How Soccer Players Would do Stream Joins

Jens Teubner Rene Mueller
Systems Group, Department of Computer Science, ETH Zurich
firstname.lastname@inf.ethz.ch

ABSTRACT

In spite of the omnipresence of parallel (multi-core) systems, the predominant strategy to evaluate *window-based stream joins* is still strictly sequential, mostly just straightforward along the definition of the operation semantics.

In this work we present *handshake join*, a way of describing and executing window-based stream joins that is highly amenable to parallelized execution. Handshake join naturally leverages available hardware parallelism, which we demonstrate with an implementation on a modern *multi-core system* and on top of *field-programmable gate arrays (FPGAs)*, an emerging technology that has shown distinctive advantages for high-throughput data processing.

On the practical side, we provide a join implementation that substantially outperforms CellJoin (the fastest published result) and that will directly turn any degree of parallelism into higher throughput or larger supported window sizes. On the semantical side, our work gives a new intuition of window semantics, which we believe could inspire other stream processing algorithms or ongoing standardization efforts for stream query languages.

1. INTRODUCTION

One of the key challenges in building database implementations has always been an efficient support for *joins*. The problem is exacerbated in *streaming databases*, which do not have the option to pre-compute access structures and which have to adhere to *window semantics* in addition to value-based join predicates.

In this work we present *handshake join*, a stream join implementation that naturally supports *hardware acceleration* to achieve unprecedented data throughput. Handshake join is particularly attractive for platforms that support very high degrees of parallelism, such as *multi-core CPUs*, *field-programmable gate arrays (FPGAs)*, or *massively parallel processor arrays (MPPAs)* [4]. FPGAs were recently proposed as an escape to the inherent limitations of classical CPU-based system architectures [18, 19].

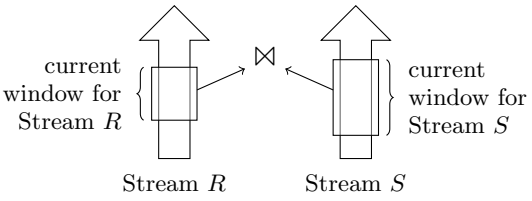


Figure 1: Window join (figure adopted from [14]).

Unlike some of the existing stream join algorithms designed for single-core execution [12, 25], handshake join is agnostic with respect to the join predicate and does not, for instance, depend on equi-join predicates to operate efficiently (handshake join shares this property with the recently proposed CellJoin [7]).

Our main contribution is a stream join algorithm that, by adding compute cores, can trivially be scaled up to handle larger join windows, higher throughput rates, or more compute-intensive join predicates. As a side effect, our work provides a new intuition to the semantics of stream joins, which ultimately might aid ongoing efforts toward a standard language and semantics for streaming databases [13].

We first recap the semantics of stream joins in Section 2, along with typical implementation techniques in software. Section 3 introduces handshake join. Section 4 discusses how handshake join could be implemented in computing systems, which we realize with a prototype implementation on top of a modern multi-core CPU (Section 5) and with a massively parallel implementation for FPGAs (Section 6). In Section 7 we relate our work to others', before we wrap up in Section 8.

2. STREAM JOINS

It is the very nature of stream processing engines to deal with unbounded, "infinite," input data. These data arrive by means of a stream and have to be processed immediately, in real time.

2.1 Windowing

Infinite input data causes obvious semantical problems when some of the classical database operators—most notably joins and aggregates—are to be evaluated over data streams. This has lead to the notion of *windows* in the streaming community. By looking at a finite subset of the input data (a window), all algebraic operations become semantically sound again.

Figure 1 (adopted from [14]) illustrates this for the case of

a join operation. The join in the middle is always evaluated only over finite subsets taken from both input streams. Windows over different input data can span different numbers of tuples, as indicated by the window sizes in Figure 1.

Various ways have been proposed to define suitable window boundaries depending on application needs. In this work we focus on *sliding windows* which, at any point in time, cover all tuples from some earlier point in time up to the most recent tuple. Usually, sliding windows are either time-based, *i.e.*, they cover all tuples within the last τ time units, or tuple-based, *i.e.*, they cover the last w tuples in arrival order. To ease discussions, we always assume count-based windows in the conceptual part of this paper. We have successfully built (and measured) implementations for both window types.

2.2 Sliding-Window Joins

The exact semantics of window-based joins (precisely which stream tuple could be paired with which?) in existing work was largely based on how the functionality was implemented. For instance, windowing semantics is implicit in the three-step procedure devised by Kang *et al.* [14]. The procedure is performed for each tuple r that arrives from input stream R:

1. *Scan* stream S's window to find tuples matching r.
2. *Insert* new tuple r into window for stream R.
3. *Invalidate* all expired tuples in stream R's window.

Tuples s that arrive from input stream S are handled symmetrically. Sometimes, a transient access structure is built over both open windows, which accelerates Step 1 at the cost of some maintenance effort in Steps 2 and 3.

The three-step procedure carries an implicit semantics for window-based stream joins:

Semantics of Window-Based Stream Joins. *For $r \in R$ and $s \in S$, the tuple $\langle r, s\rangle$ appears in the join result $R \bowtie_p S$ iff*

(a) r arrives after s and s is in the current S-window at the moment when r arrives or

(b) r arrives earlier than s and r is still in the R-window when s arrives

and r and s pass the join predicate p.

A problem of the three-step procedure is that it is not well suited to exploit the increasing degree of *parallelism* that modern system architectures support. Optimal use of many-core system demands *local* availability of data at the respective compute core, which contradicts the nature of the join problem, where *any* tuple in the opposite window represents a possible match.

Gedik *et al.* [7] discuss *partitioning* and *replication* as a possible solution. Thereby, either the in-memory representation of the two windows is partitioned over the available compute resources or arriving tuples are partitioned over cores. Local data availability then needs to be established explicitly by *replicating* the respective other piece of data (input tuples or the full in-memory windows). Especially for highly dynamic problems like the one at hand, this can cause significant coordination overhead that limits scalability to a very large number of parallel processing units (in the implementation of Gedik *et al.* the PowerPC unit (PPU) of the Cell Broadband Engine takes the coordinator role).

