Executing Stream Joins on the Cell Processor

Buğra Gedik
bgedik@us.ibm.com

Philip S. Yu
psyu@us.ibm.com

Rajesh R. Bordawekar
bordaw@us.ibm.com

Thomas J. Watson Research Center
IBM Research, Hawthorne, NY, 10532, USA

ABSTRACT

Low-latency and high-throughput processing are key requirements of data stream management systems (DSMSs). Hence, multi-core processors that provide high aggregate processing capacity are ideal matches for executing costly DSMS operators. The recently developed Cell processor is a good example of a heterogeneous multi-core architecture and provides a powerful platform for executing data stream operators with high-performance. On the down side, exploiting the full potential of a multi-core processor like Cell is often challenging, mainly due to the heterogeneous nature of the processing elements, the software managed local memory at the co-processor side, and the unconventional programming model in general.

In this paper, we study the problem of scalable execution of windowed stream join operators on multi-core processors, and specifically on the Cell processor. By examining various aspects of join execution flow, we determine the right set of techniques to apply in order to minimize the sequential segments and maximize parallelism. Concretely, we show that basic windows coupled with low-overhead pointer-shifting techniques can be used to achieve efficient join window partitioning, column-oriented join window organization can be used to minimize scattered data transfers, delay-optimized double buffering can be used for effective pipelining, rate-aware batching can be used to balance join throughput and tuple delay, and finally SIMD (single-instruction multiple-data) optimized operator code can be used to exploit data parallelism. Our experimental results show that, following the design guidelines and implementation techniques outlined in this paper, windowed stream joins can achieve high scalability (linear in the number of co-processors) by making efficient use of the extensive hardware parallelism provided by the Cell processor (reaching data processing rates of ≈ 13 GB/sec) and significantly surpass the performance obtained form conventional high-end processors (supporting a combined input stream rate of 2000 tuples/sec using 15 minutes windows and without dropping any tuples, resulting in ≈ 8.3 times higher output rate compared to an SSE implementation on dual 3.2Ghz Intel Xeon).

VLDB '07, September 23-28, 2007, Vienna, Austria.

1. INTRODUCTION

Many of today's data processing tasks, such as e-business process management, systems and network monitoring, financial analysis, and security surveillance, need to handle large volumes of events or readings that come in the form of data streams and produce results with low-latency. This entails a shift away from the "store and then process" model of DBMSs, towards the "on-the-fly processing" model of emerging data stream management systems (DSMSs) [1, 5, 7, 31]. The ability to sustain fast response time in the face of large volumes of streaming data is an important scalability consideration for DSMSs, since stream rates are unpredictable and may soar at peak times.

In this paper, we study the use of heterogeneous multi-core processors for achieving high-throughput and low-latency in windowed data stream operators. We particularly focus on windowed stream joins, which are fundamental and costly operations in DSMSs, and are representative of the general class of windowed operators. They form the crux of many DSMS applications, such as object tracking [18], video correlation [17], and news item matching [12]. Windowed stream joins are heavily employed in DAC [34], one of the reference applications we are building on top of System S [22]. DAC is a disaster assistance claim processing application, and uses stream joins in several occasions to find matching items in different but time-correlated streams.

Our discussion is based on the Cell processor [19] – a state-of-the-art heterogeneous multi-core processor. Although the Cell processor was initially intended for game consoles and multimedia rich consumer devices, the major advances it brought in terms of performance have resulted in a much broader interest and use. High-end Cell blade servers for general computing are commercially available [25], and research on porting various algorithms to Cell are under way in many application domains [4, 27].

A heterogeneous multi-core architecture is often characterized by a main processing element accompanied by a number of co-processors. For instance, the Cell processor consists of the PPE (PowerPC Processing Element) which serves as the main processing element, and the eight SPEs (Synergistic Processing Elements) which are the co-processors providing the bulk of the processing power. SPEs do not have conventional caches, but instead are equipped with local stores, where the transfers between the main memory and the local stores are managed explicitly by the application software. This is a common characteristic of heterogeneous multi-core processors, such as network processors [21] (see Section 2.2 for differences).

A major challenge in making stream joins truly scalable

is to fully analyze the execution flow, identify the potential bottlenecks, and devise solutions to remove these bottlenecks. We identify four major problems in scalable and high-performance execution of stream joins on heterogeneous multi-core processors like Cell:

P1) We need load balancing mechanisms to evenly distribute the load among the SPEs, in the presence of changes in input stream rates and system load. Unfortunately, the basic approach of evenly distributing incoming tuples among SPEs does not scale due to memory bandwidth bottlenecks (see Section 3), even though it trivially balances the load. As a result, more elaborate dynamic window partitioning schemes are needed.

P2) We need to organize the data structures maintained by the join operator (such as join windows and input batches) to facilitate low-overhead movement of data between the PPE and the SPEs. This includes minimizing the number of direct memory transfers (DMAs) used.

P3) We need techniques to mask the memory transfer delays associated with the data movements between the PPE and the SPEs. This includes overlapping processing with asynchronous data transfers, as much as possible.

P4) We need to optimize the core join code to take full advantage of data and instruction-level parallelism of SPEs.

Our work makes the following four contributions toward addressing these problems, in order to accelerate windowed stream joins using a heterogeneous multi-core processor:

S1) We provide a lightweight dynamic window partitioning mechanism to distribute the load among the eight SPEs. This is particularly important, since as the join windows continue to slide and tuples arrive and leave the windows at potentially varying rates, the segments of the join windows assigned to an SPE can change in terms of both content and size. The unique properties of our solution to this problem are two fold. First, we use basic windows to reduce the frequency of changes in the assignments of join window segments to different SPEs. Second, we develop an efficient pointer-shifting technique to quickly and incrementally adjust the SPE assignments when they change.

S2) We describe a column-oriented memory organization for maintaining the join windows. This organization minimizes the amount of data that needs to be fetched by the SPEs. More importantly, it locates the same attributes of successive tuples on contiguous regions of the memory, enabling SPEs to take full advantage of the SIMD (single-instruction multiple-data) instructions, without the overhead of data re-organization or scattered DMA transfers.

S3) We employ double-buffering techniques at the granularity of individual basic windows to mask data transfer delays. We analytically study the optimal configuration of basic window sizes to maximize the join throughput. We also develop a rate-aware dynamic tuple batching technique to balance tuple delay and join throughput.

S4) We provide optimizations at the SPE-side, targeted toward increasing the performance of the join by making use of the vectorized SIMD operations (data parallelism) and double-issued instructions (instruction-level parallelism).

Our experimental results show that, following the design guidelines and implementation techniques outlined in this paper, windowed stream joins can (*i*) achieve high scalability: linear in the number of SPEs used, (*ii*) make efficient use of the extensive hardware parallelism provided by the Cell processor: reaching data processing rates of 13.4 GB/sec at combined input stream rate of 2000 tuples/sec using 15 minutes windows and without dropping any tuples, and (*iii*) significantly surpass the performance obtained form conventional high-end processors: ≈ 8.3 times that of an SSE implementation on dual 3.2 Ghz Intel Xeon processor.

2. PRELIMINARIES

In this section, we provide basic information about windowed stream joins and present relevant details of the Cell processor architecture.

2.1 Windowed Stream Joins Overview

Since data streams are potentially unbounded, stream joins are performed over windows defined over input streams. The windows maintained over data streams can be count-based, such as the last 1000 tuples; or time-based, such as tuples from the last 10 minutes. In the case of time-based windows, size of a join window in terms of tuples is also dependent on the rates of the streams. The stream rates may not be stable and can change as a function of time. In the rest of the paper, we use time-based windows without loss of generality. Our main discussion is on nested loop-based join processing (NLJ), although we describe straightforward extensions to hash-based equi-joins as well as to the general case of multi-way (m-way) joins (see Section 7). To illustrate some of the more interesting SIMDization scenarios, we use band-joins [10]. Otherwise, our approach applies to all join conditions.

