How Soccer Players Would do Stream Joins

Jens Teubner
Systems Group, Dept. of Computer Science
ETH Zurich, Switzerland
jens.teubner@inf.ethz.ch

Rene Mueller
IBM Almaden Research Center
San Jose, CA, USA
muellerr@us.ibm.com

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 semantic 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.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query Processing

General Terms

Algorithms

Keywords

stream joins, parallelism, data flow

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.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

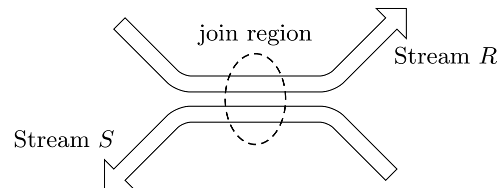


Figure 1: Handshake join idea.

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)* [5]. FPGAs were recently proposed as an escape to the inherent limitations of classical CPU-based system architectures [23, 24].

The intuition behind handshake join is illustrated in Figure 1. We let the two input streams flow by in opposing directions—like soccer players that walk by each other to shake hands before a match. As we detail in the remainder of this report, this view on the join problem has interesting advantages with respect to parallel execution and scalability.

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 (handshake join can deal with *any* join predicate, including non-equi-joins). As a side effect, our work provides a new look on the semantics of stream joins. This aspect of our work might be inspiring for ongoing efforts toward a standard language and semantics for stream processors [4, 16]. Likewise, we think that the design principles of handshake join, mainly its data flow-oriented mode of execution, could have interesting applications outside the particular stream processing problem. As such, our work can help cut the multi-core knot that hardware makers have given us.

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.

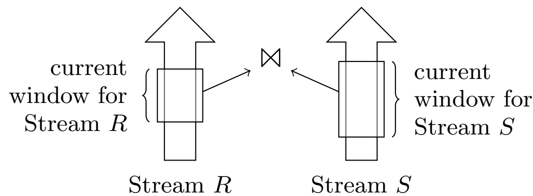


Figure 2: Window join (figure adopted from [17]).

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 semantic 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 2 (adopted from [17]) illustrates this for the case of a join operation. The join 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 2.

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. Handshake join will deal with both types of window specification equally well.

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.* [17]. 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 (t_i denote tuple arrival timestamps, T_i denote window sizes)*

(a) r arrives after s ($t_r > t_s$) and s is in the current S-window at the moment when r arrives (i.e., $t_r < t_s + T_S$) or

(b) r arrives earlier than s ($t_s > t_r$) and r is still in the R-window when s arrives ($t_s < t_r + T_R$)

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 systems demands *local* availability of data at the respective compute core, because systems increasingly exhibit *non-uniform memory access (NUMA)* characteristics, where the cost of memory access increases with the distance of an item in memory to the processing core that requests it. Such non-uniformity is contrary to the nature of the join problem, where *any* input tuple might have to be paired with *any* tuple from the opposite stream to form a result tuple.

Gedik *et al.* [10] 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 corresponding other piece of data (input tuples or the full in-memory windows).

Partitioning and replication are both rather expensive operations. The latter involves data movement which—in addition to the necessary CPU work—may quickly overload the memory subsystem as the number of CPU cores increases. A dedicated coordinator core has to *re-partition* the dataset every k input tuples (where k is a small configuration parameter), at a cost that grows linearly with the number of processing cores n [10]. Such cost may still be acceptable for the eight cores of the Cell processor, but will be hard to bear on upcoming systems with large n.

In this work, by contrast, we aim for a work distribution scheme that remains scalable even when the number of processing cores grows very large (we tested with hundreds of cores).

3. HANDSHAKE JOIN

In the most generic case, the three-step procedure of Kang *et al.* [17] 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. Only 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

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

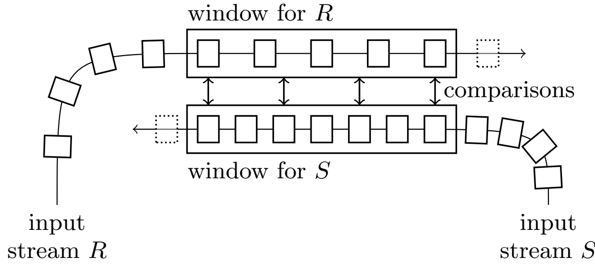


Figure 3: Handshake join sketch. Streams flow by each other in opposite directions; comparisons (and result generation) happens in parallel as the streams pass by.

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 custom 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.

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 3 (this particular illustration assumes tuple-based windows).

Tuples from the two input streams R and S, marked as rectangular boxes □, are pushed through respective join windows. When entering the window, each tuple pushes all existing window content one step to the side, such that always the oldest tuple "falls out" of the window and expires (we detail later how to implement such behavior). Both join windows are lined up next to each other in such a way that window contents are pushed through in opposing directions, much like the players in soccer (cf. Figure 3).

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.

We next look at handshake join and its characteristics from the abstract side. In Sections 4–6 we then discuss techniques to implement handshake join on different types of actual hardware, including multi-core systems and FPGAs.

3.3 Semantics

To understand the semantics of handshake join, consider the situation at moment t_r when tuple r enters its join window (illustrated in Figure 4).

A tuple from S can now relate to r in either of the following three ways (indicated in Figure 4 as s_1 through s_3):

(1) Tuple s_1 is so old that it already left the join window for input stream S, *i.e.*, $t_r > t_{s_1} + T_S$. Thus, r will not see s_1 and no attempt will be made to join r and s_1.