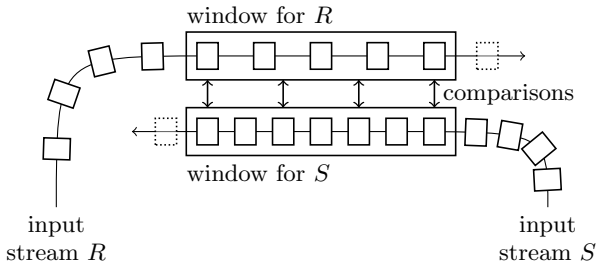


Figure 2: Handshake join idea. Streams flow by each other in opposite directions; comparisons (and result generation) happens in parallel as the streams pass by.

3. HANDSHAKE JOIN

In the most generic case, the three-step procedure of Kang *et al.* [14] corresponds to a *nested loops-style* join evaluation. To evaluate a stream join $R \bowtie_p S$, the scan phase first *enumerates* all combinations $\langle r, s\rangle$ of input tuples $r \in R$ and $s \in S$ that satisfy the window constraint. Then, the resulting tuple pairs are *filtered* according to the join predicate p and added to the join result. For certain join predicates (such as equality or range conditions), specialized in-memory access structures, typically hash tables or tree indices, can help reduce the number of pairs to enumerate.

3.1 Soccer Players

As sketched before, the enumeration of join candidates may be difficult to distribute efficiently over a large number of processing cores. Traditional approaches usually assume a central coordinator that partitions and replicates data as needed over available cores. But it is easy to see that this will quickly become a bottleneck as numbers of cores increase. The aim of our work, scale-out to very high degrees of parallelism, thus largely defeats any enumeration scheme that depends on centralized coordination.

It turns out that we can learn from soccer players here. Soccer players know very well how all pairs of players from two opposing teams can be enumerated without any external coordination. Before the beginning of every game, it is tradition to shake hands with all players from the opposing team. Players do so by *walking by* each other in opposite directions and by *shaking hands* with every player that they encounter.

Very naturally, the procedure *avoids bottlenecks* (each person has to shake only one hand at a time), keeps all interaction *local* (fortunately—people's arm lengths are limited), and has a very simple *communication pattern* (players only have to walk one step at a time and there is no risk of collision). All three aspects are desirable also in the design of parallel algorithms.

3.2 Stream Joins and Handshaking

The handshake procedure used in sports games inspired the design of *handshake join*, whose idea we illustrated in Figure 2.

Tuples from the two input streams R and S, marked as rectangular boxes $\square$, are pushed through respective join windows. Upon window entrance, each tuple pushes all existing window content one step to the side, such that always

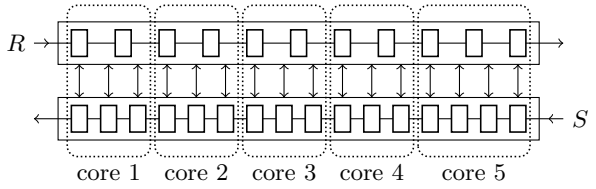


Figure 3: Parallelized handshake join evaluation. Each compute core processes one segment of both windows and performs all comparisons locally.

the oldest tuple "falls out" of the window and expires. In software, this process could be modeled with help of a *ring buffer* or a *linked list*[1]; in FPGA-based setups, a *shift register* would serve the same purpose. Both join windows are lined up next to each other in such a way that window contents are pushed through in opposing directions (cf. Figure 2).

Whenever two stream tuples $r \in R$ and $s \in S$ encounter each other (in a moment we will discuss what that means and how it can be implemented), they "shake hands", *i.e.*, the join condition is evaluated over r and s, and a result tuple $\langle r, s \rangle$ appended to the join result if the condition is met. Many "handshakes" take place at the same time, work that we will parallelize over available compute resources.

To keep matters simple, we assume that there is a new item inserted into only *one* window at any one time. An implementation is free to lift this restriction, provided that it properly deals with race conditions.

3.3 Semantics

While a stream item $r \in R$ travels along its join window, it will always encounter at least those S-tuples s_i that were already present in S's join window when r entered the arena. Likewise, if r arrived earlier than some S-tuple s, r and s will meet eventually (and thus form a join candidate) whenever r is still in the R-window at the moment when s arrives.

Observe how this semantics *coincides* with the window semantics implied by the three-step procedure of Kang *et al.* [14]. Thus, handshake join implements the *same* semantics as existing techniques; only the order of tuples in the result stream (and thus also per-tuple latencies) might change.[2] In addition, handshake join *(a)* gives a new *intuition* on what window-based joins mean and *(b)* opens opportunities for effective *parallelization* on modern hardware. Next, we will demonstrate how to exploit the latter.

3.4 Parallelization

Figure 3 illustrates how handshake join can be parallelized over available compute resources. Each processing unit (or "core") is assigned one *segment* of the two join windows. Tuple data is held in local memory (if applicable on a particular architecture), and all tuple comparisons are performed locally.

Data Flow vs. Control Flow. This parallel evaluation became possible because we converted the original *control flow* problem (or its procedural three-step description) into a *data flow* representation. Rather than synchronizing join execution from a centralized coordinator, processing units are now driven by the flow of stream tuples that are passed on directly between neighboring cores. Processing units can observe locally when new data has arrived and can decide autonomously when to pass on tuples to their neighbor.

The advantages of data flow-style processing have been known for a long time. Their use in shared-nothing systems was investigated, *e.g.*, by Teeuw and Blanken [23]. Modern computing architectures increasingly tend toward shared-nothing setups, and new data flow-oriented formulations were proposed only recently for distributed database architectures [10] or to solve the frequent item problem on FPGAs [24].

Communication Pattern. In addition, we have established a particularly simple *communication pattern*. Processing units only interact with their immediate neighbors, which may ease inter-core routing and avoid communication bottlenecks. In particular, handshake join is a good fit for architectures that use *point-to-point* links between processing cores—a design pattern that can be seen in an increasing number of multi-core platforms (examples include HyperTransport links, the QuickPath interconnect used in Intel Nehalem systems, or Tilera's iMesh architecture with current support for up to 100 cores).

Scalability. Both properties together, the representation as a data flow problem and the point-to-point communication pattern along a linear chain of processing units, ensure scalability to large numbers of processing units. Additional cores can either be used to support larger window sizes without negative impact on performance, or to reduce the workload per core, which will improve throughput for high-volume data inputs.

