Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 963539, 12 pages
doi:10.1155/2011/963539

Research Article

A Streaming High-Throughput Linear Sorter System with Contention Buffering

Jorge Ortiz[1] and David Andrews[2]

[1] *Information and Telecommunication Technology Center, The University of Kansas, 2335 Irving Hill Road, Lawrence, KS 66045, USA*
[2] *Computer Science and Computer Engineering, The University of Arkansas, 504 J. B. Hunt Building, Fayetteville, AR 72701, USA*

Correspondence should be addressed to Jorge Ortiz, jorgeo@ku.edu

Received 28 July 2010; Accepted 15 January 2011

Academic Editor: Aravind Dasu

Copyright © 2011 J. Ortiz and D. Andrews. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Popular sorting algorithms do not translate well into hardware implementations. Instead, hardware-based solutions like sorting networks, systolic sorters, and linear sorters exploit parallelism to increase sorting efficiency. Linear sorters, built from identical nodes with simple control, have less area and latency than sorting networks, but they are limited in their throughput. We present a system composed of multiple linear sorters acting in parallel to increase overall throughput. Interleaving is used to increase bandwidth and allow sorting of multiple values per clock cycle, and the amount of interleaving and depth of the linear sorters can be adapted to suit specific applications. Contention for available linear sorters in the system is solved through the use of buffers that accumulate conflicting requests, dispatching them in bulk to reduce latency penalties. Implementation of this system into a field programmable gate array (FPGA) results in a speedup of 68 compared to a MicroBlaze processor running quicksort.

1. Introduction

Sorting is an essential function for many scientific and data processing applications. Extensive research has optimized multiple software sorting algorithms for general-purpose computing, thereby increasing application performance. The need for higher performance has also motivated the migration of sorting algorithms into specialized hardware to exploit spatial parallelism. Nonetheless, many of the assumptions made to increase performance on a general-purpose processor do not hold for custom hardware implementations. Thus, reconfigurable applications typically do not enjoy the benefits of software-based sorting algorithms. When directly translated into hardware, software algorithms can quickly degrade into a series of data retrievals, comparisons, swaps, and writes; all problems that can be magnified in systems with low processor speeds, limited storage, disabled caches, and high-latency memory access times.

A conventional sorting system involves data acquisition and collection, processing, dynamic and long-term storage, and sorting and dispatch. Many hardware approaches use linear sorting to keep a sorted list with in-order insertions but fail to optimize throughput, the rate at which data elements are processed. The system's sorting throughput is limited by the weakest link, which in many cases is the sorting stage. Even though parallelism can speed up hardware-based sorting, sorted results are generally attained only one at a time. Thus, a hardware-based linear sorter would only achieve a maximum throughput of one sorted output per clock cycle, regardless of the system's available and exploitable parallelism. Additionally, specialized hardware like priority schedulers might have heterogeneous data processing and arrival times, which need to be considered for the system to work in a pipelined fashion. Pipelined systems are not able to start the next stage before completing the current one, so stalling on a stage potentially halts the pipeline's progress until data becomes ready. This property essentially invalidates other hardware sorting approaches like Bitonic sorting networks [1, 2], which are able to achieve high-throughput only when acting on fully available

data sets. To achieve low-latency pipelining and increase throughput through parallelism in a sorting system, a novel sorter implementation is needed.

In this paper, we present an alternative approach for linear sorters that solves the previously identified problems by

(i) expanding the linear sorter implementation and making it versatile, reconfigurable and better suited for streaming input and output (Section 3),

(ii) parallelizing the linear sorter for increased throughput for scenarios with random and uniform distribution of streaming data (Section 4),

(iii) minimizing latency costs by buffering of contending parallel sorting requests (Section 5),

(iv) implementing the high-throughput linear sorter and outmatching the performance of current linear sorter approaches (Section 6).

Additional features have been added from our work in [3] to further reduce sorting latency using a contention buffering scheme. Moreover, we provide a detailed explanation of our interleaved linear sorter system implemented as a component of a superscalar reconfigurable processor core. The innovative features of our interleaved approach fill the need for a high-throughput pipelined sorting unit, whose adaptability makes it particularly useful for streaming and reconfigurable systems.