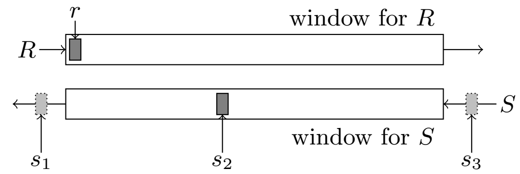


Figure 4: Handshake join semantics.

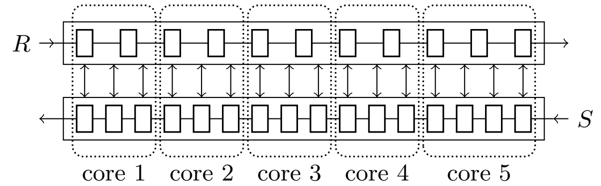


Figure 5: Parallelized handshake join evaluation. Each compute core processes one segment of both windows and performs all comparisons locally.

(2) Tuple s_2 is somewhere in the join window of S when r enters the arena, *i.e.*, s_2 is older than r ($t_r > t_{s_2}$), but still within the S-window (that is, $t_r < t_{s_2} + T_S$).

As r and s_2 move along their join windows, the two tuples are guaranteed to meet eventually, and $\langle r, s_2 \rangle$ will be added to the join result if they pass the join predicate.

(3) Tuple s_3 has not arrived yet (*i.e.*, $t_r < t_{s_3}$). Whether or not a join $r \bowtie s_3$ will be attempted depends on s_3's time of arrival and on the window specification for stream R. Once s_3 arrives, both factors will determine whether r takes a role that is symmetric to cases *(1)* or *(2)* above.

Cases *(1)* and *(2)* are the ones where r arrives after s, and a join will be attempted when $t_r < t_s + T_S$. Hence, these cases coincide with part *(a)* of the classical definition of stream join semantics (Section 2.2). Case *(3)* is the situation where r arrives earlier than s. It follows by symmetry that this case yields the same output tuples as covered by part *(b)* in Section 2.2.

In summary, handshake join produces the exact same output tuples that the classical three-step procedure would, and we can use handshake join as a safe replacement for existing stream join implementations.

Handshake join typically produces its output in a different *tuple order*. A certain degree of local disorder is prevalent in real applications already (and stream processors may explicitly be prepared to deal with it [18]). If necessary, it can be corrected with standard techniques such as *punctuations* [21].

Even though our view at window-based stream join differs significantly problem's classical treatment, handshake join still yields the same output semantics. On the one hand side, this may provide new insights into ongoing work on defining standards and abstract specifications for stream processors [4, 16]. On the other hand, handshake join opens opportunities for effective *parallelization* on modern hardware. Next, we will demonstrate how to exploit the latter.

3.4 Parallelization

Figure 5 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 [28]. Modern computing architectures increasingly tend toward shared-nothing setups. Recently, we demonstrated how data flow-oriented approaches can thus be advantageous to compute distributed joins [9] or to solve the frequent item problem on FPGAs [29].

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; the messaging primitives in Intel’s SCC prototype [14]; 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.

Soccer Players. We note that the same properties are what make the handshake procedure in soccer effective. Most importantly, soccer players organize themselves using only local interaction.

In addition, the soccer analogy exhibits the same *locality* property that is beneficial for us. While soccer players emphasize locality for the limited length of their arms, in handshake join we benefit from an efficient use of hardware with NUMA-style memory characteristics.

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 one 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 5, for instance).

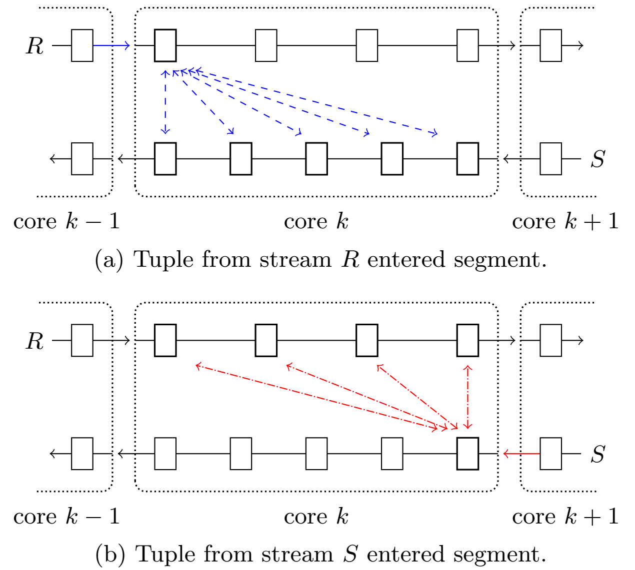


Figure 6: Immediate scan strategy. A tuple entering from stream R or S will trigger immediate comparisons ⤢ or ⤡ in the segment of processing core k (respectively).

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.

Immediate Scan Strategy. One particular strategy (which we will call *immediate scan strategy*) that can be used to process a segment k is illustrated in Figure 6. In this illustration, we assume that every tuple $r \in R$ is compared to all S-tuples in the segment immediately when r enters the segment. Figure 6(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 6(b). The strategy will operate correctly *regardless* of how window sizes relate to each other.