3.5 Encountering Tuples

We yet have to define what exactly it means that stream tuples in the two windows "encounter" each other and which pairs of stream elements need to be compared at any time. Note that, depending on the relative window size configuration, stream items might never line up in opposing positions exactly. Both our previous handshake join illustrations were examples of such window size configurations (only every second R-tuple lines up exactly with every third S-tuple in Figure 3, for instance).

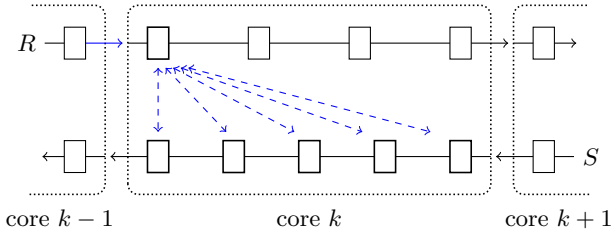
For proper window join semantics, the only assumption we made in Section 3.3 was that an item that enters either window will encounter all current items in the other window *eventually*. That is, there must not be a situation where two stream items can pass each other without being considered as a candidate pair. Thus, any local processing strategy that prevents this from happening will do to achieve correct overall window semantics.

Eager Scan Strategy. One particular strategy (which we will call *eager scan strategy*) that can be used to process a segment k is illustrated in Figure 4. In this illustration, we assume that every tuple $r \in R$ is compared to all S-tuples in the segment at the moment when r enters the segment. Figure 4(a) shows all tuple comparisons that need to be performed when a new R-tuple is shifted into the segment.

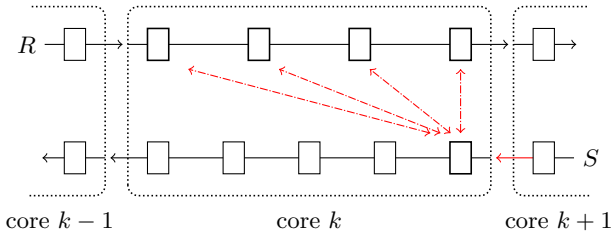
Likewise, when a new tuple $s \in S$ enters the segment, it is immediately compared to all R-tuples that are already in the segment, as illustrated in Figure 4(b). The strategy

[1]Similar data structures are typically used to determine expired tuples for Step 3 of the three-step procedure in [14].

[2]This disorder can easily be corrected with help of punctuations [16].



(a) Tuple from stream R entered segment.



(b) Tuple from stream S entered segment.

Figure 4: Eager scan strategy. A tuple entering from stream R or S will trigger comparisons ↙↗ or ↖↘ in the segment of processing core k (respectively).

will operate correctly regardless of the window size configuration. Similarly, segmentation can be chosen arbitrarily, which might be useful for very particular data characteristics or in systems with a heterogeneous set of processing cores.

Other Strategies. Observe that the eager scan strategy is equivalent to the three-step procedure of Kang *et al.* [14], but it is now used to process only a segment of the full stream join problem. Thus, Kang's three-step procedure may be seen as a specific implementation of handshake join that runs on a single core and uses the eager scan strategy.

One consequence is that we can plug in any (semantically equivalent) stream join implementation to operate on each processing unit, including those that use additional access structures or advanced join algorithms. Handshake join then becomes a mechanism to distribute and parallelize the execution of an existing algorithm over many cores. In Section 5 we demonstrate this by running *merge join* on the individual processing cores.

3.6 Time-Based Windows

Our discussion so far focussed on the processing of tuple-based windows, where every newly arriving tuple directly triggers the expiration of the oldest entry in the respective window. For time-based semantics, by contrast, tuple additions and removals have to be performed independently.

Handshake join can naturally be applied to such a scenario, too. Time stamps now determine when tuples have to be propagated between segments (or dropped after the last segment), rather than the arrival of new data. A system that uses handshake join only to orchestrate the distributed execution of an existing stream join implementation will thus readily be prepared to support time-based windows (provided that the local join implementation is).

Bursty Input Streams. For tuple-based windows, fluctuations in the input data rates will directly and equally affect

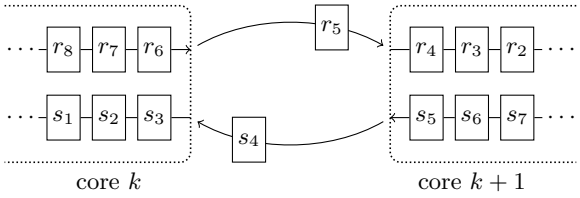


Figure 5: Missed-join pair problem. Tuples sent via message queues might miss each other while on the communication channels.

all involved processing cores. By contrast, with time-based semantics, a workload peak initially will only hit the first segment of the respective join window. The peak will form a "bubble" that then travels down the chain of processing cores. The effect of "bubbles" on core utilization is part of our current research agenda.

4. REALIZING HANDSHAKE JOIN

Our discussion so far was primarily based on a high-level intuition of how tuples flow between segments in a handshake join setup. We now map the handshake join intuition onto realistic communication primitives that can be used to realize handshake join on actual hardware, including commodity CPUs or FPGAs.

4.1 Lock Step Forwarding

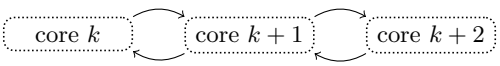
In Section 3, we assumed that a newly arriving tuple would synchronously push all tuples of the same stream through the respective window. Essentially, this implies an atomic operation over all participating processing cores, *i.e.*, upon a new tuple arrival, all cores must simultaneously forward their oldest tuple to the respective left/right-next neighbor.

Obviously, an implementation of such lock step processing would obliterate a key idea behind handshake join, namely high parallelism without centralized coordination. This problem arises at least for commodity hardware, such as multi-core CPU systems. It turns out that lock step processing is still a viable route for FPGA-based implementations, as we shall see in Section 6.1.

4.2 Asynchronous Message Queues

As an alternative, the data flow of handshake join suggests the use of *asynchronous message passing* between neighboring cores, a communication mode that is known to scale well with increasing core counts [3].

In particular, one pair of *FIFO queues* (indicated as arrows ⇝ below) between any two neighboring processing cores is sufficient to support data propagation along the chain of cores in either direction:



.

FIFO queues can be tuned for high efficiency and provide fully asynchronous access. In particular, no performance-limiting locks or other atomic synchronization primitives are necessary to implement point-to-point message queues.