2. Background

Several popular sorting algorithms (e.g., quicksort, mergesort, and heapsort) use divide-and-conquer techniques to achieve efficiency [4]. Intuitively, one would assume they are suitable for a parallel hardware implementation. Regrettably, upon breakdown to a register-transfer level representation, these algorithms are plagued with data movement, synchronization, bookkeeping, and memory access overhead. The sorting speed is highly dependent on a fast and robust computing platform, the type of platform that is inadequate for mobile, embedded, real-time, low-power, or reconfigurable systems.

Hardware sorting makes extensive use of concurrent data comparisons and swaps each clock cycle, rather than relying on the sequential execution of multiple assembly operations like its software counterpart. Due to its parallelism, a hardware implementation can speed up sorting applications, even at lower clock frequencies. Hardware comparisons can occur simultaneously on multiple pairs of elements. A first approach involves making multiple concurrent small operations in comparators that cascade into a well-structured network and is called a sorting network. Inserting serial input to a systolic array of sorter cells provides a second approach to hardware sorting. The third and final approach involves a single large parallel computation over multiple independent nodes and is called a linear sorter.

2.1. Sorting Networks. Sorting networks use parallel wires and multiple levels of swap comparators to shuffle data into a sorted output array of wires. Each swap comparator consists of two data inputs A and B and their two outputs H and L, where $H = \max(A, B)$ and $L = \min(A, B)$. This component can be used recursively to construct a sorting network S_N, capable of sorting N data inputs. Simple implementations for sorting networks use the same principles as well-known sort algorithms. Figures 1(a) and 1(b) show two common implementations, where wires are represented by horizontal and swap comparators by vertical lines. Figure 1(a) resembles a bubble sort: swap-comparators first sink the lowest value to the bottom wire and then operate on the remaining S_{N-1} network recursively. Figure 1(b) acts like an insertion sort, assuming a sorted network S_{N-1} already exists and then using swap comparators to position the remaining input accordingly. While simple, both sorting networks suffer from latency with the worst case scenario having $\mathcal{O}(N)$.

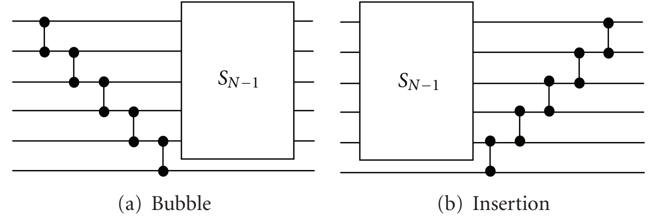


Figure 1: Sorting networks of size N.

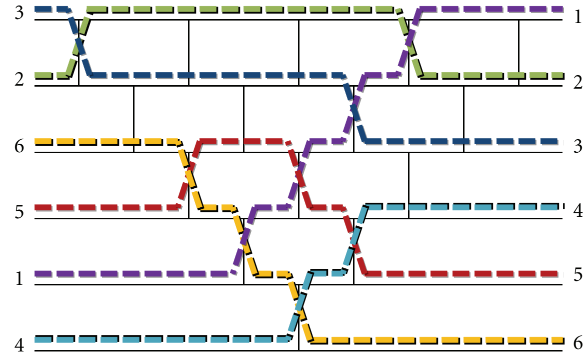


Figure 2: Parallel sorting network of size 6.

The recursive structure of the bubble and insertion networks is identical but inverted. Collapsing them to allow parallel comparisons results in an identical sorting structure. Figure 2 displays this resulting structure sorting a sample set of data of size 6 with 15 comparators. Sorting latency is reduced through parallel comparisons occurring simultaneously on multiple pairs of wires. As the size of the network grows, the number of comparators increases in a quadratic manner. In general, for a network of size N, the amount of comparators required will be $N \times (N-1)/2$.

Merging networks are more efficient than insertion or bubble networks, which are impractical due to their large depth. These networks keep two lists of ordered data and produce a single ordered list after merging. Batcher was the first to propose sorting networks, by introducing the odd-even mergesort and bitonic sort networks [1]. The odd-even

mergesort network sorts all odd and even data in separate lists before merging, while bitonic networks first sort values in monotonically increasing and decreasing lists and then merge the lists into a single sorted sequence. These types of sorting networks are common in network crossbars for asynchronous transfer mode (ATM) switching, usually in the form of a Batcher-Banyan switch [5–7].