Here we summarize the operation of a windowed stream join. Let us denote the ith stream as S_i and a tuple from S_i as t_i. Streams can be viewed as unbounded sets of timestamp ordered tuples. Each stream has a specific schema. We denote the join window on S_i as W_i and the length of the join window in time units as w_i. The window lengths are parameters of the windowed join operator. We denote the timestamp of a tuple t by $\tau(t)$ and current time as $\tau(T)$. A join window keeps the tuples fetched from its associated input stream until they expire. A tuple t_i is considered as expired if and only if it is at least w_i time units old, that is $\tau(t_i) \leq \tau(T) - w_i$. The join window W_i on S_i is maintained such that we have $\forall t_i \in W_i, \tau(T) > \tau(t_i) > \tau(T) - w_i$. In other words, we have a sliding window W_i of size w_i time units over the stream S_i.

When a tuple t_i is fetched from stream S_i, it is compared against the tuples resident in the window of the opposing stream, say W_j, and join results are generated for matching tuples. After the result processing is complete, t_i is inserted into W_i and join windows are checked for expired tuples. If found, expired tuples are removed from the join windows. The join is performed both ways in alternating fashion, that is $S_i \bowtie W_j$ is performed for a newly fetched tuple t_i and $S_j \bowtie W_i$ is performed for a newly fetched tuple t_j. The particular join type we consider in this paper is the band-join, where we have one or more join conditions in the form of $X_l \leq t_i.A - t_j.B \leq X_u$. In other words, tuples t_i and t_j match if and only if the difference between attribute A of t_i and B of t_j is within the interval $[X_l, X_u]$. The band join interval can be set to $[0, 0]$ in order to represent equi-joins.

2.2 Cell Processor Overview

The Cell processor is a single-chip multiprocessor with nine processing elements (one PPE and eight SPEs) operating on a shared, coherent memory. The PPE is a general-

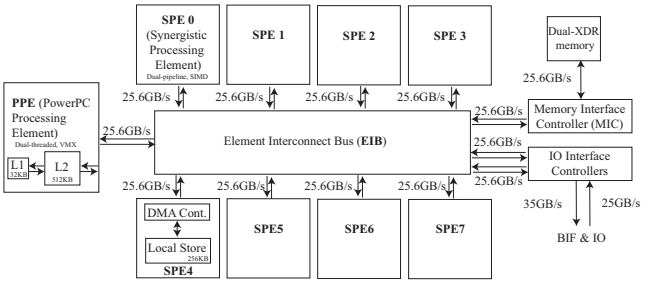

Figure 1: Architecture of the Cell processor.

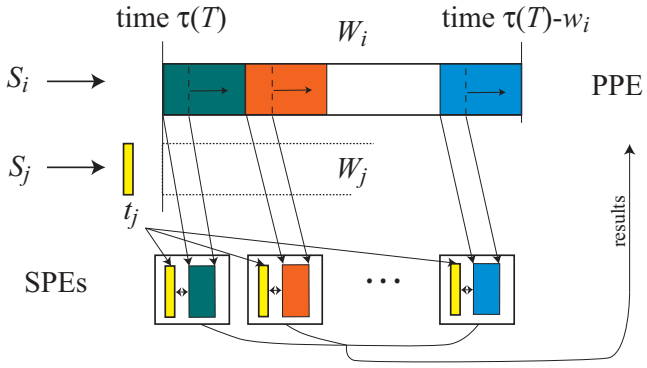

Figure 2: Illustration of program structure for a binary windowed stream join. The join processing is shown for the direction $S_j \bowtie W_i$.

purpose, dual-threaded, 64-bit RISC processor and runs the system software. Each SPE on the other hand, is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD applications. Besides data parallelism from rich set of SIMD instructions, SPEs also provide instruction-level parallelism in the form of dual pipelines, where certain types of instructions can be dual-issued to improve the average Cycles-Per-Instruction (CPI) of an application. Each SPE has full access to coherent shared memory and the memory-mapped I/O space.

A significant difference between the SPEs and the PPE is how they access the main memory. Unlike the PPE, which is connected to the main memory through two level of caches, SPEs access the main memory with direct memory access (DMA) commands. Instead of caches, each SPE has a 256 KB private local store. The local store is used to hold both instructions and data. The load and store instructions on the SPE go between the register file (128 x 128-bit) and the local store. Transfers between the main memory and the local store are performed through *asynchronous* DMA transfers. This is a radical design compared to conventional architectures and programming models, because it explicitly parallelizes the computation and transfer of data, thus avoiding the Von Neumann Bottleneck [3]. On the down side, it is programmers' task to manage such transfers and take advantage of the high aggregate bandwidth made available by the Cell architecture [26]. Moreover, SPEs lack branch prediction hardware and hence conditionals should be avoided as much as possible to keep the pipeline utilization high. Figure 1 gives a basic view of the Cell architecture.

For a typical application, the relationship between the PPE and the SPEs can be summarized as follows. The SPEs depend on the PPE to run the operating system and in most of the cases the top-level control logic of the application. On the other hand, the PPE depends on the SPEs to provide the bulk of the application performance. The PPE can take advantage of this computational power by spawning SPE threads. Such threads are not fully preemptable. In addition to coherent access to the main memory, there are several other ways for the PPE and the SPEs to communicate with each other, such as mailboxes and signals.

In brief, we want to exploit the following hardware parallelisms available on the Cell processor to scale windowed data stream processing operators, such as stream joins:

- The computational power from the eight synergistic precessing elements (SPEs) and the PPE
- Asynchronous and parallel DMAs available for high-bandwidth memory transfer
- Vectorized operations and the dual-issued instructions on the SPEs

3. STREAM JOINS ON THE CELL - DESIGN CHOICES

In this section, we describe the design choices made in implementing stream joins on the Cell processor. These choices relate to three important aspects of any program that runs on a high-performance heterogeneous multi-core processor: (*i*) How do we partition the work between the processing elements to maximize the effectiveness of parallelism?; (*ii*) How do we organize the memory to facilitate efficient transfers?; and (*iii*) How do we take advantage of the SIMD instructions?

3.1 Join Program Structure

Before discussing the fundamental issue of partitioning the join processing among the 8 SPEs, we first describe where the join windows are stored and managed in the system.

We choose to manage the join windows on the PPE-side and store them in the main memory. This is mainly because the local stores of the SPEs are limited in size (256 KB each), and not all SPEs may be available during runtime. Managing the join windows on the SPE-side will significantly limit the maximum window size that can be supported. Moreover, since stream rates may not be stable, no guarantees can be given that a given window size in terms of time length can be supported. Even though the local store sizes may increase in the future versions of the Cell processor, maintaining the join windows in the main memory is more scalable. This is because the join state is not stored on the SPE-side and thus the number of SPEs used can be dynamically changed in an SPE-transparent manner. As a result, in our design the PPE is responsible for managing the join windows. This is a lightweight job and matches well with the non-compute intensive nature of the PPE in general.

For stream joins, a unit job can be considered as fetching one or more tuples from one of the streams and matching them against the tuples in the opposite join window. This job can be parallelized in two ways, that is either by (*a*) replicating the fetched tuples to each SPE and partitioning the join window to be searched for matching tuples among the SPEs, or by (*b*) partitioning the fetched tuples among

the SPEs, and replicating the join window to each SPE.

However, option (b) has major shortcomings. First, in order to take advantage of all the SPEs with option (b), we need to fetch enough number of tuples from the input stream to ensure that the partitioning of the fetched tuples assigns at least one tuple to each SPE. This will increase the tuple delay for slower streams, especially when all 8 SPEs are used. Second, if we have a requirement that the fetched tuples have to be processed in sequence to preserve ordering, then option (b) completely fails. Finally, an even more problematic drawback of option (b) is its high memory bandwidth requirement. With option (a), a join window is transfered once from the main memory to the local stores for each unit of job, whereas this has to be done 8 times for option (b) when using all SPEs. We reach join window processing rates of around 13.4 GB/sec in our experiments, where a unit job contains 4 tuples, which corresponds to a memory bandwidth requirement of 3.35 GB/sec. Using option (b) at such processing rates would make the memory access bandwidth a bottleneck ($3.35 \cdot 8 = 26.8$ GB/sec vs. 25.6 GB/sec available, see Figure 1).