Though supportive of increased parallelism, asynchronous communication between cores can bear an important risk. As illustrated in Figure 5, tuples from opposing streams

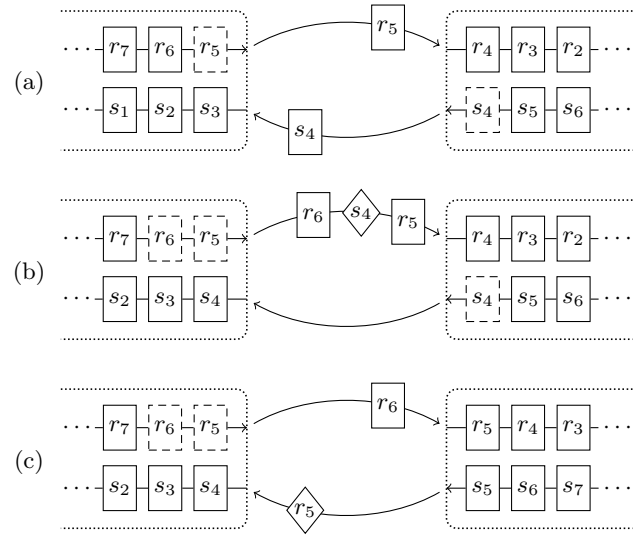


Figure 6: Two-phase tuple forwarding. Tuples are kept in local join windows and removed only after an acknowledgement message has been received from the neighboring core.

might *miss* each other if they both happen to be in the communication channel at the same time (in Figure 5, tuples r_5 and s_4 are both in transit between the neighboring cores k and $k+1$; thus, no comparison is attempted between tuples r_5 and s_4).

Two-Phase Forwarding. The missing of join candidates can be avoided by separating tuple forwarding into a two-phase process. In the first phase, the sending core C_{send} places the tuple t into the message queue, but still keeps a copy of t in its local join window (see Figure 6(a); tuples r_5 and s_4 have both been placed into message queues). Tuple t is marked as being forwarded (indicated using dashed boxes in Figure 6), but remains available to be joined with incoming tuples.

The second phase is initiated after the receiver core C_{recv} has accepted the tuple and after it has placed t into the C_{recv}-local join window. Core C_{recv} will send an *acknowledgement message*, indicated as a diamond-shaped message in Figure 6, back to the source core C_{send} to confirm the arrival of t. In Figure 6(b), we assume that tuple s_4 has been accepted by the left core (and paired up with r_5 through r_7); an acknowledgement message $\langle s_4 \rangle$ was placed into the message queue to notify the right core; then the left core initiated the send phase for another tuple $[r_6]$.

Figure 6 illustrates how the two-phase protocol avoids the missed-join pair problem. When, after the situation in Figure 6(b), the right core accepts tuple r_5, the tuple will still be compared with s_4. This is because s_4 has only been marked as forwarded, but not yet been removed from the join window (note that r_5 did not see s_4 on the left core before). Next, the right core will process the acknowledge message $\langle s_4 \rangle$ and remove s_4 from the bottom join window (Figure 6(c)) before it accepts r_6. Thus, r_6 (which was compared with s_4 already on the left core) will not meet s_4 on the right core again.

FIFO Queues. Note that tuple data and acknowledge messages must be sent over the same message channels. In the example of Figure 6, correct semantics is only guaranteed if the acknowledgement for s_4 is received and processed by the right core *in-between* the two tuples r_5 and r_6.

```
1  Procedure: handshake_join ()
2  while true do
3      if message waiting in leftRecvQueue then
4          process_left () ;
5      if message waiting in rightRecvQueue then
6          process_right () ;
7      expire_outdated_tuples () ;
8  Procedure: process_left ()
9  msg ← message from leftRecvQueue ;
10 if msg contains a new tuple then
11     r_i ← extract tuple from msg ;
12     scan S-window to find tuples that match r_i ;
13     insert r_i into R-window ;
14     place acknowledgement for r_i in leftSendQueue ;
       /* trigger expiration if tuple-based      */
15     mark oldest not-yet-sent R-tuple for expiration ;
16 else
       /* msg is an acknowledgement message      */
17     remove oldest tuple from S-window ;
18 Procedure: expire_outdated_tuples ()
   /* handle R-tuples                            */
19 foreach r_i that should expire, oldest first do
20     if r_i is not marked as sent then
21         place message with r_i into rightSendQueue ;
22         mark r_i as sent ;
   /* handle S-tuples analogously (not shown)    */
23 ...
```

Figure 7: Handshake join with asynchronous message passing (ran on each core).

Figure 7 describes an implementation of handshake join based on asynchronous message passing (only one direction of tuple flow shown; opposite direction is handled symmetrically). This code is ran on every participating core. Procedure `handshake_join ()` selects one of the two incoming message queues and invokes a `process_... ()` handler to process the message.

The `process_... ()` handlers essentially implement Kang's three-step procedure [14], but also place acknowledgement messages on the return queue (line 14). Tuples expire (*i.e.*, are considered in `expire_outdated_tuples ()`) if they have been marked for expiration in a `process_... ()` handler (tuple-based semantics) or after their time stamps have expired (time-based windows).

Our illustrations in Figure 6 suggested an explicit tuple reference in each acknowledgement message. In practice, this information is redundant, since an acknowledgement will always reference the oldest tuple in the predecessor core. This is reflected in line 15 of Figure 7, which does not actually inspect the content of the acknowledgement message.

4.3 Synchronization at Pipeline Ends

Another potential synchronization bottleneck concerns the

ends of the two stream pipelines. If tuple-based semantics is requested, each newly arriving input tuple must trigger the expiration of another tuple at the other end of the window. In a handshake join setup, an expired tuple must be discarded, *e.g.*, in the *rightmost* window segment whenever a new tuple is accepted by the *leftmost* processing core.

Such behavior can be realized at negligible overhead with help of the acknowledgement mechanism described previously. To this end, each data source generates *two* messages for each input tuple, which are sent to the opposing ends of the handshake join pipeline. The actual tuple is sent to its respective window input (*e.g.*, R-tuples are sent to the leftmost processing core), while an acknowledgement message is sent simultaneously to the opposite end (*e.g.*, rightmost core), where it will cause the necessary removal of the oldest tuple.[3]

We are now interpreting acknowledgement messages as explicit tuple removal instructions. Similar devices have been found useful in existing stream processing engines, where they are referred to as − *elements* (Stanford STREAM, [2]), *negative tuples* (Nile, [8]), or *deletion messages* (Borealis, [1]). These systems will readily provide all necessary functionality to plug in handshake join seamlessly.