The main advantage of sorting networks is their ability to receive a parallel input block of data, rather than feeding input data serially. However, the detriments include large numbers of processing elements (PE), high latency, and the need to resort a full set of data upon a single new insertion. Martínez et al. reduced sorting network latency by introducing levels of pipelining [8], while Tabrizi and Bagherzadeh introduced a tree-like structure to reduce the area and PE complexity [9]. Nevertheless, sorting networks in general cannot efficiently handle progressive incoming data, and their large area and complexity may hinder implementation [10].

2.2. Systolic Sorters. Systolic arrays, which are a matrix arrangement of processing cells, can be structured to sort serial input. This approach was first proposed by Leiserson [11] in the form of a systolic priority queue. Queue data is fed serially into the systolic system. It traverses the length of the priority queue going forward then traverses it again in reverse direction towards the output. When two data elements, traversing in opposite directions, meet in a queue cell, the one with the maximum value keeps traversing in reverse (towards the output), while the minimum is sent back to the forward path. A special case of the systolic priority queue occurs if all inputs are loaded before starting queue extraction. The priority queue then becomes a systolic sorter [12], but the system requires a systolic array length equal to the number of all input elements to be sorted.

Each cell in the systolic sorter becomes a processing element with two inputs and two outputs. Figure 3 shows this basic cell structure, with `Input_F` and `Output_F` aligned with the forward direction and `Input_R` and `Output_R` with the reverse path. On collisions where the two inputs have valid data, `Output_R` = max(`Input_F`, `Input_R`) and `Output_F` = min(`Input_F`, `Input_R`), which consequently advances the maximum values towards the reverse path culminating on the priority queue's output. The cell's regular structure is an important consideration for VLSI systolic arrays. Without it, this cell is essentially a registered swap comparator.

The structure of the systolic sorter is a linear array of processing cells. Figure 4 demonstrates the regular structure of a priority queue with five cells, with the forward path heading right, the reverse path pointing left, and the right-most cell reversing the traversal paths. Registered outputs, rather than combinatorial logic, are a key factor of the cell and the priority queue functionality. Serial data is sent to the systolic sorter's input every other clock cycle, allowing comparisons between adjacent forward and reverse queue values, which would otherwise bypass each other.

The extra clock cycle needed per input for a systolic priority queue of length N decrees the sorting latency accordingly. $2N-1$ cycles are needed to feed input data and output the first sorted value, then another $2(N-1)$ cycles are needed to flush the remaining $N-1$ sorted values into the output, for a total latency of $4N-3$ clock cycles. The systolic sorter depth then is just N cells, an improvement over a sorting network's $N \times (N-1)/2$ required swap comparators, at the expense of increased sorting latency.

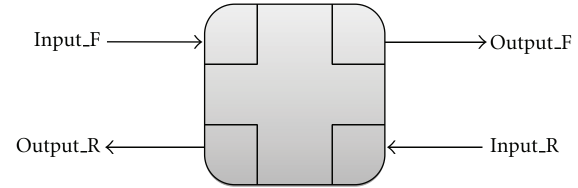


Figure 3: Systolic sorter cell interface.

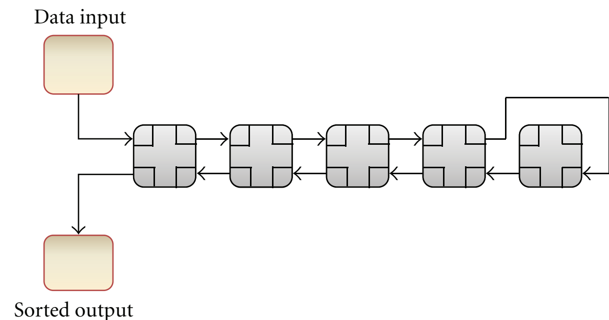


Figure 4: Systolic sorter array structure.

The size and latency requirements for sorting networks and systolic sorters are an impediment for implementation. Large data sets demand $\mathcal{O}(N^2)$ swap comparators in sorting networks and $\mathcal{O}(N)$ cells in systolic sorters, putting a strain on the system's area requirements. However, systolic sorters can be implemented as components of sorting architectures than can handle large or infinite streaming data sets and provide constant-time sorting to fixed-size sorting systems for arbitrary size data [13]. Furthermore, they can be employed to manipulate cost-performance tradeoffs in hybrid sorting systems that decompose sorting into sequential and parallel parts [14].