Following option (a), each SPE processes its assigned part of the join window in parallel and the load is balanced evenly, independent of the number of tuples fetched. The results consisting of the matching tuples are then collected at the PPE-side. Even though the processing of a join window can be seen as an example of embarrassingly parallel computation, the continuously changing nature of the join windows create challenges in job partitioning. Figure 2 provides an illustration of how the stream joins are structured using option (a) (window partitioning).

3.2 Column Oriented Join Windows

Since the transfers between the SPE local stores and the main memory are explicitly managed by the application, we need to make an informed decision on how to organize the memory used for the join windows. There are two basic types of memory organizations for storing tuples in the join windows, namely *row-oriented* (tuple-oriented) and *column-oriented* (attribute-oriented). Row-oriented organization is a commonly applied approach in traditional relational DBMSs for organization of tuples on the disk, whereas column-oriented organization is more commonly used for read-optimized relational databases [30].

For performing stream joins on the Cell processor, we promote the use of column-oriented memory organization. In a row-oriented approach, same attributes of different tuples are not stored within a contiguous region of memory, as opposed to column-oriented organization. Figure 3 illustrates this, in which different tuples are represented by different colors and the stream schema contains four attributes: A, B, C, and D. Assume that in this particular example one of the join conditions is on attribute B. Noting that the column-oriented organization help cluster together all the B attributes, we list the advantages of this organization compared to the more traditional row-oriented approach as:

- The SPEs can transfer only the join attributes, instead of transferring all the tuples in their assigned segment of the join windows. Even though this can also be achieved in a row-oriented architecture, it requires to gather attributes from non-contiguous regions of the memory, which can be achieved by utilizing the DMA-list operation supported by the Cell processor. However, it is more efficient to issue a DMA operation to bring a block of attributes from a contiguous region of the memory. This is illustrated in Figure 3.
- Clustering together the same attributes is crucial for the performance of the join operator on the SPE side, since the SIMD instructions, which can operate on a vector of attributes at once to significantly speed-up join processing, can only work if the attributes are on a contiguous region of the memory and can be loaded into the SPE registers without any overhead.

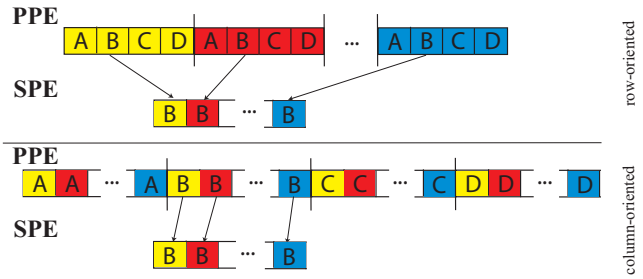

Figure 3: Illustration of row-oriented and column-oriented memory organization.

The complete details of the join window organization is given later in Section 4.

3.3 Unit blocks and SIMD

The SIMD instructions on the SPE-side operate on vectors of 128-bits, in the form of 16 `char`s, 8 `short`s, 4 `int`s/`float`s, or 2 `long`s/`double`s. For instance, a single SIMD instruction can sum 4 pairs of `int`s at once. The SPE has a rich set of such SIMD instructions that operate on vectors of various types. To take advantage of such instructions, the data has to be operated on in multiples of 128-bit vectors. As a result, we define a *unit block* as the minimum unit of data needed for vectorized join processing. A unit block includes b number of tuples, where we have:

$$b = \frac{128}{\min_{A \in \mathcal{A}} s(A)} \quad (1)$$

Let us denote the set of join attributes by $\mathcal{A}$ and the size (in bits) of an attribute $A \in \mathcal{A}$ as $s(A)$. Then the number of tuples required to fill a vector formed by A attributes is given by $128/s(A)$. When there are more that one join attributes, the minimum size one is used to compute b. Concretely, we set b such that for the minimum size attribute we have only one corresponding vector in a unit block, whereas for other join attributes we have one or more integral number of associated vectors. Noting that primitive types have sizes that are powers of 2, Equation 1 follows.

As an example, if a join has two band conditions, one on attribute A which is an `int` (32-bit) and another on attribute B which is a `double` (64-bit), then a unit block contains $128/32 = 4$ tuples, i.e., $b = 4$. In this case a unit block will contain a single vector of 4 `int`s for the A attribute and 2 vectors of 2 `double`s each for the B attribute.

4. COORDINATOR-SIDE OPERATION

In this section, we describe the major operations carried out by the PPE (as a coordinator), which include mainte-

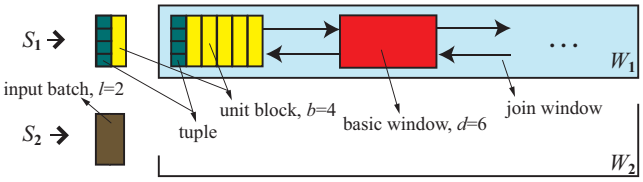

Figure 4: Illustration of join data structures.

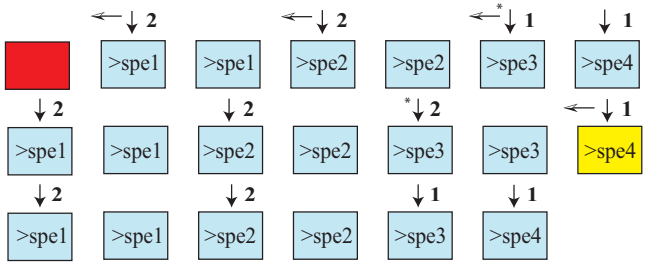

Figure 5: Illustration of dynamic window partitioning and pointer adjustments.

nance of the join windows, initiation of the join processing, and collection of join results. Key features of join window maintenance include using a lightweight, dynamic window partitioning mechanism to balance the work across multiple SPEs, while minimizing the delays due to memory transfers between the PPE and the SPEs. The unique feature of join processing initiation is the tuple batching mechanism used to maximize the throughput when input rates are high, and minimize the tuple delays when the input rates are low. The result processing is designed to allow join processing to overlap with the result transfers.

4.1 Window Partitioning

The PPE is responsible for managing the join windows. It does this by organizing each join window as a doubly-linked list of *basic windows*, where join window W_i has $B_i(T)$ number of basic windows at a given time T. Since the windows are time-based, this number is rate dependent and not fixed. However, during general mode of operation we have $B_i(T) > N$, where N is the number of SPEs used by the stream join. A basic window constitutes a single unit of transfer (from PPE to SPEs) for the join attributes, and contains a fixed-number of unit blocks, denoted by d. In other words, an SPE will transfer its assigned segment of the join window one basic window at a time. Figure 4 illustrates how the join windows are structured. On the PPE-side, a basic window contains all the attributes of the tuples it stores. When an SPE transfers the basic window to its local store, only the portion that includes the join attributes are copied (which is a contiguous region within the complete basic window). In the rest of the paper, when we refer to basic window size, we only consider the part of the basic window that contains the join attributes.

There are two motivations behind managing join windows as a set of basic windows, namely hiding memory transfer delays, and efficient job partitioning and re-adjustment.