The latter property is interesting when it comes to the distribution of load within a handshake join configuration. In particular, it means that segment borders can be chosen arbitrarily, which can be used to very flexibly balance load between the involved processing cores. Our software implementation (details in Sections 4 and 5) autonomously re-balances segment boundaries to achieve an even load distribution over all participating cores.

Other Strategies. Observe that *locally* on each processing core, immediate scanning is essentially identical to the three-step procedure of Kang *et al.* [17]. The difference is that the procedure is used only as part of an overall join execution. Further, we allow the processing cores some flexibility to arrange with their neighbors on the precise segmentation

(*i.e.*, on the local window size). In a sense, the algorithm of Kang *et al.* is only one particular instance of handshake join that runs with a single processing unit and uses the immediate scan strategy.

One consequence is that we can plug in any (semantically equivalent) stream join implementation to act as a local processing core. This includes algorithms that use additional access structures or advanced join algorithms (as they might apply for certain join predicates; handshake join is agnostic to that). Handshake join then becomes a mechanism to distribute and parallelize the execution of an existing algorithm over many cores.

3.6 Handshake Join and its Alternatives

Among existing approaches to parallelize the execution of stream joins in a multi-core environment, CellJoin [10] clearly has been most successful. In essence, CellJoin treats stream joins as a *scheduling* and *placement* problem. A central coordinator assigns workload chunks ("basic windows" of input data) to individual cores for processing. Thereby, CellJoin assumes that all chunks have to be fetched from a global memory beforehand.

The nature of a nested loops-style join evaluation makes such a strategy difficult to scale. During join processing, every pair of tuples has to be co-located on some core at least once.[1] The *movement of data* is thus much more inherent to the join problem than the placement of CPU work or data.

This is why we address the problem from a *data flow perspective* instead. While the handshake join strategy does not necessarily reduce the aggregate amount of moved data, it leads to a much more regular and organized communication pattern. In particular, there is no hot spot that could become a bottleneck if the system is scaled up. In Section 5, we illustrate the communication pattern for a concrete piece of hardware and show how handshake join leads to linear scaling even for very large core counts.

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 communication primitives that can be used to implement handshake join on actual hardware, including commodity CPUs or FPGAs.[2]

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. For instance, 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

[1]CellJoin first *replicates* all new input tuples to every core. That is, all data is moved to every core at least once.

[2]We currently experiment with graphics processors (GPUs) as another promising platform to implement handshake join.

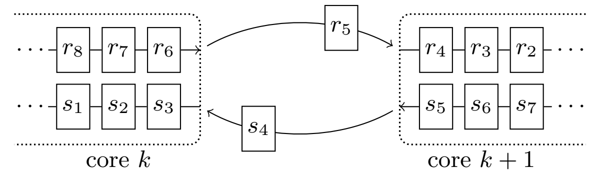


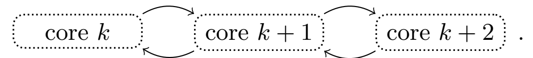
Figure 7: Missed-join pair problem. Tuples sent via message queues might miss each other while on the communication channels.

still a viable route for FPGA-based implementations, as we shall see in Section 6.2.

4.2 Asynchronous Message Queues

Handshake join can also be run in an *asynchronous communication* mode. We will now demonstrate how to implement handshake join based on *message passing* between neighboring cores, a communication mode that is known for its scalability advantages 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. Some architectures even provide explicit messaging primitives in hardware [14].

Though supportive of increased parallelism, asynchronous communication between cores can bear an important risk. As illustrated in Figure 7, tuples from opposing streams might *miss* each other if they both happen to be in the communication channel at the same time (in Figure 7, 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. To avoid the missing of join candidates, we have to make sure that in such cases, the two in-flight tuples will still meet on exactly one of the two neighboring cores, say the right one, C_{right}. To this end, we use an *asymmetric* core synchronization protocol. C_{right} (and only C_{right}) will keep around copies of sent data for short moments of time. These copies will be used to join tuples on C_{right} that would otherwise have missed each other in-flight.

More specifically, we introduce *two-phase forwarding* on the right core. Whenever the right core C_{right} places a tuple s_i into its left send queue, it still keeps a copy of s_i in the local join window, but marks it as *forwarded.* This is illustrated in Figure 8(a): tuple s_4 was sent to C_{left} and marked as forwarded on C_{right} (indicated using a dashed box).

The forwarded tuple remains available for joining on C_{right} until the second phase of tuple forwarding, which is initiated by an *acknowledgment message* from C_{left} (indicated as a diamond-shaped message in Figure 8). In the example, s_4 is still kept around on C_{right} to partner with r_5.

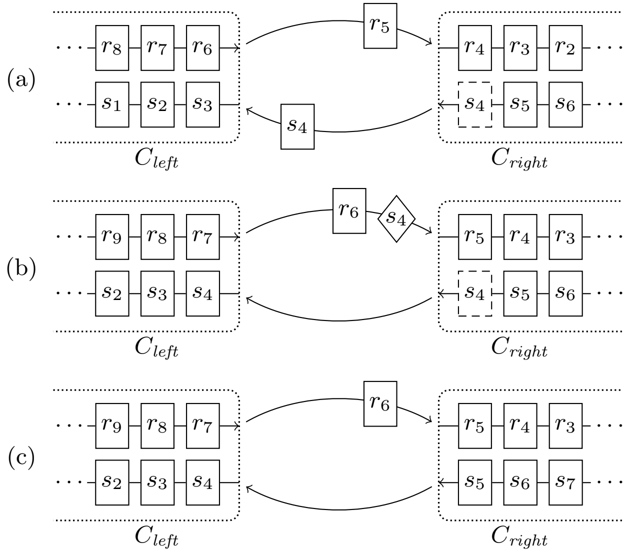


Figure 8: Two-phase tuple forwarding (on right core). Tuples are kept in C_{right}-local join window and removed only after an acknowledgment message from C_{left}.