5. HANDSHAKE JOIN IN SOFTWARE

Handshake join can effectively leverage the parallelism of modern multi-core CPUs. To back up this claim, we evaluated the behavior of handshake join on a recent Intel Nehalem EX machine (Intel Xeon X7560 @ 2.27 GHz) that sports four eight-way CPUs or a total of 32 CPU cores (we left HyperThreading turned off).

Our implementation uses the same lock-free messaging mechanism as detailed in [3] and uses a benchmark setup similar to that of [7] (time-based windows, fixed tuple sizes of 24 and 28 bytes, band join condition over two attributes, join selectivity $1 : 250,000$). We ran all experiments with symmetric input data rates; reported rates are always *per stream*.

Since our main interest is in scalability, we refrained from low-level optimizations (such as the SIMD optimizations in CellJoin [7]) and implemented the pseudo code of Figure 7 (*i.e.*, the *eager scan strategy*) in a straightforward fashion. Processing cores are realized as Linux threads and all pinned to separate CPU cores. A single *dispatcher thread* generates data, feeds it into the pipeline ends, and collects results from all processing cores.

Figure 8 illustrates the achievable input data throughput for a varying number n of processing cores. Since the task itself is essentially a nested-loops join (cf. line 12 in Figure 7), CPU load will grow quadratically with the input data rate (both input rates are increased simultaneously) and we can at best expect a throughput growth proportional to the square root of n. An ideal square root shape is indicated in Figure 8 as a dashed line, which we can see is very closely followed by the actual measurements.

As can be seen in Figure 8, additional processing cores can be used to support larger join windows, higher throughput, or a combination of both. In terms of absolute performance, CellJoin [7] is the most relevant baseline for our work. CellJoin achieves somewhat higher throughput than

[3]Some additional care will be needed during warm-up, when join windows are not yet properly filled with data.

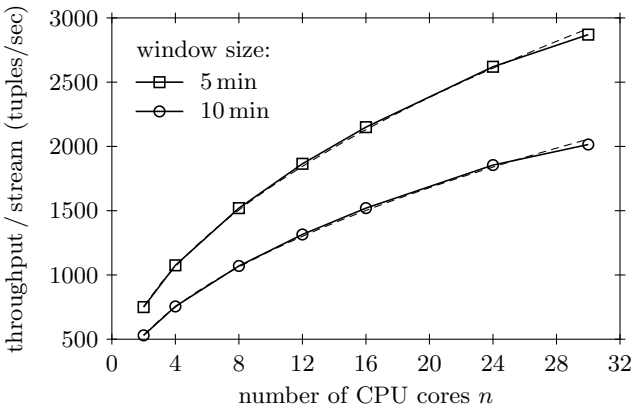


Figure 8: Handshake join scalability on multi-core CPU (Intel Nehalem EX; eager scan strategy). Dashed lines would be ideal scaling.

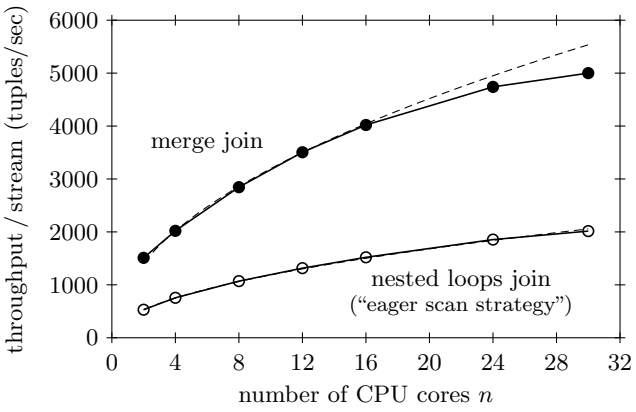


Figure 9: The use of merge join for local join execution increases throughput, yet scales well to large core numbers (10 min windows).

our prototype (approximately 3000 tuples/sec and 1500 tuples/sec for window sizes of 5 and 10 minutes), but depends on aggressive SIMD tuning to do so.

Handshake Join Under Pressure. Instead of scanning tuples eagerly, our band join problem would also allow the use of *merge join* for local join execution (after batching data in basic blocks and sorting them). This alternative promises higher throughput rates, but also puts additional pressure on the dispatcher thread. The dispatcher will now have to deliver higher input data rates; perform additional basic block pre-processing (including sorting); and collect more result data from the processing cores.

Figure 9 illustrates how handshake join reacts to such pressure (we used a basic block size of 128 tuples). In spite of the throughput pressure, our implementation scales well up to 16 cores (4000 tuples/sec). Beyond 16 cores, our implementation became bottlenecked in its result collection routine (2×5000 input tuples/sec will yield more than 120,000 tuples/sec on the output!). Performance goes back to ideal scaling once we use a more restrictive join predicate that produces reasonable output sizes (not shown in graph).

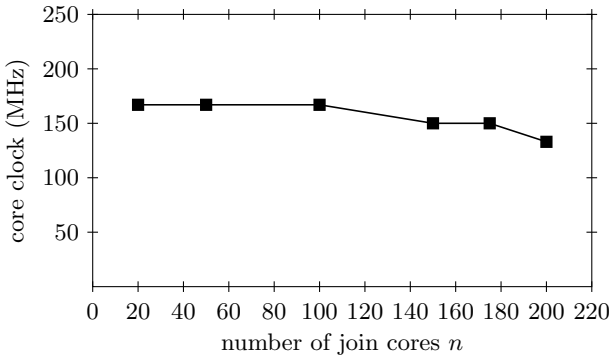


Figure 10: Scalability of FPGA Handshake join with a constant segment size of 8 tuples per window and core.

6. HANDSHAKE JOIN ON FPGAS

32 CPU cores clearly do not mark the end of the multi-core race. To see how handshake join would scale to very large core numbers, we used *field-programmable gate arrays (FPGAs)* as a simulation platform, where the only limit to parallelism is the available chip space. FPGAs themselves are an interesting technology for database acceleration [18, 19, 24], but our main focus here is to demonstrate scalability to many cores.