Modified systolic sorters that support large data sets nonetheless suffer from large sorting latencies, since incoming values must still traverse the length of the systolic sorter twice. While appropriate for offline sorting, these large latencies create obstacles for streaming data, an encumbrance that is compounded with the extra multicycle input rate of systolic sorters. By contrast, linear sorters, the third approach to hardware sorting, allow for single-clock insertion and single-clock sorting latency.

2.3. Linear Sorters. Linear sorters, which keep a sorted list while inserting new elements in-order, are an alternative approach in hardware-based sorting. The principle is the same as insertion sort in a software-based linked list: new inserted data is positioned at the corresponding place in the sorted list, thus keeping the list sorted. Although in

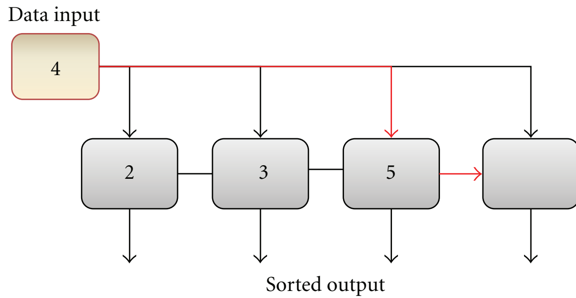


Figure 5: Linear sorter node configuration.

hardware one can iteratively traverse every node in the list when inserting values [15], a more appropriate method is to send incoming values to all nodes in parallel. Near-neighbor interconnections exist among nodes and their left and right neighbors, which allow each sorting node to make an autonomous decision about its new value and positioning in the list. This decision process involves each node acquiring a new value by comparing the incoming value against the node's current value and that of its neighbors. Although the underlying hardware structure of the linear sorter is unchanged, each node and its corresponding value can be thought of as shifting right, shifting left or holding its current value in the context of the sorted list. Figure 5 shows a linear sorter and the interconnections between its nodes. The data input (with a value of 4) is propagated to all nodes, forcing the third one (with a value of 5) to "shift" its value to the right, allowing the reception of the newly inserted value on the sorted list.

Linear sorters have small logic and control footprints, regular structures, and relatively straightforward hardware implementations using shift registers. Additionally, these sorters are particularly appropriate for streaming data, where low sorting latency and continual sorting is crucial. The main drawback of linear sorters is the serial nature of both their inputs and outputs. Even though the output of a linear sorter can be accessed in parallel (as depicted in Figure 5), at most, one value can be erased from the queue during continuous operation of streaming data. The fixed size of a linear sorter adds extra restrictions as it limits the size of growing lists before depleting node availability. When the sorter is full, new inserted values must replace old ones. This can be done using a FIFO that outputs the Top N results of a list [16]. Alternative approaches augments the incoming data with an associated tag that indicates either an insertion or a deletion [17]. Such a linear sorter must additionally allow nodes to shift values left in addition to shifting right, holding, and inserting. Furthermore, it is also possible to sort on tags rather than on the data itself [18]. This approach is useful when implementing priority schedulers, or for preserving the order of data with identical tags. A final approach is to use a linear merge sorter, where two FIFO sorted queues are merged into a single sorted queue through linear sorters [19]. Regardless of the approach, with only a serial output, linear sorters are confined to an output rate of one value per clock cycle, limiting the overall throughput of the system.

Both sorting networks and linear sorters can capitalize on increased throughput to improve their performance. Sorting networks can be pipelined to increase sorting throughput but at a high area cost due to their depth. Regardless, sorting networks still suffer from a latency of $\mathcal{O}(\log^2 N)$ and are still unable to handle data streams [20]. Linear sorters, on the other hand, have a single clock cycle of latency and reduced area but by default are unable to produce increased throughput. We propose an extension to linear sorters which effectively increases their throughput by using parallel linear sorters and interleaving logic.

3. Configurable Linear Sorter

The core of our high-throughput sorting system is based on a linear sorter. Each insertion is broadcast to all nodes, where each node either holds its value, receives the input value, or shifts right. Reconfiguration was a key property to provide versatility and adaptability; hence, the size of the data D, the depth of the linear sorter N, and the sorting direction are configurable parameters of the sorting system. We further extend the linear sorter with tags, so that sorting is performed on the tags rather than the data, as in [18]. Because the tags also have a variable bit length, an application can specify both the size of the data and the tag. The benefits are twofold: the linear sorter minimizes area consumption, while adapting to the application's specific requirements and the logic delay for the comparison operation is reduced for tags smaller than the data width. In case the sorting must depend on the data itself, the value of the tag can be replaced with the data.