Figure 8(b) shows the state after C_{left} and C_{right} have processed the data on their message queues. s_4 was joined with $r_6, r_7, \ldots$ on C_{left} and r_5 saw $s_4, s_3, \ldots$ on C_{right}. C_{left} further indicated its reception of s_4 with the acknowledgment message ⟨s_4⟩, then forwarded another tuple r_6 to its right neighbor.

Message ⟨s_4⟩ notifies C_{right} that any tuple sent afterward will already have seen s_4. Thus, C_{right} removes the tuple from its local join window (cf. Figure 8(c)) before accepting the next tuple r_6. This way, each necessary tuple pairing is performed exactly once (on some core).

FIFO Queues. Note that tuple data and acknowledge messages must be sent over the same message channels. In the example of Figure 8, correct semantics is only guaranteed if the acknowledgment for s_4 is received and processed by the right core *in-between* the two tuples r_5 and r_6.

Figure 9 describes an implementation of handshake join based on asynchronous message passing. This code is run 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 the three-step procedure of Kang *et al.* [17]. In addition, acknowledgment messages are placed in the queues whenever a new input tuple was accepted. The removal of tuples from the two windows is asymmetric. S-tuples are marked forwarded after sending (line 26) and removed when the acknowledgment was seen (line 15). R-tuples, by contrast, are removed from their window immediately after sending (line 29).

Our illustrations in Figure 8 suggested an explicit tuple reference in each acknowledgment message. In practice, this information is redundant, since an acknowledgment will always reference the oldest tuple in the predecessor core. This is reflected in line 15 of Figure 9, which does not actually inspect the content of the acknowledgment message.

```
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      forward_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 else
       /* msg is an acknowledgment message      */
15     remove oldest tuple from S-window ;

16 Procedure: process_right ()
17 msg ← message from rightRecvQueue ;
18 if msg contains a new tuple then
19     s_i ← extract tuple from msg ;
20     scan R-window to find tuples that match s_i ;
21     insert s_i into S-window ;
22     place acknowledgment for s_i in rightSendQueue ;

23 Procedure: forward_tuples ()
24 if S-window is larger than that of left neighbor then
25     place oldest non-forwarded s_i into leftSendQueue ;
26     mark s_i as forwarded ;
27 if R-window is larger than that of our right neighbor
   then
28     place oldest r_i into rightSendQueue ;
29     remove r_i from R-window ;
```

Figure 9: Handshake join with asynchronous message passing (ran on each core).

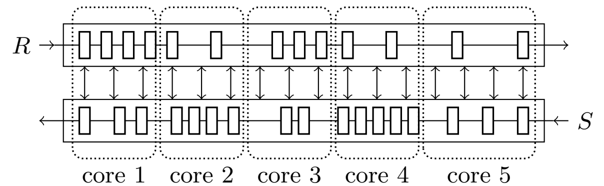


Figure 10: Data items need not be distributed evenly over processing cores for a correct join result.

4.3 Autonomic Load Balancing

We expect the highest join throughput when all available processing cores run just at the maximum load they can sustain without becoming a bottleneck in the system. To this end, load should be *balanced* over compute resources in a flexible manner and—to maintain scalability—without a need for centralized control.

The handshake join code in Figure 9 includes such an *autonomic load balancing* scheme, realized by the tuple forward-

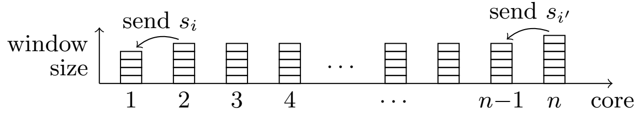


Figure 11: Tuple forwarding based on local window sizes; results in autonomous load balancing.

ing procedure `forward_tuples ()`. This scheme is based on the insight that the produced join output is *not* affected by the exact distribution of tuples over processing segments. In fact, the join output would still be correct even when the distribution was very skewed, as illustrated in Figure 10.

This gives us the freedom to forward data (*i.e.*, forward work) at an arbitrary time and we use *load* to guide the forwarding mechanism. Procedure `forward_tuples ()` implements this in a fairly simple manner, which in practice we found sufficient to achieve good load balancing even when the number of cores becomes large or when input data becomes bursty.

The idea is illustrated in Figure 11 for stream S. All processing cores have mostly even-sized local S-windows. New data is pushed in on the right end, which increases the load on core n. Tuple expiration (discussed below) removes tuples from core 1. Cores n and 2 will observe the local imbalances and send S tuples toward their left. Like water in a pipe, tuples will be pushed through the handshake join window, evenly distributed over all processing cores.

The strategy works well in homogeneous environments like ours, where load is directly proportional to window sizes. To tune handshake join for different environments, such as cloud infrastructures or virtualized hardware, all necessary code changes remain local to `forward_tuples ()`.