Current FPGA chips provide chip space to instantiate up to a few hundred simple *join cores*. The cores contain local storage for the respective window segments of R and S and implement the join logic. In this paper we implement simple nested loops-style processing. To keep matters simple (with simpler cores we can instantiate more of them), we only look at *tuple-based* windows that fit into on-chip memory (flip-flop registers).

Our prototype sticks to a simple stream format with 32-bit join keys and 32 bits of payload per tuple. We assume an equality join predicate and return 96 bit-wide result tuples. Hardware-provided *shift registers* enable *lock step tuple forwarding* at low cost (cf. Section 4.1). We defer further details of the implementation into Appendix A.

6.1 Scalability of Parallelism

FPGAs provide a very direct measure of the scalability that an algorithm can provide. In a scalable design, the *maximum clock frequency* at which a circuit can be operated is independent of the configured size of the circuit. For algorithms that scale less favorably, the clock frequency will have to be turned down as the configuration size increases (Appendix A.5 details how clock frequency and throughput are related).

The maximum clock frequency that can be achieved for our circuit is shown in Figure 10 for different core numbers. For this experiment we scaled up the number of cores n, but left the per-core window size constant at eight tuples per core (for $n = 100$, *e.g.*, the overall window size will be $100 \times 8 = 800$ tuples per stream). As can be seen in the figure, clock frequencies remain largely unaffected by the core count, which confirms the scalability of handshake join.

On the right end of Figure 10, our design occupies more than 96 % of all available FPGA BRAM resources. As such, we are operating our chip far beyond its maximum recommended resource consumption (70–80 %) [6]. The fact that our circuit can still sustain high frequencies is another indication for good scalability. Earlier work on FPGA-based stream join processing suffered a significant drop in clock speeds for the window sizes we consider, even though their system operated over only 4 bit-wide tuples [20].

7. RELATED WORK

The handshake join mechanism is largely orthogonal to a number of (very effective) techniques to accelerate stream processing. As motivated in Section 3.5, handshake join could, for instance, be used to coordinate multiple instances of *double-pipelined hash join* [12, 25] or window joins that use indexes [9]. If handshake join alone is not sufficient to sustain load, load shedding [22] or distribution [1] might be appropriate countermeasures.

Handshake join's data flow is similar to the *join arrays* proposed by Kung and Lohman [15]. Inspired by the then-new concept of *systolic arrays* in VLSI designs, their proposed VLSI join implementation uses an array of bit comparison components, through which data is shifted in opposing directions.

The only work we could find on stream joins using FPGAs is the *M3Join* of Qian *et al.* [20], which essentially implements the join step as a single parallel lookup. This approach is known to be severely limited by on-chip routing bottlenecks [24], which causes the sudden and significant performance drop observed by Qian *et al.* for larger join windows [20]. The pipelining mechanism of handshake join, by contrast, does not suffer from these limitations.

A potential application of handshake join outside the context of stream processing might be a parallel version of *Diag-Join* [11]. Diag-Join exploits time-of-creation clustering in data warehouse systems and uses a sliding window-like processing mode. The main bottleneck there is usually throughput, which could be improved by parallelizing with handshake join.

8. SUMMARY

The multi-core train is running at full throttle, and the available *hardware parallelism* is still going to increase. Handshake join provides a mechanism to leverage this parallelism and turn it into *increased throughput* for stream processing engines. In particular, we demonstrated how *window-based stream joins* can be parallelized over very large numbers of cores with negligible coordination overhead. Though the focus of this work is more on scalability than on low-level optimizations, our prototype implementation reaches throughput rates that significantly exceed those of CellJoin, the best published result to date [7].

Key to the scalability of handshake join is to avoid any coordination by a centralized entity. Instead, handshake join only relies on local core-to-core communication (*e.g.*, using local message queues) to achieve the necessary core synchronization. This mode of parallelization is consistent with folklore knowledge about *systolic arrays*, but also with recent research results that aim at many-core systems [3].

The principles of handshake join are not bound to the assumption of a single multi-core machine. Rather, it should be straightforward to extend the scope of handshake join to *distributed stream processors* in networks of commodity systems (such as the *Borealis* [1] research prototype) or to

support massively-parallel multi-FPGA setups (such as the BEE3 multi-FPGA system [5] or the *Cube* 512-FPGA cluster [17]).

Acknowledgements

We are grateful to Intel Switzerland for providing us access to pre-release multi-core hardware. Nehalem EX was a great platform to assess the characteristics of handshake join.

9. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2005.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. `http://infolab.stanford.edu/~usriv/papers/streambook.pdf`.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, 2009.

[4] M. Butts. Synchronization Through Communication in a Massively Parallel Processor Array. *IEEE Micro*, 27(5):32–40, 2007.

[5] J. Davis, C. Thacker, and C. Chang. BEE3: Revitalizing Computer Architecture Research. Technical Report MSR-TR-2009-45, Microsoft Research, 2009.

[6] A. DeHon. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100 % LUT utilization). In *Proc. of the Int'l Symposium on Field Programmable Gate Arrays (FPGA)*, pages 125–134, 1999.

[7] B. Gedik, P. S. Yu, and R. Bordawekar. CellJoin: A Parallel Stream Join Operator for the Cell Processor. *The VLDB Journal*, 18(2):501–519, 2009.

[8] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 19(1):57–72, 2007.

[9] L. Golab, S. Garg, and T. Öszu. On Indexing Sliding Windows over Online Data Streams. In *Proc. of the 9th Int'l Conference on Extending Database Technology (EDBT)*, Crete, Greece, 2004.

[10] R. Gonçalves and M. Kersten. The Data Cyclotron Query Processing Scheme. In *Proc. of the 13th Int'l Conference on Extending Database Technology (EDBT)*, Lausanne, Switzerland, 2010.

[11] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: A Opportunistic Join Algorithm for 1:N Relationships. In *Proc. of the 24th Int'l Conference on Very Large Databases (VLDB)*, New York, NY, USA, 1998.

[12] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. of the 1999 ACM SIGMOD Int'l Conference on Management of Data*, Philadelphia, PA, USA, 1999.

[13] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a Streaming SQL Standard. *Proc. of the VLDB Endowment*, 1(2), 2008.