3.1. Linear Sorter Functionality. Due to the finite nature of our linear sorter depth N, by default, only the Top N results are kept. If sorting greater than N values is required, the replacement policy can be augmented with external logic that checks for full conditions. When the sorter has only one free node available and an insertion occurs, an output signal bit is set. This enables the external input feeding logic to start buffering rather than discarding new input data.

The output of the linear sorter can be accessed in parallel by retrieving multiple node values at once. This is useful for systems which process data in batches, since the linear sorter depth N can be set to match the batch size. Once an input batch is received and processed, the linear sorter can be reset to start the process again for a new input batch. For continuous operation, the output must be serial, the top value must be erased, and the queue must be informed of this action. To accomplish these properties, an additional external signal makes a request to delete the top value while retrieving it, thus freeing up nodes. This operation is akin to the `pop()` operation used in standard FIFO queues and stacks.

Because the leftmost value is deleted, the node's functionality needs to be enhanced to include a left shift, in addition to the default operations shift right, hold, and insert. Figure 6 shows the added functionality for each node of the linear sorter while sorting nine values, with the vertical

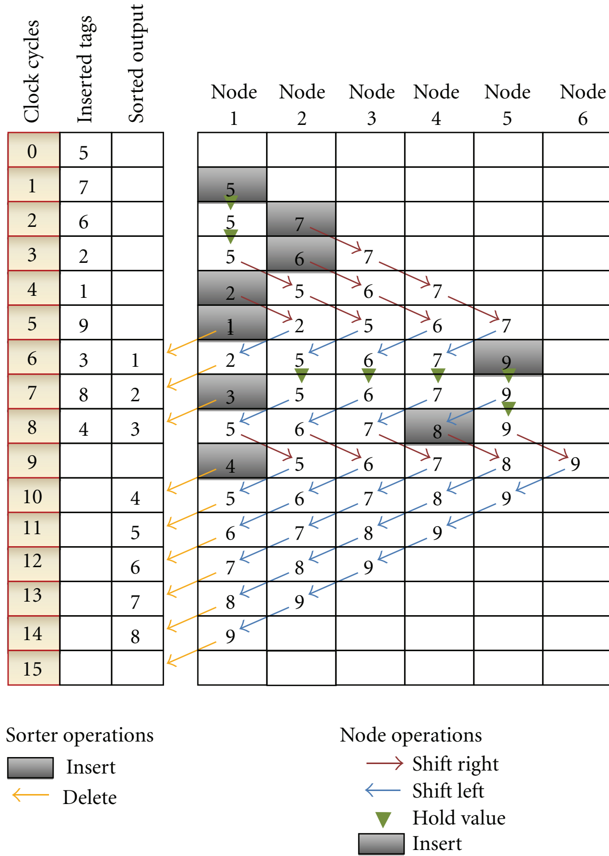


Figure 6: Sample execution of modified linear sorter.

axis indicating clock cycles. During each clock cycle, a subset of the active nodes either shift left, shift right, hold, or insert.

The output logic for this sample linear sorter deletes contiguous tags (and their associated data). Once the first tag is received, the output logic will keep deleting as it retrieves data with incrementally contiguous tags. When it stops deleting, the linear sorter continues to receive input, and deletion resumes when the next contiguous tag is in the top of the queue. The gap in the sorted output column of Figure 6 indicates a pause in delete requests from the output logic. The worst case scenario, where the first tag is inserted last, does not affect the sorting rate, but it does increase the latency of results.

To achieve continuous processing of streaming input, the external delete signal is kept at a constant high after accumulating enough input data. This setup creates a priority queue, where the local maxima is always deleted from the top of the queue. However, an external delete signal adds the advantage of testing a tag or value before deletion. Consequently, the linear sorter waits until certain conditions are met rather than just deleting the current top value (which might have been superseded in subsequent insertions). The example in Figure 6 shows an inactive deletion during clock cycle 9, to ensure a predetermined order. Had the deletion been constant, the next top tag value would have been $\{5\}$ instead of $\{4\}$, producing the output $\{1, 2, 3, 5, 4, 6, 7, 8, 9\}$. This feature is important if either