4.4 Synchronization at Pipeline Ends

Observe that the algorithm in Figure 9 did not explicitly mention the desired window configuration, neither the window type (tuple- or time-based) nor any window size. Instead, handshake join assumes that tuple insertion and removal are signaled by an outside driver process that knows about the precise window configuration.

Since the exact flow of tuples through the handshake join pipeline is immaterial to the join result (see above), these signals only have to be sent to the two *ends* of the processing pipeline. That is, for each input stream the driver process feeds data into one end of the pipeline and takes out expired tuples at the opposite end.

The mechanism to realize tuple insertion and removal by the driver process uses the same acknowledgment messages that helped us avoid the missed-join pair problem above. Earlier we reacted to acknowledgments by removing "forwarded" tuples from the join window. As illustrated in Figure 12, the driver process now takes advantage of this reaction and simply sends acknowledgment messages to trigger the removal of tuples at the far end of each stream window.

The generation of acknowledgment messages is straightforward to implement in the driver process. Depending on the window configuration, the driver either has to count tuples (tuple-based windows) or it must keep a list of the tuples in each window and monitor timestamps accordingly.

In Figure 9, we constructed two-phase forwarding in an asymmetric way and ignored acknowledgment messages for

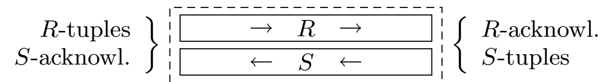


Figure 12: A driver sends tuple data and acknowledgments to opposite ends of the pipeline to realize a given window configuration.

```
17  Procedure: process_right ()
24  else
25      /* msg is an acknowledgment message      */
25a     remove oldest tuple from R-window ;

25  Procedure: forward_tuples ()
29  if R-window is larger than that of our right neighbor
    then
30      place oldest r_i into rightSendQueue ;
31      mark r_i as forwarded ;
```

Figure 13: Rightmost core runs a slightly different code to handle tuple removal via acknowledgment messages.

R (this was to make sure that in-flight tuples will be processed on C_{right} only). Messages from the driver process, by contrast, must be interpreted in a symmetric way (the join problem itself is symmetric). To this end, the *right-most* code in any handshake join setup runs a slightly different code that is symmetric to both input streams. Figure 13 lists the necessary changes, based on the `handshake_join ()` code in Figure 9.

To make handshake join sound at both ends of the pipeline, we further assume that window sizes are 0 for non-existent "neighbors." This helps to "pull" data from both input streams toward the respective pipeline end.

Explicit signals that trigger tuple removal have interesting uses also outside handshake join. In fact, they have become a common implementation technique for stream processors, referred to as $-$ *elements* (Stanford STREAM, [2]), *negative tuples* (Nile, [11]), or *deletion messages* (Borealis, [1]) depending on the system. 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 2.2 GHz AMD Opteron 6174 "Magny Cours" machine. The machine contains 48 real x86-64 cores, distributed over 8 NUMA regions. The system was running Ubuntu Linux, kernel version 2.6.32.

5.1 Non-Uniform Memory Access

The eight NUMA regions are connected through a set of point-to-point HyperTransport links, indicated as dashed lines in Figure 14 (refer to [6] for details). Observe that HyperTransport links do *not* form a fully connected mesh. Rather, two CPU cores can be up to two HyperTransport hops apart (*e.g.*, a core in NUMA region 0 and a core in

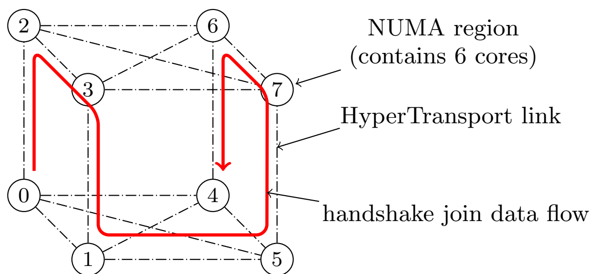


Figure 14: AMD "Magny Cours" architecture (8 dies × 6 cores). Handshake join pipeline laid out along point-to-point HyperTransport links.

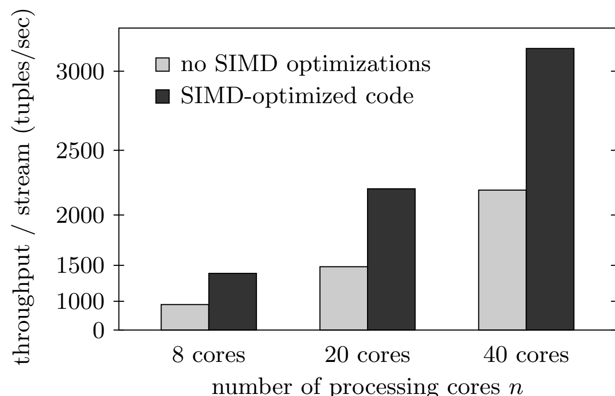


Figure 15: Effect of SIMD optimization. Use of SIMD compiler intrinsics yields a two-fold performance improvement (15 min windows).

NUMA region 3). Similar topologies have become prevalent in modern many-core systems.

Handshake join is well prepared to support a system of this kind. As indicated by the arrow in Figure 14, the data flow of the algorithm can be laid out over the available CPU cores such that only short-distance communication is needed and no congestion occurs on any link or at any NUMA site.

Our prototype implementation[3] uses the `libnuma` library to obtain a data flow as shown in Figure 14 and an asynchronous FIFO implementation similar to that of [3] (within NUMA regions as well as across).