[14] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proc. of the 19th Int'l Conference on Data Engineering (ICDE)*, pages 341–352, Bangalore, India, 2003.

[15] H. T. Kung and P. L. Lohman. Systolic (VLSI) Arrays for Relational Database Operations. In *Proc. of the 1980 ACM SIGMOD Int'l Conference on Management of Data*, Santa Monica, CA, USA, 1980.

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the 2005 ACM SIGMOD Int'l Conference on Management of Data*, Baltimore, MD, USA, 2005.

[17] O. Mencer, K. H. Tsoi, S. Craimer, T. Todman, W. Luk, M. Y. Wong, and P. H. W. Leong. CUBE: A 512-FPGA Cluster. In *Proc. of the Southern Programmable Logic Conference (SPL)*, São Carlos, Brazil, 2009.

[18] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML Filtering Through a Scalable FPGA-Based Architecture. In *Proc. of the 4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2009.

[19] Netezza Corp. `http://www.netezza.com/`.

[20] J.-B. Qian, H.-B. Xu, Y.-S. Dong, X.-J. Liu, and Y.-L. Wang. FPGA Acceleration Window Joins Over Multiple Data Streams. *Journal of Circuits, Systems, and Computers*, 14(4):813–830, 2005.

[21] E. S. Shin, V. J. Mooney, and G. F. Riley. Round-robin arbiter design and generation. In *Proc. of the 15th Int'l Symposium on System Synthesis (ISSS)*, pages 243–248, New York, NY, USA, 2002. ACM.

[22] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, Berlin, Germany, 2003.

[23] W. B. Teeuw and H. M. Blanken. Control Versus Data Flow in Parallel Database Machines. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 4(11):1265–1279, 1993.

[24] J. Teubner, R. Mueller, and G. Alonso. FPGA Acceleration for the Frequent Item Problem. In *Proc. of the 26th Int'l Conference on Data Engineering (ICDE)*, Long Beach, CA, USA, 2010.

[25] A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. of the 1st Int'l Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, FL, USA, 1991.

APPENDIX

A. HANDSHAKE JOIN ON FPGAS

The main motivation behind the FPGA-based implementation that we analyzed in this paper was to show scalability to configurations with a large number of cores. Yet, we feel that our implementation could also guide the development of a high-performance stream join implementation for FPGAs. In this section we provide some of the implementation details of our circuit, in combination with a further experimental analysis of its behavior.

A.1 FPGA Basics

In essence, an FPGA is a programmable logic chip which provides a pool of digital logic elements that can be configured and connected in arbitrary ways. Most importantly the FPGA provides logic elements from of the following types: *lookup tables*, *flip-flop registers* and small blocks of RAM (so-called *BRAM*). Lookup tables are configured to implement arbitrary logic functionality. They are directly connected to flip-flop registers which represent 1-bit distributed fast memory. Units of block RAM can be combined to instantiate larger amounts of on-chip memory. Finally, a configurable *interconnect network* can be used to combine lookup tables and flip-flop registers into complex logic circuits.

FPGAs can be configured to implement arbitrary logic circuits. These circuits are expressed by the developer using a *hardware description language* such as VHDL or Verilog. High-level logic descriptions are *synthesized* into low-level bitstreams and then loaded into the FPGA. In this work, we divide the available chip space into a large number of very simple processing units. Such a model can make efficient use of the available compute density of the FPGA.

A.2 Architecture Overview

Figure 11 illustrates the high-level view of our handshake join implementation on an FPGA. The windows of the R and S streams are partitioned among n cores. The cores are driven by a common clock signal that is distributed over the entire chip. The synchronous operation of the cores avoids any buffering (such as FIFO) between the cores and, thus, reduces the complexity of the implementation. The tuples move in lock-step through the window. The windows represent large shift registers which can be efficiently implemented in hardware.

Following the basic handshake join algorithm (Figure 7) for each core we need to provide a hardware implementation of the segment for the R and S windows, a digital circuit for the join-predicate, and scheduling logic for the tuples and the window partitions. The figure shows the two 64 bit-wide shift registers (labeled 'R window' and 'S window', respectively) that hold the $\langle k, v_R\rangle$ and $\langle k, v_S\rangle$ tuples. When a new tuple is received from either stream, the tuple is inserted in the respective shift register and the key is compared against all keys in the opposite window. In this paper we use a simple nested-loop join, *i.e.*, the elements of the window are compared sequentially. This is done by a simple sequencer that implements the tuple scheduling logic.

Assuming two tuple-based windows with sizes W_R and W_S, the comparison requires $\max(\lceil W_R/n\rceil, \lceil W_S/n\rceil)$ clock cycles. The number of clock cycles can be reduced at the cost of increased circuit complexity by instantiating multiple predicate evaluation sub-circuits, thereby allowing for a parallel execution of the predicates.

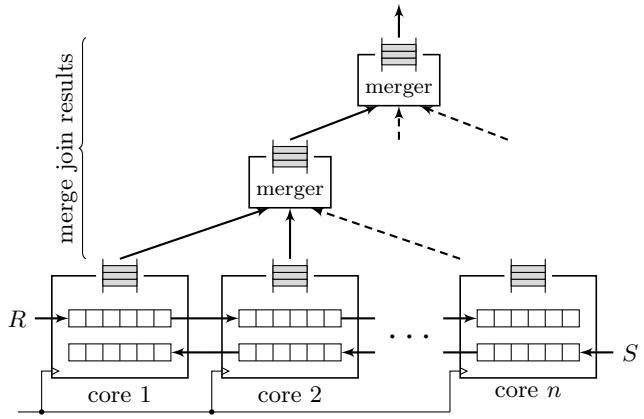


Figure 11: FPGA Implementation of Handshake Join for Tuple-based Windows.