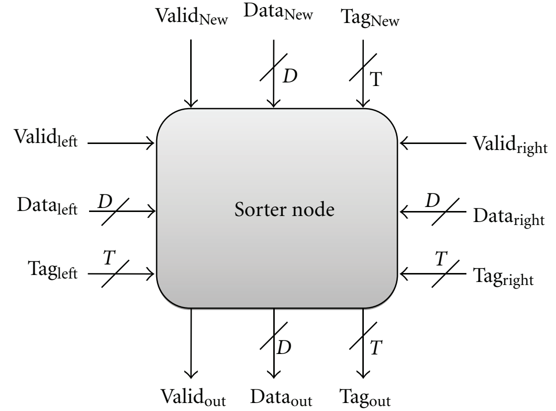


Figure 7: Sorter node interface. D = Data Width, T = Tag Width.

continuity is necessary (like pixels on an image) and/or data values cannot be discarded. A sample situation is processing instructions by integer, floating point, and other functional units with different latencies back into program order using linear sorters. In general, ordered data on any heterogeneous parallel processing system that needs to be reordered must have its values wait until previous sorted data has been dispatched. To ensure this, tags can be initially assigned incremental values before entering out-of-order execution stages.

Careful consideration must be used with the external delete signal to avoid deadlock problems. The linear sorter should be configured with a depth N large enough to allow a sliding window for contiguous values. If, for example, the system can guarantee that no two contiguous data values can arrive more than M clock cycles away, then the minimum value of N should be $M + 1$. This also specifies the maximum latency of sorted results.

3.2. Sorter Node Architecture. To achieve the linear sorter functionality we described, each sorting node implements the appropriate interface and logic. The sorter node interface has three sets of inputs and a set of outputs, each set containing signals for validity, tag, and data, as shown in Figure 7. The widths of the data (D) and tags (T) can be specified as a parameter during synthesis. The three input sets come from the left neighbor, the right neighbor, and the linear sorter insertions being forwarded to all nodes. The node's output set is sent to both its left and right neighbors, so that they too can make sorting decisions autonomously. This architecture coincides with those of single linear sorters [16, 20], which stores (key, data) pairs and compares them against neighbors and incoming values.

The decision logic to shift, insert, or hold a node value is a function of these three sets of inputs, specifically, the tags and validity bits. Shifting decisions are summarized on Table 1. Isolated insertions (row 1) right-shift all tags smaller than the new tag. An isolated deletion (row 2) always left-shifts every node. A left-shift can also occur while both inserting and deleting if the new tag is greater than the node's right neighbor (row 3); otherwise, the node inserts the new tag

if that tag is also greater than its own (row 4). For isolated insertions, nodes will also insert the new tag if it falls between its left neighbor and its own tag (row 5) or if the new tag is added at the end of the queue (row 6). Otherwise, the nodes keep their current tag and data values.

The sorter node has registers of length D, T, and 1 for the data, tag and validity bit, respectively. Three less than comparators test the inserted tag value against both neighbors' and the node's tags. Finally, additional Boolean gates are used to assess the input validity and the operations being requested.

4. Interleaved Linear Sorter System

The throughput of a linear sorting system is limited to one input per clock cycle and can be improved if parallel inputs are used. Multiple linear sorters are configured in parallel to create an interleaved linear sorter (ILS) to overcome this throughput limitation. An ILS system distributes parallel inputs to the linear sorters, trying to minimize stalling from full conditions through even distribution. Assuming random or incrementally contiguous tag numbers, we can use interleaving to equally distribute the values among all linear sorters. Typically, this is done by setting the ILS width, W (the number of linear sorters), to a power of two, which reduces the interleaving modulo arithmetic to simple bit trimming. Despite the average even distribution of this method, there is the distinct possibility that two or more tags will match the same sorter, which requires preemption and buffering. All inputs are then serviced by the hardware in round-robin fashion, without keeping state of which inputs were serviced in the previous clock cycle. When contention for a linear sorter occurs, only the first conflicting request is serviced. The interleaving controller sets an output signal to indicate this input stalling, and the data stream can resume after the buffer, using registers common in pipelined systems.

