configurability [3, 4, 7, 9, 10], our target architecture, like the Q100, is an ASIC that allows limited per-query configurability. Such circuits sacrifice the bit- and gate-level configurability of an FPGA for denser logic and higher clock frequencies. Another positive consequence of reduced configurability is that configuring DB-Mesh requires fewer bits, reducing device configuration time. This can lead to increased opportunity to re-program the device more frequently to fit the workload.

While this paper focuses on nested-loops joins, our platform aims to support a variety of database operators. We have therefore chosen a relatively generic array design and we avoid join-specific specializations of the systolic array. Our join protocol resembles a "handshake join" [5, 9], but only one relation is flowing, and output results have to flow through a transfer network to output ports.

## 7 CONCLUSIONS

In previous work, we described the Q100, a small, fast accelerator hard-coded for common database operators. In the present paper, we describe DB-Mesh, a configurable fabric that is designed for database work, and can execute functions that cannot be executed on the Q100, either because the operator isn't common enough to hard-code it, or because a query needs more instances of an operator than the Q100 provides.

We have demonstrated that joins requiring all pairs of records to be matched can be mapped to DB-Mesh. While straightforward mappings risk deadlock if records contain more than one packet of data, we prove that our mapping is free from deadlock. We have also examined different protocols for transferring join results to output ports, highlighting their performance as the resources devoted to the join are varied.

## 8 ACKNOWLEDGMENTS

This research was supported by National Science Foundation grants IIS-1422488 and CCF-1065338. Kim's contributions were supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

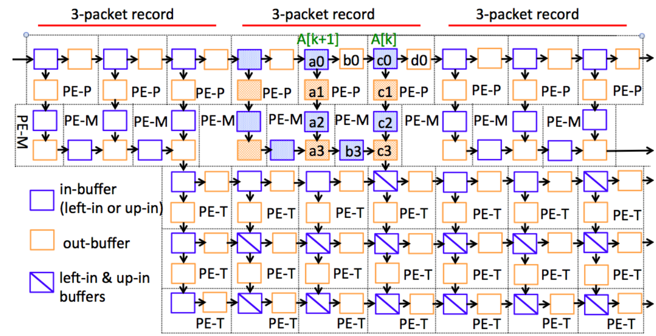


**Figure 10: Dataflow between buffers in DB-Mesh for Joins of Long Records**

## REFERENCES


[1] Bingyi Cao, Kenneth A. Ross, Martha A. Kim, and Stephen A. Edwards. 2015. Implementing Latency-Insensitive Dataflow Blocks. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. ACM, Austin, Texas, 179–187.

[2] H. T. Kung and Philip L. Lehman. 1980. Systolic (VLSI) Arrays for Relational Database Operations. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 105–116.

[3] Mohammedreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2013. Flexible Query Processor on FPGAs. *PVLDB* 6, 12 (2013), 1310–1313. http://www.vldb.org/pvldb/vol6/p1310-sadoghi.pdf

[4] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2015. The FQP Vision: Flexible Query Processing on a Reconfigurable Computing Fabric. *SIGMOD Record* 44, 2 (2015), 5–10.

[5] Yasin Oge, Takefumi Miyoshi, Hideyuki Kawashima, and Tsutomu Yoshinaga. 2013. A fast handshake join implementation on FPGA with adaptive merging network. In *Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, Baltimore, Maryland, 44:1–44:4.

[6] Jian Ouyang, Wei Qi, Yong Wang, Yichen Tu, Jing Wang, and Bowen Jia. 2016. SDA: Software-Defined Accelerator for general-purpose big data analysis system. In *Proceedings of HotChips*. IEEE, Cupertino, California.

[7] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. *SIGOPS Operating System Review* 50, 2 (June 2016), 651–665.

[8] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (third ed.). McGraw-Hill, Inc., New York, NY, USA.

[9] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 625–636.

[10] Jens Teubner, Louis Woods, and Chongling Nie. 2012. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, Scottsdale, Arizona, 229–240.

[11] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2014. Hardware Partitioning for Big Data Analytics. *IEEE Micro* 34, 3 (May 2014), 109–119.

[12] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Salt Lake City, Utah, 255–268.


## A PROOF DETAILS

As described in Section 4.1, DB-Mesh processes joins using the processing chain (P), the merge chain (M) and the transfer network (T). For an all-to-all join in DB-Mesh, each packet of probe table records streams across P; along the way, once there is a match, packets of the probe record and corresponding build record are copied downwards to memory through T and M. In this model, we define *deadlock* as a situation where either

- A generated output record (or part of it) never exits DB-Mesh through an output port, or
- A probe table record does not make it through the entire length of the processing chain.