4.1.1 Hiding Memory Transfer Delays

Basic windows can be used to hide the delays due to memory transfers initiated on the SPE side, through the use of double-buffering (see Section 5 for details). To understand this better, consider the following two extreme scenarios: In one extreme case we have as large basic windows as possible, that is one basic window per SPE and thus $B_i(T) = N$. In another extreme case we have as small basic windows as possible, i.e., $d = 1$, and thus large number of basic windows per SPE ($B_i(T)/N >> 1$). In the first scenario, an SPE has to wait for the transfer of all the join window tuples that it will process, before starting the actual join processing. This will result in a large transfer delay and will hurt the join throughput. On the other hand, the second scenario enables us to overlap the memory transfers with the processing of join window tuples through the use of double buffering. However, since the basic windows are too small, issuance of many small asynchronous DMA transfer commands will accumulate into a large overall transfer delay. Again, this will reduce the throughput. We analytically study this trade-off in Section 6 and describe how the basic window size can be set to minimize the delays due to memory transfers and achieve high throughput.

4.1.2 Dynamic Window Partitioning

Basic windows enable more efficient dynamic job partitioning, as well as more efficient tuple admission and expiration. A new basic windows is inserted into a join window only when the first basic window becomes full. Similarly, expired tuples are removed at the granularity of basic windows. As a result, at any time the first basic window is partially full, whereas the last basic window includes a mix of expired and non-expired tuples. The latter implies that the last basic window has to be time checked during join processing. The partitioning of the join windows among the SPEs needs to be changed only when the list of basic windows are updated. This happens when there is an insertion or removal of a basic window, which happen at a significantly less frequency compared to the arrival rate of tuples at the join operator. As we will describe shortly, upon such changes the job partitioning is updated in $\mathcal{O}(N)$ time.

To maintain the job partitioning, the PPE keeps N number of pointers that correspond to the first basic windows to be processed by each SPE. This is done for each join window. The PPE also keeps the number of basic windows assigned to each SPE from a join window. SPEs are assigned consecutive basic windows from the join window. Let us denote the number basic windows assigned to SPE $j \in [1..N]$ from window W_i as $c_i(j)$. Then we have:

$$c_i(j) = \begin{cases} \lceil B_i(T)/N \rceil & \text{if } j \leq B_i(T) \bmod N \\ \lfloor B_i(T)/N \rfloor & \text{otherwise} \end{cases} \quad (2)$$

The PPE is responsible for updating this partitioning when a basic window is added or removed from the join windows. When a new basic window is added to W_i, the SPE that gets an additional job can be defined as $max\{j \mid j \in [1..N] \text{ and } c_i(j-1) > c_i(j)\}$, assuming we have $c_i(0) = \infty$. Once this SPE is determined, all starting basic window pointers for SPEs with an index smaller than or equal to j are shifted one level up toward the start of the join window. Similarly, when a basic window is removed from the join window W_i, the SPE that loses a job can be defined as $min\{j \mid j \in [1..N] \text{ and } c_i(j+1) < c_i(j)\}$, assuming we have $c_i(N+1) = -\infty$. Once this SPE is determined, all starting

basic window pointers for SPEs with an index larger than j are shifted one level up toward the start of the join window. This re-adjustment procedure achieves the mentioned $\mathcal{O}(N)$ time, and is independent of the number of basic windows present in the join windows. Since N is small (8 for a single Cell processor) and re-adjustment happens infrequently, dynamic job partitioning has minimal overhead.

Figure 5 illustrates the pointer movements needed for dynamic window partitioning, for a join window of 6 basic windows partitioned among $N = 4$ SPEs. In this scenario, initially the first 2 SPEs are assigned 2 basic windows each, and the 2 remaining SPEs are assigned one basic window each (see first row), according to Equation 2. The starting basic window pointers and the number of basic windows assigned are indicated for each SPE in Figure 5. A new basic window arrives (drawn in red) and the third SPE is the one that gets an additional basic window as a result of this. This is marked with $*$ in the figure. Since SPEs are assigned consecutive basic windows, all starting basic window pointers *before or at* the marked pointer are shifted one level left to yield the new partitioning, in which the first 3 SPEs are assigned 2 basic windows, and the remaining SPE is assigned one basic window (see second row), satisfying Equation 2. The figure also shows removal of a basic window due to expiration, that is the last one (drawn in yellow). After the expiration, the third SPE is the one that loses a basic window from its partition, and again this is marked with an $*$. As a result, all the starting basic window pointers *after* the marked pointer are shifted one level left to yield the new partitioning (see third row).

4.2 Input Batching

One of the operations performed by the PPE is to initiate the join processing. This is achieved by notifying each SPE about their assigned job, using inter-processor communication. In our join operator, the PPE sends the following information to each SPE to initiate the join processing: (i) the address of the first basic window to be processed by the SPE, (ii) the number of basic windows to be processed, (iii) the address of the *input batch*, that is the input tuples to be processed against the assigned basic windows, and (iv) the size of the input batch in terms of unit blocks. The direction of the join (whether we are performing $S_i \bowtie W_j$ or $S_j \bowtie W_i$) and whether the last basic window assigned to an SPE is to be time checked are embedded into the second and fourth messages respectively, using negation. The size of the input batch is an important factor in maximizing the throughput of the join as well as minimizing the average tuple delay. We define the latter as the total time it takes for a tuple to finish its processing since the time it was fetched from one of the input streams. Throughput is the average number of tuples processed per time unit. The correct batch size to use depends strongly on the stream rates and the performance of the join operation under current rates.

4.2.1 Batch Size Trade-off

Let us analyze the average tuple delay, which is made up of two components. The first component is the time a tuple stays in the input batch, waiting for the batch to fill up. The second component is the time it takes to process a batch divided by the number of tuples in the batch. When the stream rates are high, the second component dominates, since tuple inter-arrival times will be small (shorter delay due to first component) and the join windows will contain more tuples (longer delay due to second component). Larger batch sizes minimize the overhead of tuple processing, since the memory transfer overheads are incurred once per batch. As a result, larger batches are better for achieving high throughput and low tuple delay when the stream rates are high. However, when the stream rates are low, the first component will dominate the average tuple delay. For instance, when the tuple inter-arrival time is larger than twice the time it takes to process a unit block of tuples against the join window, increasing the batch size beyond a single unit block will increase the average tuple delay. Furthermore, it won't bring any increase in the output rate (thus in throughput), since the join is already able to handle all the incoming tuples with the smallest possible batch size. In summary, a clear trade-off exists in setting the batch size.

4.2.2 Dynamic Tuple Batching

We take advantage of the batch size trade-off by performing dynamic, rate-aware tuple batching. Concretely, we keep small buffers at the inputs of the stream join operator. We check the buffer associated with a given stream to look for new tuples in order to fill the corresponding input batch. We admit as many tuples from the buffer into our input batch, such that the admitted tuples result in an integral number of unit blocks in the batch and this number does not exceed a threshold of l unit blocks. l defines the maximum batch size and is a parameter to the operator. If the buffer does not contain enough tuples to fill a single unit block in the batch, we skip the processing for this stream and go on fetching the tuples for the next stream. If we have one or more unit blocks in our batch after fetching tuples, we perform the join by notifying the SPEs, as described earlier. This results in rate-aware tuple batching. When the stream rates are low, there won't be any build-ups in the buffers and the batch will operate with a single unit block. When tuples start to accumulate in the buffers due to high input rates, dynamic tuple batching will increase the batch size.

4.3 Result Handling

After initiating the join for an input batch, the PPE waits for results and the completion of the join by all SPEs that were assigned portions of the join window. One of our aims is to overlap the result transfers and processing with the join processing. Result processing involves generating the tuples of the output stream using the matched tuples. Since the SPEs only work on the join attributes, they do not report the matching tuples directly, but indirectly through the use of match entries in the form of $\langle a, i, j \rangle$. Here a is the address of the basic window that has generated a matching tuple, i is the index of this tuple in the basic window, and j is the index of the corresponding matching tuple in the input batch. One naïve approach for reporting results is to accumulate all the match entries at the SPE-side and report them back to the PPE at the end of join processing. However, this way of result reporting introduces delays due to result processing and transfer after the join processing is complete on the SPE-side. Here we describe a technique for hiding this delay by asynchronously streaming the results back to the PPE in small batches, as the results are produced. This is especially effective when the selectivity of the join is high.