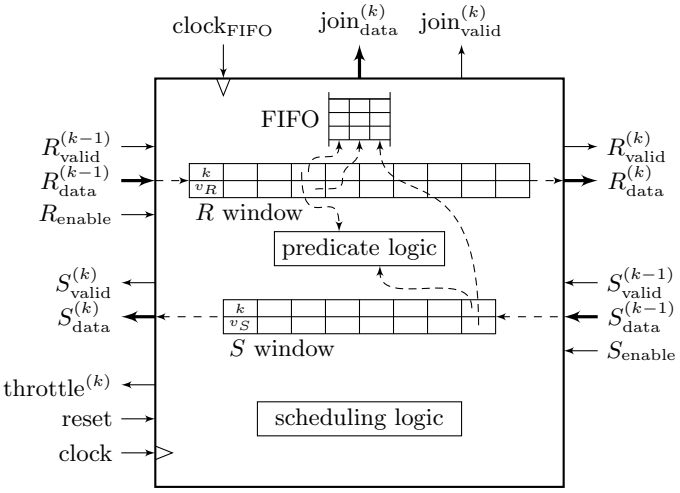


Figure 12: Join Core Implementation on FPGAs.

As illustrated in the top half of Figure 11, each join core will send its share of the join result into a FIFO queue (indicated as ▤). A merging network will merge all sub-results in to the final join output at the top of Figure 11.

A.3 Join Core

As depicted in Figure 12, the stream data for R and S is directly fed into and out of the cores. An additional "valid" line is used to represent whether the data lines contain a valid tuple. Our current implementation allows processing one tuple per clock cycle. The "enable" signals specify whether in any given clock cycle a new tuple is shifted along the R or S stream. These two signals are asserted by a simple admission control circuit when the data streams enter the chip. Every join core can raise a "throttle" signal when its FIFO is close to be come full. The admission controller can use this information to either (1) discard new tuples when they enter the chip or (2) drop tuples waiting in the output FIFO. As such, the throttle signal allows us to handle overload situations in a controlled manner.

The signal is asserted before the FIFOs are actually full such that enough free slots are available for all join tu-

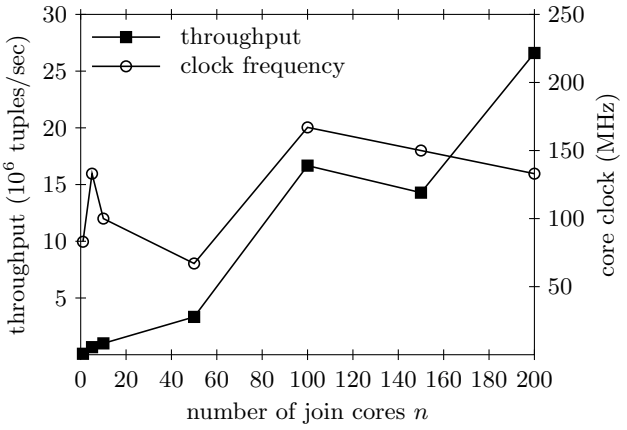


Figure 13: Throughput and clock frequency for two windows of size 1,000 implemented using different numbers of cores.

ples that could be generated from the current input tuple. In other words, we assert the throttle signal if less than $\max(\lceil W_R/n \rceil, \lceil W_S/n \rceil)$ entries are free in the output FIFO of a core.

A.4 Merging Network

Joins can potentially generate an large amount of result tuples. It is therefore crucial to siphon off the result data from the join cores to avoid overflows in the local output FIFOs. The output bandwidth is determined by the speed of the top-most merger FIFO, which can accept and deliver one tuple per clock. The merging network is driven by different clock than the shift-register and the tuple scheduling logic. This allows us to balance throughput and latency depending on the join hit-rate. For example, a high clock frequency in the merging network and a comparatively small shift-clock can be used for joins that are known to yield a high hit-rate. In the scenario of a low hit-rate, a small data volume is generated and the shift-circuit can be operated at higher speeds.

The merger elements in the merging network each consists of a FIFO element and control logic that reads from a number of inputs FIFOs, *i.e.*, the child mergers. We vary the fan-in of the mergers in the range of $2, \ldots, 8$ elements. The mergers can only accept one tuple per clock. Thus, if more than one chips FIFO has data available, access needs to be controlled such that starvation is avoided. This is equivalent to an arbitration problem where the merger FIFO is the resource being arbitrated between the child FIFOs. Our implementations uses the well-known *round-robin token passing arbiter* [21]. It guarantees fairness (no starvation) among the input FIFOs and determines the FIFO to read in less than one clock cycle.

Our Virtex-6 chip provides FIFOs as 36 kbit block RAM (BRAM) elements. They can be configured to a maximum width of 36 bits and a depth of 1024 elements. Therefore, in order to store 96-bit result tuples three 36 kbit BRAM are required. The XC6VXL760T chip has capacity for 240 96-bit result FIFOs (either as output FIFOs of the cores or in the mergers). In the current design they represent the limiting factor with prevents us from increasing the number of cores in Figure 10.

A.5 Optimal Number of Cores

In order to obtain a high throughput for given window sizes W_R and W_S the number of join cores n needs the maximized and thereby reducing the size window partition per core. The throughput can be computed as

$$\text{throughput} = \frac{1}{\max(\lceil W_R/n \rceil, \lceil W_S/n \rceil)} \cdot f_{\text{clk}}$$

(the denominator is the maximum number of clock cycles needed to scan the local join windows; see Section A.2).

The remaining parameter that determines the overall achievable throughput is the clock frequency f_{clk} of the join circuit. In VLSI the highest possible clock frequency is determined by complexity of the circuit, *i.e.*, the number of components and density of circuit connections. It is not clear how the clock frequency depends on the core number $\leftrightarrow$ window size trade-off. Few cores have comparatively large window partitions and therefore a higher complexity inside the cores. When a large number of cores is instantiated the complexity of each core is low, while the wiring complexity between the cores is higher.

In order to measure this impact, we choose two fixed-sized global windows $W_R = W_S = 1,000$ elements. We vary the number of cores between 1–200 cores partition the global windows evenly over the cores. We then determine the highest possible clock frequency of the join circuit. For this analysis we choose a moderate timing for the result circuit of 83 MHz, resulting in an output tuple rate of 83 M tuples/sec. We synthesize each circuit for the Virtex-6 XC6VLX760T FPGA chip.

Figure 13 shows the clock frequency obtained and the resulting tuple throughput per input stream. The highest clock frequency is reached for $n = 100$ cores and 10 tuples per window and core. At this point the complexity of a core and the interconnect are in balance. When increasing the number of cores the clock frequency decreases. For 150 cores the decrease in clock frequency cannot be compensated by the speed-up of the additional processing cores, resulting in a lower throughput number than for $n = 100$ cores.