DB-Mesh can be conceptualized as an array of in/out-buffers. Figure 10 is a more detailed version of Figure 4 showing the internal buffers. The following are important properties of buffers and other structures in DB-Mesh:

P1 The bottom-row out-buffers of M ensure that all packets of a previous record get out before any packets from later records. For example, records flowing downwards to a PE-M element may need to be held up until all packets from previous records, flowing from the left, have passed. By using counters, the PE-M can know whether previous records are complete, even if the next packet is not immediately available in an adjacent buffer.

P2 Buffers in DB-Mesh do not interleave packets of different records. Again, using counters, processing elements can know when they are in the middle of a packet sequence for a record, and are thus required to block potential incoming packets from new records.

P3 The memory won't send any packet to DB-Mesh from the left side until all packets of the record is ready.

P4 The memory won't be indefinitely blocked.

P5 The inputs are finite relations (not infinite streams), so livelock is not a concern.

LEMMA A.1. *Suppose a packet $A[k]$ from a probe table record advances along the PE-P chain. If the following packet $A[k+1]$ from the same record is in the in-buffer of the left neighbor PE-P, then $A[k+1]$ must also advance.*

*Proof:* The simpler case is when no join result is being generated. In that case, no downward traffic is generated by either $A[k]$ or $A[k+1]$, and both are free to move right if there is space. The more interesting case is when join results are being generated downwards by both $A[k]$ and $A[k+1]$. Recall that join results for a record are generated when all probe packets align with corresponding build packets. We argue by contradiction. Suppose $A[k+1]$ doesn't advance while $A[k]$ does. The starting state is as illustrated in Figure 10: $A[k+1]$ and $A[k]$ are in the buffers labeled a0 and c0. The only reason that $A[k+1]$ cannot advance is that buffers a1 and a2 are filled with packets from a previous record, $B$, and $A[k+1]$ needs to be copied down into a1. (If a1 was full but a2 was not, then data could flow from a1 to a2 and from a0 to a1, since PEs each have two buffers.) One step before this, a3 and b3 (or buffers to the left of them) must be filled with packets from a record earlier than $B$. Otherwise, packets in a1 and a2 would have moved on the previous step and a1 should be free now. By property P1, c1 and c2 must filled with packets of $B$ at this moment. Since $A[k]$ needs to have also generated its join record, the fact that c1 and c2 are full contradicts the assumption that $A[k]$ advances.

Together with property P3, we have the following corollary.

COROLLARY A.2. *Consecutive packets from probe table records are always found in the same or consecutive PE-Ps.*

LEMMA A.3. *No buffer in a PE-M will be reserved but empty forever.*

*Proof:* To be reserved, the buffer must have already seen one or more packets of this record. Based on P1, that means that the merging chain and processing chain are free of all earlier records that could block progress. Each path down to the merge chain has a capacity of two packets, which in total is enough to hold the entire join record. Since probe table records stay contiguous (Lemma A.1), all probe table packets become ready to generate join results at the same time; there are no trailing packets to wait for. Therefore, after a finite number of steps, all packets for the join record will flow to the reserved buffer. Once the final packet passes, the buffer will no longer be reserved.

LEMMA A.4. *No buffer in a PE-T will be reserved but empty forever.*

*Proof:* Data flows from some merging chain to the reserved PE-T. Since there is no permanent impediment to flow out of the merging chain (Lemma A.3), the PE-T will eventually see all packets as the path to the PE-T is reserved for this join record (Property P2).

LEMMA A.5. *No buffer in PE-M or PE-T can hold a packet forever.*

*Proof:* Suppose a buffer does hold a packet forever. The cause of this blockage must be that the right neighbor (or possibly the down neighbor if there is one) is permanently unable to accept the packet. This cannot be due to normal (i.e., unblocked) processing in the neighbors, because the workload is finite (Property P5). The neighbor cannot be permanently blocked waiting for a packet (Lemma A.3 and Lemma A.4). That means the neighbor itself must be holding a packet forever. We can then shift attention to the neighbor and apply the same argument to show that it must have a neighbor that is holding a packet forever. But eventually we must get to an output port, because DB-Mesh is acyclic, which contradicts the assumption that the node adjacent to the output port is permanently blocked (Property P4).

COROLLARY A.6. *All packets in PE-M or PE-T will arrive at memory in a finite number of steps.*

LEMMA A.7. *All-to-all joins in DB-Mesh are deadlock-free.*

*Proof:* All join results exit the network in a finite number of steps, according to Corollary A.6. The only possible remaining source of deadlock is a probe table packet that does not make it all the way across the processing chain. We argue by contradiction. Let $P$ be the rightmost packet in the processing chain that becomes permanently blocked. Because it is the rightmost in the processing chain, it cannot be blocked on the right. It must therefore be blocked because of backpressure from the merging chain and transport chain. However, according to Corollary A.6 that block must eventually be released, and $P$ can subsequently progress, completing the contradiction.