Concretely, the PPE maintains N number of result buffers, one for each SPE. Each SPE also keeps two re-

sult buffers of the same size in its local store. SPEs put their generated match entries into one of their local result buffers. When the buffer fills up, they notify the PPE (using a mailbox message) that they want to report the match entries they accumulated so far. In its notification message, an SPE includes the address of its result buffer that holds the recent match entries. The PPE uses this address to fetch these match entries into the corresponding result buffer it stores for the SPE. However, in order not to block the SPE on the transfer of the result buffer and the processing of the match entries within, the PPE dispatches the job of result transfer and processing to a result thread on the PPE-side. Before dispatching this job to the result thread, the PPE makes sure that any previous result buffers from this SPE are processed (which should not require any wait during normal mode of operation). The PPE then sends a message to the SPE in order to enable continuation of the join processing. After receiving this message, the SPE goes on with join processing by switching the result buffer it uses. The main PPE thread continuous servicing other SPEs, while the result thread works on transferring the match entries and processing results. This way, the result transfer and processing is performed by the PPE in parallel to the join processing performed on the SPE-side.

5. CO-PROCESSOR-SIDE OPERATION

In this section, we describe the SPE-side operation, which includes the core join processing functionality. The key features of join processing include minimizing the delay due to memory transfers, taking advantage of SIMD instructions to expedite processing, and optimizing the join code for taking advantage of instruction-level parallelism.

5.1 Join Processing

As we have mentioned earlier in Section 4.1, the join processing on the SPE-side involves double buffering to hide the delays due to memory transfers. Concretely, the SPE first fetches the input batch and the first basic window that is assigned to it, using a single DMA-list transfer. The join processing can start after this transfer completes. The DMA request for bringing the next basic window is issued, before the first basic window is processed for finding matches against the tuples in the input batch. Note that each basic window includes the address of the basic window following it in the join window. By the time the processing of the first basic window completes, the next basic window should have arrived and the processing can continue after the DMA request for the third basic window is issued. The process continues in this manner by keeping two basic window buffers. While one of the buffers is being searched for matches, the other buffer is filled in from the main memory in parallel. In summary, the delay due to memory transfers consists of the time needed to bring in the initial basic window (together with the input batch) and the cycles spent for issuing asynchronous DMA requests for the following basic windows.

Even though we pay the penalty of waiting for a memory transfer only once, most of the times this can be avoided by caching. Concretely, each SPE can cache the first basic window they retrieved from their assigned segment of a join window, together with the address of the basic window. If a new join processing request over a join window has the same first basic window address with the last processing request on the same join window, then there is no need for fetching the first basic window again and the cached one can be used to start the processing immediately. Only exception to this is the case where the first basic window assigned to an SPE is the first basic window of the join window. In this case, the number of active tuples in the basic window may have increased since the last processing was performed on the join window, whereas the basic window address stayed the same. This is because, the first basic window of the join window may be partially full at any time. We solve this problem by sending the number of active tuples in the first basic window to be processed by an SPE, as part of the join processing initiation parameters. This way, an SPE can detect if it is assigned the first basic window of the join window and if so can fetch only the new tuples.

The described caching mechanism is very effective for binary joins, because the first basic window to be processed by an SPE does not change unless a new basic window is added or removed from the join window. Unfortunately, this caching mechanism does not extend to hash-based equi-joins, as it will be later discussed in Section 7. Moreover, the local store size may become a limiting factor for using it in m-way joins. Because of these, we do not use this caching mechanism in our experiments.

5.2 Taking Advantage of SIMD

We now describe how SIMD instructions can be used to speed-up the join processing. For illustration purposes, we will consider a band join condition in the form $X_l \leq t_i.A - t_j.B \leq X_u$ and will assume that the attributes A and B are both `floats`. As a result, a 128-bit vector contains 4 join attributes. Furthermore, let us assume that we have A attributes in our input batch and B attributes in our current basic window. The join is performed in an NLJ fashion, but using SIMD instructions. For the current attribute in the input batch, we first create a 128-bit vector that contains the exact same attribute value multiple times. In the running example, we will create a vector of 4 identical `floats` using the current A attribute of the input batch. Let us denote this vector by v_a. We iterate over the current basic window that contains the B attributes to find tuples matching with the current A attribute. This iteration is performed in a vector at a time manner. Let us denote the current vector we are processing from the current basic window as v_b. Each v_b vector we process includes B attributes belonging to 4 different tuples. We use one SIMD instruction to subtract v_a from v_b to get 4 differences in one result vector, say v_d, and two SIMD instructions to compare v_d against the boundary conditions X_l and X_u to get 2 vectors of 4 comparison results each, say v_l and v_u. We use one SIMD instruction to logically and v_l and v_u to yield a vector, say v_r, that contains 4 results indicating whether we have a match or no match for the 4 B attributes we had in the v_b vector. In summary, 5 SIMD instructions are used to compute the result of four matchings, whereas the same processing requires 16 instructions without the SIMD support. Figure 6 gives a summary of the SIMD supported join computation. The match entries described earlier in Section 4.3 are generated using the result vector v_r and the indexes of matching tuples in the input batch and the basic window.

5.3 Optimizing the Join Code

We can further improve the performance of the join code at the SPE-side by considering the following two properties

initially	$a = a_1, v_b = \langle b_1, b_2, b_3, b_4 \rangle$
SIMD replicate a	$v_a = \langle a_1, a_1, a_1, a_1 \rangle$
SIMD subtract v_a from v_b	$v_d = \langle \cdots, a_1 - b_i, \cdots \rangle$
SIMD compare v_d with X_l	$v_l = \langle \cdots, a_1 - b_i \geq X_l, \cdots \rangle$
SIMD compare X_u with v_d	$v_u = \langle \cdots, X_u \geq a_1 - b_i, \cdots \rangle$
SIMD and v_l and v_u	$v_r = \langle \cdots, X_u \geq a_1 - b_i \geq X_l, \cdots \rangle$

Figure 6: Set of SIMD instructions used to compare a single attribute against a vector of attributes.

of the hardware:

- The SPEs can issue certain instructions on their two pipelines in parallel to improve the performance, by reducing the average number of cycles it takes to execute an instruction, that is the CPI of an application.
- The SPEs do not have branch predictors and thus loops with small inner bodies are expected to hurt performance due to branch related pipeline stalls. This should be avoided as much as possible.

In consideration of these properties, we unroll the inner NLJ loop and perform the join processing for 8 vectors worth of attributes from the current basic window during one iteration. The load and store instructions can be dual-issued with the floating point instructions used to compute matching join attributes. For instance, loading the second vector of attributes from the basic window can be done in parallel with performing the subtraction of the input batch attribute vector from the first vector of attributes that are already loaded. Loading/storing 8 vectors from/to the basic window to/from the registers within the body of the inner loop allows the optimizing compiler of the Cell processor [11] to improve the CPI performance of the application by increasing the number of dual-issued instructions. Unrolling also reduces branching overhead due to the inner NLJ loop. The loop can be unrolled in larger numbers with diminishing returns. Fixed number of registers (128 for SPEs) limits the benefit from further unrolling. The match entry computation part of the join processing requires one branch per compared tuple, and thus the overall CPI of the join is not minimal in general. Ideally, the CPI of an application can be as low as 0.5, at which point all instructions are dual-issued.

6. BASIC WINDOW SIZE

In this section, we provide an analytical model for finding the optimal basic window size, which minimizes the time it takes to process join windows at the SPE-side, including the time spent for memory transfers that are not performed in parallel. In other words, our aim is to find the best setting for d, the number of unit blocks within a basic window.