5.2 Experimental Setup

For easy comparison with existing work we used the same benchmark setup that was also used to evaluate CellJoin [10]. Two streams $R = \langle \mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{float}, \mathsf{z} : \mathsf{char}[20] \rangle$ and $S = \langle \mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{float}, \mathsf{c} : \mathsf{double}, \mathsf{d} : \mathsf{bool} \rangle$ are joined via the two-dimensional band join

$$\texttt{WHERE } r.\mathsf{x} \texttt{ BETWEEN } s.\mathsf{a} - 10 \texttt{ AND } s.\mathsf{a} + 10$$
$$\texttt{AND } r.\mathsf{y} \texttt{ BETWEEN } s.\mathsf{b} - 10. \texttt{ AND } s.\mathsf{b} + 10. \;.$$

The join attributes contain uniformly distributed random data from the interval 1–10,000, which results in a join hit rate of $1 : 250,000$. As in [10], we ran all experiments with symmetric data rates, that is $|R| = |S|$.[4]

We implemented handshake join processing cores as Linux threads. Two additional threads asynchronously generate input data (driver process) and collect join results.

5.3 SIMD Optimization

Like most modern general-purpose processors, the AMD Opteron provides a number of *vector* (or *SIMD*) *instructions*, using which the computational density of each CPU core can be increased substantially. In handshake join, the *scan step* is a candidate for SIMD optimization. The majority of the CPU work is spent in this phase.

With the 128 bit-wide SIMD registers of our machine, up to four value comparisons can be performed within a single CPU instruction. The idea has a catch, however. In x86-64 architectures, SIMD instructions operate largely *outside* the main CPU execution unit and have only limited functionality. Most importantly, there are no *branching primitives* that operate on SIMD registers and it is relatively expensive to *move data* between SIMD registers and general-purpose registers (*e.g.*, to perform branching there).

Therefore, we modified our code to evaluate full join predicates *eagerly* using bit-wise SIMD operations (effectively the full predicate is evaluated even when one band condition was early found to fail). The positive effect on this strategy on branch (mis)prediction was already investigated by Ross [26]. Here we additionally benefit from a reduction of data movement between register sets; only the final predicate result is moved to general-purpose registers after SIMD-only evaluation.

As shown in Figure 15, vectorization using SIMD instruction yields significant performance advantages. The SIMD-optimized implementation can sustain more than double the load of the non-optimized code.

In CellJoin, Gedik *et al.* [10] used similar techniques to leverage the SIMD capabilities of the Cell SPE processors (also with substantial performance improvements). The additional low-level optimizations in [10] (loop unrolling and instruction scheduling) should be applicable to handshake join, too. Unfortunately, such optimizations make code highly processor dependent, while our current code only uses back-end-independent intrinsics to guide an optimizing C compiler (`gcc`).

5.4 Scalability

The main benefit of handshake join is massive scalability to arbitrary core counts. To verify this scalability, we ran handshake join instances with 4 to 44 processing cores (we left four CPU cores available for the driver and collector threads as well as for the operating system). For each configuration, we determined the maximum throughput that the system could sustain without dropping any data.

Figure 16 illustrates the throughput we determined for window sizes of ten and fifteen minutes (the y-axis uses a quadratic scale, since the workload grows with the square of the input data rate). The result confirms the scalability advantages of handshake join with respect to core counts and input data rates. Somewhat harder to see in Figure 16 is that, alternatively, additional cores can be used to support larger join windows while maintaining throughput.

[3]Source code can be downloaded via `http://people.inf.ethz.ch/jteubner/publications/soccer-players/`.

[4]Workload and output data rates grow with $|R| \times |S|$ or *quadratically* with the reported stream rate.

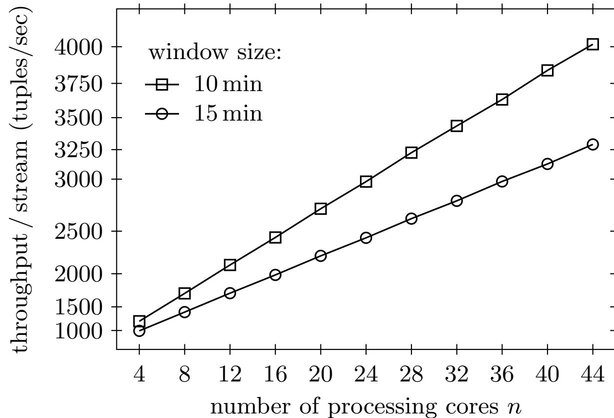


Figure 16: Handshake join scalability on multi-core CPU (AMD Opteron "Magny Cours"; immediate scan strategy). y-axis uses quadratic scale.

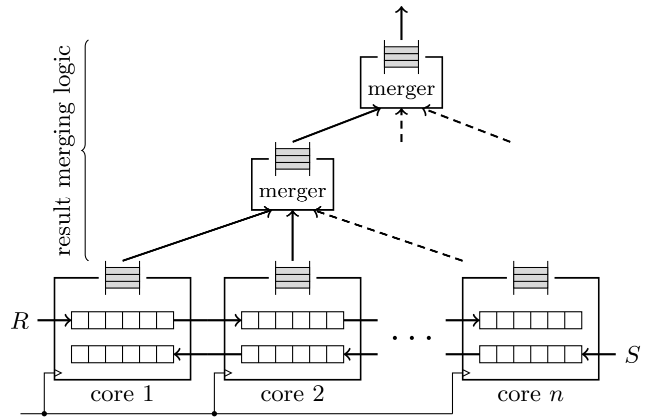


Figure 17: FPGA Implementation of Handshake Join for Tuple-based Windows.