Figure 8 shows a sample execution of an ILS. The ILS width $W = 4$, indicates the quantity of linear sorters (LS) acting in parallel. The numbering i for the ILS's four linear sorters, LS0, LS1, LS2, and LS3, indicates its interleaving value. Hence, a tag value T will be sent to linear sorter i if $T \bmod W = i$. During the first clock cycle of this execution, each of the parallel inputs tags (and their corresponding data values) are dispatched to different linear sorters. During the second clock cycle, only the first of two clashing tags $\{5, 1\}$ is serviced. The remaining one $\{1\}$ is dispatched in the next clock cycle, while the input is stalled. Even though the same situation happens again during clock cycle 5, all 16 values are sorted after only 6 clock cycles and are then ready to be sent to the outputs at a parallel rate of W. In this particular example, the outputs are ready to start streaming in contiguous order as early as the fifth clock cycle.

The ILS registers the values for multiple conflicting tags and dispatches them over several clock cycles. This solves possible situations in which three or more tags match the same LS. For instance, an input of $\{0, 4, 8, 12\}$ would cause this situation for the example in Figure 8. During these worst case scenarios, the throughput advantages of an ILS degrade gracefully to that of a single linear sorter system. Nevertheless, additional area is required to implement W linear sorters and their input logic for the benefit of the average and best case scenarios. In these cases, the ILS shows a distinct advantage in throughput over a typical linear sorter system.

16 total values to be sorted — # Linear sorters = 4

= preempted value — Linear sorter depth = 4

Figure 8: Sample sorting of 16 values by 4 linear sorters.

Table 1: Logic for sorter node's shift operation.

	Operation	Delete	Insert	NewTag less than LeftTag	MyTag	RightTag
1	Right Shift	No	Yes	Yes		
2	Left Shift	Yes	No			
3	Left Shift	Yes	Yes			No
4	New Value	Yes	Yes	No	No	Yes
5	New Value	No	Yes	No	Yes	
6	New Value	No	Yes	Yes	N/A	

4.1. Input Distribution and Latency. The average latency for sorting W values arriving in parallel will vary predictably if we assume a uniformly distributed set of tags. With $W = 1$, there are no conflicts, and the single tag is always processed in one clock cycle. When $W = 2$, there is a 50% chance that the second tag's interleaving value will be the same as the first one. This condition adds an extra clock cycle of latency 50% of the time, therefore giving an overall average of 1.5 clock cycles for $W = 2$. For larger values of W, we used Monte Carlo simulations rather than relying on deterministic algorithms, due to the increasingly complex dependencies with previous tag's interleaving values. Table 2 shows the results of 2^{30} simulations, for $W \in \{1, 2, 4, 8, 16\}$. Because of the added clock cycle latency, throughput results obtained for different ILS widths are normalized over their average latency. This ensures that the calculated ILS throughput is adjusted to account for the additional contention introduced by the parallel linear sorters.

The previous analysis is dependent upon a uniform distribution of tags among the W linear sorters. This is

Table 2: Average latency for interleaved linear sorter of width W.

Number of linear sorters W	Clock cycles
1	1.000
2	1.500
4	2.125
8	2.597
16	3.078

easily accomplished if the tags to be sorted are effectively random (from an interleaving perspective), or if they have been assigned incrementally contiguous or evenly distributed numbers. On the other hand, nonuniform distributions require additional custom input logic. The input logic to interleave multiple values to the linear sorters plays an important role in determining the overall throughput of an ILS. As the ILS width W increases, so will the logic and routing delay, diminishing the overall clock frequency of the system as we will demonstrate in Section 6. In terms of area, the depth of each linear sorter in an ILS can be set independently to accommodate different types of distributions.

4.2. Linear Sorter Depth. One of the key attributes of a linear sorter is its regular structure. Because of this regularity, a linear sorter of depth N can be implemented with little effect over the system's logic and routing delays. Therefore, regardless of the linear sorter depth, its implementation maintains timing performance. This is an asset, since ideally the linear sorter depth N needs to be large enough to prevent full conditions.

In simple cases, full linear sorters stall the input flow until one of its elements is removed. With more complex systems, there is a possibility that the logic controlling the sorted output streaming is input dependent (e.g., the system starts streaming after the tag number 1 is identified as the top sorted value). If the ILS needs to validate a condition before streaming its output, it will accumulate values until the condition is met. It is then possible that one of the queues becomes full and not able to service inputs or outputs, that is, a deadlock condition. Because only a single value can be inserted and deleted in each linear sorter, the depth N must be large enough to allow a sliding window of values. That is, at any given time, every input value received can be dependent only on the inputs received less than N clock cycles before. Fortunately, N can be easily changed to accommodate this restriction.