Let us denote the average rate of the stream on which our join window is maintained as λ. Then the average number of basic windows we have within the join window is given by $L = (\lambda \cdot w)/(b \cdot d)$, recalling that b denotes the number of tuples in a unit block and w denotes the length of the join window. The size of a single basic window is given by $H = d \cdot b \cdot s(\mathcal{A})$, recalling that $s(\mathcal{A})$ is the size of the join attributes. The time it takes to process a join window on the SPE-side can be divided into two components: (i) time spent on memory transfers, and (ii) time spent on join evaluation. The time needed to transfer a memory block of size X bytes can be modeled as $\alpha_m + \beta(X)$, where α_m is the time needed to issue the DMA commands and $\beta(X)$, which is a non-linear function (see [26] for Cell DMA performance), gives the time spent for actually moving the data. Then the time spent due to the first component is given by $\alpha_m \cdot L + \beta(H)$, since we wait for the transfer of the basic window (of size H) only once, whereas the time to issue a DMA request is incurred for every basic window in the join window (L times). The time needed to evaluate the join for one basic window can be modeled as a linear function (see Section 8 for validation of this assumption) $\alpha_n + \beta_n \cdot X$, where α_n represents the fixed per-basic window overhead and $\beta_n \cdot X$ represent the time spent for matching tuples in a basic window of size X bytes. Then the time spent due to the second component is given by $L \cdot (\alpha_n + \beta_n \cdot H)$. Note that this is equivalent to $L \cdot \alpha_n + \lambda \cdot w \cdot s(\mathcal{A}) \cdot \beta_n$ and the part dependent on d is given by $L \cdot \alpha_n$. As a result, smaller basic windows increase the join evaluation overhead.

The Cell architectures limits the size of DMA transfers to 16 KBs. On the other extreme, it does not make sense to make DMA transfer smaller than 128 bytes, which is the cache line size and the DMA cost is fixed up to the cache line size. As a result, we have $d^* \in 2^x, x \in [7..14]$, where d^* denotes the optimal number of unit blocks within a basic window. We have:

$$d^* = \underset{d \in 2^{[7..14]}}{argmin} \; (\alpha_m \cdot L + \beta(H)) + L \cdot (\alpha_n + \beta_n \cdot H) \qquad (3)$$

For simplicity, assume that the function $\beta(X)$ is linear and has the form $\beta_m \cdot X$. Then solving Equation 3 gives the following closed form:

$$d^* = \sqrt{\frac{\alpha_m + \alpha_n}{\beta_m} \cdot \frac{\lambda \cdot w}{b^2 \cdot s(\mathcal{A})}} \qquad (4)$$

Equation 4 shows that optimal basic window size increases with increasing stream rates and window sizes. We further study this in Section 8.

7. DISCUSSIONS

In this section, we discuss how our approach can be extended to hash-based equi-joins and multi-way joins, and how resource-adaptation can be performed to handle changing load conditions within a single Cell processor.

7.1 M-way Stream Joins

An m-way stream join can be processed in MJoin [33] fashion by taking a given input tuple through m-1 number of join windows, following the join order associated with the tuple's source stream. To port this to Cell, we use a slightly different strategy to balance the load across the SPEs than the one used for binary joins. In particular, we maintain N partitions over each join window as usual, but we only distribute one of the join windows across the N SPEs during the processing of an input batch. The join window to distribute depends on the processed input batch and the join order for that batch. Concretely, the first join window in the join order associated with the input batch is partitioned among the SPEs. For instance, if the join order for stream S_1 is S_3, S_2, S_4 for a 4-way join (we are performing

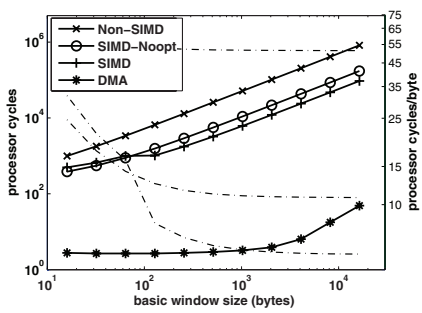

Figure 7: Processor cycle performance of join w./w.o. optimizations.

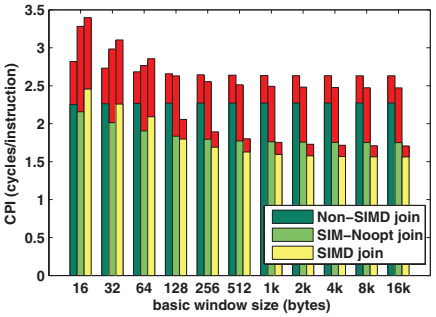

Figure 8: CPI (Cycles per Instruction) performance of join w./w.o. optimizations.

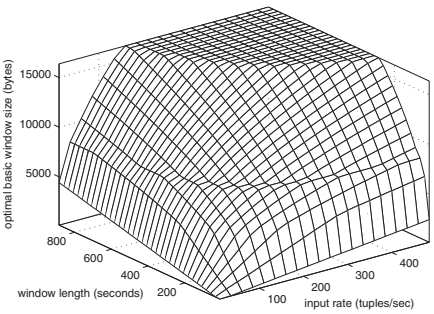

Figure 9: Optimal basic window size (analytical estimate).

$S_1 \bowtie W_3 \bowtie W_2 \bowtie W_4$), then only window W_3 is partitioned among the SPEs when processing an input batch from S_1. When a match is found by an SPE in its assigned segment of the partitioned join window, it fully processes the next join window (W_2), and in the case of further matches the following join window (W_4). This procedure partitions the load evenly across the SPEs.

7.2 Hash-based Equi-joins

Windowed stream joins with equality conditions can be evaluated using a symmetric hash-based join algorithm [23]. Our approach easily extends to such hash-based joins. Instead of managing each join window as a linked list of basic windows, we maintain it as a set of hash buckets, where each hash bucket is a linked list of basic windows. As a result, the PPE maintains partitioning of the basic windows among the N SPEs for each hash bucket, instead of each join window. Since a new tuple can affect only one hash bucket, there is no additional computational complexity brought by maintaining a dynamic partitioning for each hash bucket. For a hash-based join, an SPE will only process the basic windows in its assigned segment of the hash bucket corresponding to the current tuple. One subtle issue is that, since it is highly likely that the successive tuples will hash into different buckets, the caching of the first basic window assigned to an SPE is no more an effective optimization.

7.3 Resource Adaptation

So far we have assumed that the number of SPEs used, N, is fixed. In practice, we can dynamically change the number of SPEs used based on the join performance under the current input stream rates. This is especially useful when additional operators are being run on the same Cell processor, which can potentially use the free SPEs. The rate at which the tuples arrive at the join operator and the rate at which they are processed can be dynamically compared to see whether additional SPEs are needed. We can start the join by using a single SPE. In the case that the tuple processing rate is lower than the input rate, the join is forced to randomly drop some of the tuples and this can be avoided by bringing in more computing power in the form of additional SPEs. Conversely, the join can decide to free an SPE if the tuple processing rate is sufficiently high relative to the input stream rate. However, this kind of adaptation can not be performed too frequently, since there is an overhead associated with spawning a new SPE thread from the PPE, assuming there is a free SPE available to use by the join operator. Furthermore, addition of a new SPE necessitates re-adjustment of the join window partitioning. Unlike re-adjustment due to addition or removal of basic windows, this requires time proportional to the number of basic windows, instead of number of SPEs. This may also contribute to the adaptation overhead, in case the join windows contain large number of basic windows.

8. EXPERIMENTAL EVALUATION

In this section, we present a set of experimental results to study the performance of the proposed stream join operator on the Cell processor. We make use two types of platforms in our experimental study. One of them is an IBM Cell processor clocked at 3.2 Ghz. We perform our experiments using up to 8 SPEs. The other is IBM Full-System Simulator for the Cell processor [20], which makes it possible to measure the CPI of SPE-side join code. We divided the set of experiments into two categories, namely the SPE-only study and the general study. In the SPE-only part, we mainly study the performance of the core SPE code used to match a given input batch against the tuples in a basic window. These experiments are performed on the Cell system simulator. In the general study part, we focus on the performance of the overall join operator with respect to two metrics: average output rate and average tuple processing time. These experiments are performed on the Cell processor.