Comparison to CellJoin. The scalability evaluation for CellJoin is limited to the eight SPEs available on the Cell processor. For 15-minute windows, Gedik *et al.* [10] report a sustained throughput of at most 750 tuples/sec when using the Cell PPE unit *plus* all eight SPEs and their "SIMD-noopt" implementation (which is best comparable to our code; with low-level optimizations, Gedik *et al.* pushed throughput up to 1000 tuples/sec). With eight CPU cores, handshake join sustained a load of 1400 tuples/sec—a 2–3.5-fold advantage considering the quadratic workload increase with throughput.

We note that the workload of this benchmark (consisting of band joins) favors an architecture like Cell. The task is very compute-intensive (including single-precision floating-point operations toward which the Cell processor has been optimized) and amenable to SIMD optimizations. On such workloads, the Cell should benefit more from its higher clock speed (3.2 GHz vs. 2.2 GHz) than the Opteron can take advantage of its super-scalar design.

For the workload sizes at hand, both join algorithms have to fetch all data from main memory during the scan phase (in 15 minutes, a stream rate of 1000 tuples/sec accumulates to around 40 MB—too large for any on-chip cache). Thereby, CellJoin inherently depends on a high-bandwidth centralized memory (25.6 GB/s for that matter). Handshake join, by contrast, keeps all memory accesses local to the respective NUMA region and thus achieves high aggregate bandwidth even when the local bandwidth is more modest (21.3 GB/s with DDR3-1333 memory).

6. HANDSHAKE JOIN ON FPGAS

48 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 an emulation platform, where the only limit to parallelism is the available chip space. FPGAs themselves are an interesting technology for database acceleration [23, 24, 29], but our main focus here is to demonstrate scalability to many cores (in particular, we favor simplicity over performance if that helps us instantiate more processing cores on the chip).

6.1 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 *configurable logic* in terms of *lookup tables* (LUTs) and *flip-flop registers* that each represent a single bit of fast distributed *storage*. Finally, a configurable *interconnect network* can be used to combine lookup tables and flip-flop registers into complex logic circuits.

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 join cores as simple as possible, we only look at *tuple-based* windows that fit into on-chip memory (flip-flop registers).

6.2 Implementation Overview

Figure 17 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 clock signal allows us to realize *lock-step forwarding* at negligible cost, which avoids the need for FIFO queues and reduces the complexity of the implementation. Effectively, the windows represent large shift registers, which are directly supported by the underlying hardware.

Following the basic handshake join algorithm (Figure 9) 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 shift registers (labeled 'R window' and 'S window', respectively) that hold the tuple data. 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 (using a standard nested-loops implementation).

Result Collection. As illustrated in the top half of Figure 17, 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 into the final join output at the top of Figure 17.

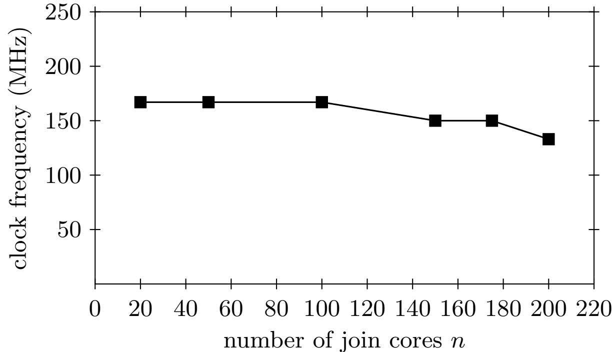


Figure 18: Scalability of FPGA Handshake join with a constant segment size of 8 tuples per window and core.

6.3 Experimental Setup

We stick to a simple stream format where join keys and payload are all 32-bit integers. We assume an equi-join and return 96 bit-wide result tuples (32-bit key plus 2×32 bits of payload).

Again our main interest is in measuring the *scalability* of handshake join. To this end, we instantiate up to 200 join cores on a Virtex-6 XC6VLX760T FPGA chip. Each of the join cores can hold eight tuples per input stream (*i.e.*, with $n = 100$ cores the overall window size will be $100 \times 8 = 800$ tuples per stream). For each configuration we determine the *maximum clock frequency* at which the resulting circuit can be operated.

In hardware design, clock frequency is an excellent indicator for scalability. In many circuit designs, the clock frequency has to be reduced according to a n^{-k} law as the circuit size is increased (larger circuit areas generally lead to longer signal paths). Only highly scalable designs allow a constant clock frequency over a large range of circuit sizes ($k \approx 0$).

6.4 Scalability and Performance

As shown in Figure 18, clock frequencies for our design remain largely unaffected by the core count (the absolute value of $150 \sim 170$ MHz is not relevant for this assessment). This confirms the good scalability properties of handshake join.

Even on the right end of Figure 18, the drop in clock frequency is not very significant. On this end our design occupies more than 96 % of the available chip area. This means we are operating our chip already far beyond its maximum recommended resource consumption of 70–80 % [8]. The fact that our circuit can still sustain high frequencies is another indication for good scalability.

By contrast, 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 [25].

6.5 Parallelism and Power Considerations

Handshake join allows to losslessly trade individual core performance for parallelism. This opens up interesting opportunities for system designers. Here we show how the flexibility of handshake join can be used to minimize the *power consumption* of an application system. More specifically, we try to find the most power-efficient handshake join configuration that satisfies a given throughput demand (which comes from the application requirements).

Table 1: Power consumption and circuit size for two windows of size 100 and a throughput of 500 ktuples/sec implemented using different numbers of cores and clock rates.

	clock	power (mW)			
cores	(MHz)	dynamic	static	total	LUTs
1	50	100	4398	4498	1.54 %
2	25	64	4396	4459	1.59 %
5	10	62	4395	4447	2.01 %
10	5	68	4306	4464	2.78 %

Table 2: Power consumption weighted by area.

	clock	weighted (mW)		
cores	(MHz)	dynamic	static	total
1	50	100	68	168
2	25	64	69	133
5	10	62	88	150
10	5	68	120	188

Again we use our FPGA framework to illustrate the concept. Assume the system has to provide a throughput of 500 ktuples/sec with a window size of 100 tuples. Configurations with 1, 2, 5, and 10 join cores can guarantee this throughput if operated at clock frequencies of 50, 25, 10, or 5 MHz, respectively. For the same throughput, however, the four configurations will require different amounts of electrical power.

Table 1 shows how core counts and clock frequencies affect the power consumption of our FPGA chip. The total power consumption of a chip results from two components: *dynamic power losses* are a result of the switching transistors (charge transfers due to parasitic capacities and short-circuit currents in CMOS technology), while *static power losses* are caused by leaking currents due to the high chip integration density.

Dynamic losses directly depend on the clock frequency of the circuit; leakage is proportional to the chip area. This can be seen in Table 1, where the dynamic component strongly decreases with the clock frequency (even though the number of cores grows). The static component, by contrast, is barely affected by the circuit configuration.

Also apparent in Table 1 is that the total power consumption is strongly dominated by the static part. This is a sole artifact of the FPGA technology that we use for this evaluation (and not indicative of the situation in custom silicon chips). In FPGAs, static losses occur on the entire chip space, even though our design occupies only 1.5 %–2.8 % of the available lookup tables (last column in Table 1). For a fair comparison with custom silicon devices, Kuon and Rose [20] suggest to weight the static FPGA power consumption by the used chip area.

Table 2 shows the weighted power consumption for our four chip configurations. Clearly, among the configurations

that we looked at, the one that uses two cores is the most power-efficient. In terms of absolute numbers, this result may be different for other architectures, in particular for realistic multi-core machines. But the ability to tune a system in this way is a general consequence of handshake join's scalability properties, with potentially significant power savings in real-world scenarios.

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* [15, 30] or window joins that use indexes [12]. If handshake join alone is not sufficient to sustain load, load shedding [27] or distribution [1] might be appropriate countermeasures.

Handshake join's data flow is similar to the *join arrays* proposed by Kung and Lohman [19]. 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 model of Kung and Lohman can also be found in *cyclo-join* [9], our own work on join processing of static data in distributed environments. In *cyclo-join*, fragments of two input relations are placed on network hosts in a *ring topology*. For join processing, one relation is rotated along this ring, such that tuples encounter each other much like in handshake join. A key difference is that handshake join properly deals with *streaming* joins, even though the semantics of such joins is highly dynamic by nature.

The only work we could find on stream joins using FPGAs is the *M3Join* of Qian *et al.* [25], which essentially implements the join step as a single parallel lookup. This approach is known to be severely limited by on-chip routing bottlenecks [29], which causes the sudden and significant performance drop observed by Qian *et al.* for larger join windows [25]. 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* [13]. 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. Our prototype implementation reaches throughput rates that significantly exceed those of CellJoin, the best published result to date [10]. Alternatively, the scalability of handshake join can be used to optimize system designs with respect to their power consumption.

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 [7] or the *Cube* 512-FPGA cluster [22]). As mentioned before, we already demonstrated the former case for static input data with *cyclo-join* [9]. In ongoing work we investigate the use of graphics processors (GPUs) to run handshake join.

Acknowledgements

This work was supported by an *Ambizione* grant of the Swiss National Science Foundation under the grant number 126405 and by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch/`).

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, T. Roscoe, 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] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. of the VLDB Endowment*, 3(1), 2010.

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

[6] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, 2010.

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

[8] 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)*, Monterey, CA, USA, 1999.

[9] P. Frey, R. Gonçalves, M. Kersten, and J. Teubner. A Spinning Join That Does Not Get Dizzy. In *Proc. of the 30th Int'l Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, 2010.

[10] 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.

[11] 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.

[12] 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.

[13] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: An 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.

[14] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. v. d. Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, February 2010.

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

[16] 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.

[17] 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)*, Bangalore, India, 2003.

[18] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous Analytics Over Discontinuous Streams. In *Proc. of the Int'l Conference on Management of Data (SIGMOD)*, Indidanapolis, IN, USA, 2010.

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

[20] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proc. of the 14th Int'l Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, 2006.

[21] 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 Int'l Conference on Management of Data (SIGMOD)*, Baltimore, MD, USA, 2005.

[22] 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.

[23] 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.

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

[25] 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.

[26] K. A. Ross. Selection Conditions in Main Memory. *ACM Trans. on Database Systems (TODS)*, 29:132–161, 2004.

[27] 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.

[28] 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.

[29] 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.

[30] 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.