In our experimental studies, we compare the following three approaches on the Cell processor.

- **Non-SIMD** represents the stream join without the use of vectorized operations on the SPE-side.
- **SIMD-Noopt** represents SIMD-enhanced join, but without the loop unrolling.
- **SIMD** represents the stream join as it is described in this paper, with all enhancements.

We also compare the performance of stream joins on the Cell processor against a dual 3.2 Ghz Intel Xeon processor (see Section 8.2.2 for details).

8.1 SPE-only Study

The experiments reported in this section use a band join on a single `float` attribute. We focus on the performance

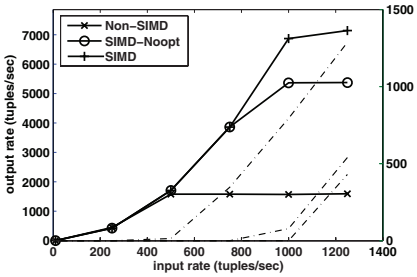

Figure 10: Scalability w.r.t. to input stream rates.

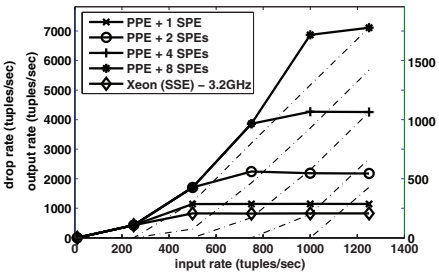

Figure 11: Impact of the number of SPEs used on join output rate.

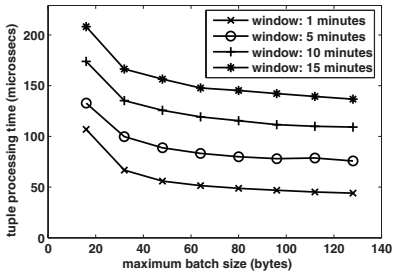

Figure 12: Impact of the input batch size on average tuple processing time.

of the core join processing performed on a single SPE using a single basic window of varying size.

8.1.1 Running Time of Join Core

Figure 7 plots the number of cycles (on the left y-axis, read from solid lines) and the number of cycles/bytes (on the right y-axis, read from dashed lines with same order of solid lines) it takes to perform the join on the SPE-side using an input batch of one unit block and for a basic window of given size (on the x-axis, in bytes). Besides the three alternative approaches mentioned earlier, Figure 7 also plots the processor cycles (but not cycles/bytes) spent for bringing a basic window into the local store of an SPE from the main memory via a DMA transfer[1]. Note that both the x-axis and the y-axis are in logarithmic scale. We observe that the non-SIMD join has around 10 times the processing cycles of the optimized SIMD join and the SIMD-Noopt join has around 1.8 times the processing cycles of optimized SIMD join. The latter difference is more visible from the cycles/bytes graphs read from the right y-axis. The flat part on the DMA graph represents the basic window sizes smaller than or equal to 128bytes (the cache line size). For such very small basic window sizes SIMD join has a slight overhead compared to SIMD-Noopt, since there is not enough data to perform loop unrolling. However, basic window sizes smaller than 128 bytes are not of practical importance due to the high memory transfer overheads they incur during a complete scan of the join windows.

8.1.2 CPI of Join Core

Figure 8 plots the average CPI (the smaller the better) for executing the join over a basic window of given size for the three approaches. The higher bars in Figure 8 (drawn in red) show the effective CPI, which excludes the *noop* instructions that do not perform actual work. We observe from Figure 8 that SIMD join has better CPI values compared to Non-SIMD and SIMD-Noopt join, after basic window size exceed 128 bytes. In particular, Non-SIMD join has up to 46% higher CPI compared to SIMD join. Even though the CPI values of SIMD join and SIMD-Noopt join seem to be close, their effective CPI values show significant difference. In particular, SIMD-Noopt has 45% higher effective CPI compared to SIMD join, whereas for Non-SIMD join this number is 54%. Note that the best CPI the SIMD join can achieve is observed to be around 1.5, which can be considered relatively high compared to the ideal value of 0.5. This is due to the branchy nature of the join code, which has to check the result of the join condition evaluation for each processed tuple to detect results.

[1]Results on DMA are reported based on actual hardware.

8.1.3 Optimal Basic Window Size

Figure 9 plots the optimal basic window size (on z-axis, in bytes) as a function of the input stream rate and join window length, based on the analytical results given in Equation 3 of Section 6. The join processing and DMA constants (α_n, β_n, α_m, and function β) used to generate this result are calculated using the results from Figure 7. We make two important observations from the figure. First, even though high stream rates and large join windows necessitate larger basic windows, there is a wide spectrum of input stream rates and join window lengths for which the optimal basic window size is significantly lower than the maximum supported size of 16 KB. Second, the flat shape of the top part of the surface shown in Figure 9 implies that there are also stream rates and window lengths for which the 16 KB maximum basic window size is a limiting factor in terms of performance.

8.2 General (PPE+SPEs) Study

In the experiments presented in this section, we consider a binary join $S_1 \bowtie S_2$. The two input streams have the following schemas: $S_1(\texttt{int}\, X, \texttt{float}\, Y, \texttt{char}(20)\, Z)$ and $S_2(\texttt{int}\, A, \texttt{float}\, B, \texttt{double}\, C, \texttt{bool}\, D)$. We define two band conditions for this join, one on the corresponding `int` attributes (X and A) and another on the corresponding `float` attributes (Y and B) of the two streams. As a result, the unit block tuple count is equal to 4, i.e. $b = 4$. The join attributes for both of the streams are generated randomly from the interval $[0, 10000]$ and a threshold range of $(-10, 10)$ is used for both of the band conditions. The default stream rate used is $\lambda = 500$ tuples/sec and the default window length used is $w = 15$ minutes. We use $d = 256$ as the default unit block count for the basic windows, which results in 8 KB basic windows. By default, tuple batching is turned off ($l = 1$), although we experiment with varying maximum batch sizes. Unless it is stated otherwise, we use all of the 8 SPEs for join processing.

8.2.1 Scalability w.r.t Input Stream Rate

Figure 10 plots the output rate of the join (on the left y-axis, using solid lines) as well as the rate at which tuples are dropped (on the right y-axis, using dashed lines) as a function of the input stream rate (on the x-axis), for different approaches. Note that, when the join cannot cope up with the incoming stream rates some tuples are dropped at the

join inputs. The lines capturing the drop rates follow the order of the legend entries, from top to bottom. We observer that SIMD join supports up to 250% higher join output rate compared to No-SIMD join and 33% better output rate compared to SIMD-Noopt join. The SIMD join is able to operate without droping any tuples from the input stream, up to input rate of 1000 tuples/sec. This number is around 500 tuples/sec for No-SIMD join and around 750 tuples/sec for SIMD-Noopt join.

Given input stream rates λ_1 and λ_2, join window lengths w_1 and w_2, and join attributes of size $s(\mathcal{A})$, the amount of data processed per time unit for a stream join is given by $\sum_{i=1,2} \lambda_{3-i} \cdot (\lambda_i \cdot w_i \cdot s(\mathcal{A}))$. Note that the term within parenthesis is the amount of data in a join window, which is processed for each incoming tuple from the opposite stream. As a result, the first term is the rate for the opposite stream of a join window. From Figure 10, we can see that at 1000 tuples/sec with 8 SPEs there is no drop and thus the join is able to reach a data processing rate of 13.4 GB/sec based on this formulation.

8.2.2 Impact of Number of SPEs (N)

Figure 11 plots the output rate of the join (on the left y-axis, using solid lines) as well as the rate at which tuples are dropped (on the right y-axis, using dashed lines) as a function of the input stream rate (on the x-axis), for different number of SPEs. All of the cases additionally make use of the PPE. The PPE not only handles join window management and result processing, but also generation of input streams as part of the experimental setup. For comparison purposes, performance of the stream join on a dual 3.2 Ghz Intel Xeon (5060 Dempsey) processor is also shown (the implementation on the Xeon uses SSE acceleration).

We make two important observations from the figure. First, the stream join shows perfect scalability with the number of SPEs, since the maximum output rate supported by N SPEs is around 2 times that of $N/2$ SPEs. This attests to minimal overhead of dynamic job partitioning among the SPEs. Second, we observe around 8.3 times higher output rate compared to the Intel Xeon processor, when all the SPEs are used. This attests to the high computational power of the Cell processor, as well as the effectiveness of our proposed techniques in exploiting this power to maximize the performance of windowed stream joins.

8.2.3 Impact of Input Batch Size

Figure 12 plots the average tuple processing time (in microseconds) as a function of the maximum input batch size, for different lengths of join windows. We make three observations from the figure. First, larger batch sizes reduce the average time it takes to process a tuple. This is because memory transfer overheads are incurred once per batch. Second, and more importantly, with increasing batch sizes the rate of reduction in the average tuple processing time decreases. It is observed that increasing the batch size beyond 128 bytes brings insignificant gains. Third, with increasing window lengths the reduction rate in average tuple processing time slightly decreases. However, this effect is limited. Going from 1 minute join windows to 15 minutes join windows, the tail of the average tuple processing delay graph is still almost flat around 128 bytes. This shows that the maximum number of unit blocks in the input batch, i.e., parameter l, need not be large. Since we use dynamic input batching, we can achieve both high throughput and low tuple delay (which includes the waiting time in the input batch) with a small maximum input batch size.

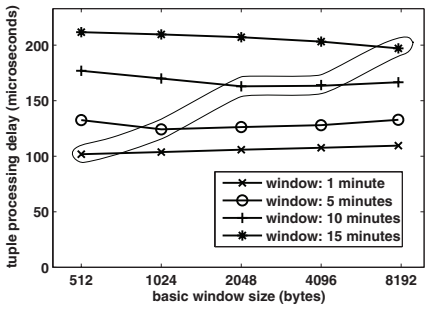

Figure 13: Impact of basic window size on average tuple processing time.

8.2.4 Impact of Basic Window Size

Figure 13 plots the average tuple processing time (in microseconds) as a function of the basic window size, for different join window lengths. Note that the x-axis is in logarithmic scale. We observe that for shorter join windows smaller basic windows provide better performance, whereas for longer join windows larger basic windows achieve better performance. The minimum average tuple processing times for different window lengths are connected to show this effect. This observation is in line with our intuition from Section 4.1 and the analytical study presented in Section 6. From Figure 13, we also observe that using a sub-optimal basic window size can cause up to 10% increase in average tuple processing time.

9. RELATED WORK

Performing joins on data streams has been actively studied in recent years [23, 13, 18, 28]. The costly nature of stream join operations and the stringent response time requirements of data stream management systems has created further interest in accelerating stream joins operators. Two lines of research emerged in this direction, namely load shedding and hardware support in the form of co-processors.

In DSMSs, load shedding [32] refers to reducing the load incurred by the system, while keeping the quality or the quantity of the output high. In the context of stream joins, this is usually done by selectively dropping some of the input tuples or selectively processing only a subset of the join windows [8, 2, 12]. Load shedding is often effective when the processing capacity of the system is not sufficient handle the processing demand of the data stream operators.

A different and new approach is to use co-processor hardware to increase the system's capacity, instead of shedding load. Most of the previous work on this topic deals with either graphics processors [16] or network processors [14]. So far the main focus of these works has been traditional database operators (such as joins [14], spatial range queries [6], sorting [15], and others [16]), with some extensions to stream mining in the case of graphics processors [16]. Graphics processors (GPUs) are characterized by high bandwidth access and low-latency (by way of sequential access) to texture memory. However, the programming model of GPUs is somewhat limited, mainly due to the lack

of random access writes.

Network co-processors on the other hand, such as Intel IXP2400 [21], are more similar to the Cell architecture considered in this paper, in the sense that they enable a more generic programming model compared to GPUs. Similar to SPEs, network co-processors also do not have hardware-managed memory hierarchy. A network processor like Intel IXP2400 has 8 micro-engines, where each micro-engine can execute multiple threads. In [14], multiple-threads per micro-engine is used to hide the memory transfer latencies, in the context of relational joins (not stream joins). In contrast, an SPE can hide the memory transfer latencies through the use of asynchronous DMAs and avoid context switching overhead. In general, compared to network processors, the Cell architecture is more powerful and capable, with data and instruction-level parallelism provided by each SPE. This makes it a good choice for low-latency and high-throughput data stream operations.

Parallel joins in relational DBMSs has been extensively studied in the past for shared nothing architectures [29, 9, 24]. In this paper, we study parallel stream joins in DSMSs and describe concepts and techniques to scale them on the Cell processor − a good example of emerging multi-core, heterogeneous architectures.

10. CONCLUSION

In this paper, we developed concepts and techniques to execute windowed stream joins, on high-performance heterogeneous multi-core processors in a scalable manner. We demonstrated the effectiveness of our approach on the IBM Cell processor. We showed that column-oriented memory organization and dynamic window partitioning enable us to better exploit the resources provided by the co-processors in a heterogeneous architecture. We developed techniques such as delay-optimized double buffering, rate-aware dynamic batch processing, and SIMD-optimized join code, that together lead to high throughput and low latency processing. Our experimental results show that, following the design guidelines and implementation techniques outlined in this paper, windowed stream join operators can achieve high scalability by making efficient use of the extensive hardware parallelism provided by heterogeneous multi-core processors like Cell and significantly surpass the performance obtained from conventional high-end processors.

11. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[2] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *ACM SIGMOD*, 2004.

[3] J. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21(8), 1978.

[4] D. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the Cell processor: A case study on list ranking. In *IEEE IPDPS*, 2007.

[5] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal*, 2004.

[6] N. Bandi, C. Sun, A. E. Abbadi, and D. Agrawal. Hardware acceleration for spatial selections and joins. In *VLDB*, 2004.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *ACM SIGMOD*, 2003.

[9] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database machine. In *VLDB*, 1986.

[10] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, 1991.

[11] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for a Cell processor. In *PACT*, 2005.

[12] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *ACM CIKM*, 2005.

[13] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.

[14] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *International Workshop on Data management on New Hardware*, 2005.

[15] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *ACM SIGMOD*, 2006.

[16] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *ACM SIGMOD*, 2005.

[17] X. Gu, Z. Wen, C. Lin, and P. S. Yu. ViCo: An adaptive distributed video correlation system. In *ACM Multimedia*, 2006.

[18] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *SSDBM*, 2003.

[19] IBM. Cell broadband engine architecture. Technical Report Version 1.0, IBM Systems and Technology Group, August 2005.

[20] IBM full-system simulator for the cell broadband engine processor. http://www.alphaworks.ibm.com/tech/cellsystemsim/, October 2006.

[21] Intel. IXP2400 network processor hardware reference manual. Technical report, Intel Corporation, May 2003.

[22] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *ACM SIGMOD*, 2006.

[23] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *IEEE ICDE*, 2003.

[24] M. S. Lakshmi and P. S. Yu. Limiting factors of join performance on parallel processors. In *IEEE ICDE*, 1989.

[25] Mercury Systems Dual Cell-based Blade. www.mc.com/literature/literature_files/Cell_blade-ds.pdf, Feburary 2007.

[26] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor inter connection network: Built for speed. *IEEE Micro*, 26(3), 2006.

[27] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *IEEE IPDPS*, 2007.

[28] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, 2004.

[29] M. Stonebraker. The case for shared nothing architecture. *IEEE Database Engineering Bulletin*, 9(1), 1986.

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

[31] Streambase systems. http://www.streambase.com/, May 2005.

[32] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[33] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of m-way join queries over streaming information sources. In *VLDB*, 2003.

[34] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. Technical Report RC24199, IBM Research, 2007.