4.3. Output Logic. The ILS system needs to accumulate values before its sorted output becomes relevant. Therefore, there must be some logic in control of output streaming. A simple choice is to start the output when one of the linear sorters is detected as full or almost full, ensuring a continuous flow of sorted values. Another option would be to test the top sorted tag value before streaming. If we were to ensure the order for predefined tags, then the output logic must test each tag in a round-robin fashion before deletion from each linear sorter (an extension of the system presented in Figure 6). Because W tag values need to be inspected in a single clock cycle, this method's logic and routing generally limits the maximum frequency of the system.

An alternative approach for sorting the ILS's output is to use a pipelined sorting network. This sorting network can resort the top values from each of the W linear sorters. Since W is generally small, the necessary processing elements, latency, and area are also kept small, thus overcoming the hindrances of a sorting network implementation.

4.4. Simulation Results. We used ModelSim to simulate and functionally verify the ILS system. On each rising clock edge, the ILS system receives a new input set of tags, and situations involving contention for the interleaved linear sorters input ports are solved through input buffering. Figure 9 shows the simulation for signals of interest when such events occur. The initial parallel input of `Input Tags` consists of tags $\{0, 1, 6, 7\}$ which are interleaved without incident, sorted immediately, and sent to the `Sorted Output` port in the next rising clock at 50 ns. The following set of inputs $\{4, 5, 2, 3\}$ do not experience any contention for interleaved linear sorters either and are sorted in one clock cycle. At this point, the `Sorted Output` is $\{0, 1, 2, 3\}$, which are the top sorted tags able to be dispensed at a rate of four sorted tags per clock cycle.

Contention for linear sorter 0 occurs in the tag set $\{8, 9, 12, 11\}$, since zero is the interleaving value for tags 8 and 12. Tag 8 is serviced immediately because of the round-robin servicing scheme, and Tag 12 is saved to the `Buffered Input` register. The `Contention` flag now indicates the input logic to start buffering, which invalidates further input to the ILS system until it services the buffered tags. Tag 12, now residing in `Buffered Input`, is serviced in the next clock cycle at 70 ns. Finally, the input tags $\{10, 13, 14, 18\}$ will have three tags in contention for a linear sorter input, and therefore, two extra clock cycles will need to be used to buffer and dispatch them.

5. Delayed Contention Buffering

There is potential for improving the linear sorter contention scheme. The `Buffered Input` signal accumulates all contending values from the current tag set and dispenses them immediately. The average latency cost associated with this procedure was previously shown in Table 2. We instead accumulate contending tags over multiple sets of inputs, dispensing them when the `Buffered Input` signal is full. Table 3 shows the average performance increase when using this delayed buffered contention resolution scheme against the previously described immediate contention resolution scheme.

When the interleaved linear sorter width W is one, there is no benefit from the contention buffer as no conflicts arise from using the single linear sorter in the system. Additionally, when the contention buffer size CW is less than the ILS width W, additional logic must be implemented for worst case contention scenarios. In these cases, it is possible to have

Table 3: Average latency speedup with contention buffering for interleaved linear sorter of width W.

ILS width W	Contention buffer size CW					
	1	2	4	8	16	32
1	0%	0%	0%	0%	0%	0%
2	0%	9%	12%	14%	15%	17%
4	0%		23%	33%	41%	46%
8	0%			30%	45%	57%
16	0%				37%	57%

Table 4: Reduced average latency for interleaved linear sorter of width W.

Number of linear sorters W	Clock cycles	Speedup %
1	1.000	0%
2	1.344	12%
4	1.595	33%
8	1.752	45%
16	2.254	57%

$W - 1$ conflicting requests for a single linear sorter that need to be buffered. Since the buffer size CW cannot hold all these requests, a second buffer would be needed, subsequently making situations in which $CW < W$ quite unappealing due to the extra logic, buffer, and latency requirements.

Table 3 shows that for cases in which $CW \geq W$, there are speedups ranging from 9% to 57%. The greater speedups are achieved for cases in which both linear sorter contention is probable (large W), and the contention buffer can hold multiple requests accumulated over one or various clock cycles (large CW). This is convenient, as it subsidizes the elevated logic and routing costs associated with large interleaved linear sorter widths, thus not only maintaining their throughput but also reducing latency costs. Because the contention buffer prevents tags from being serviced immediately, the sliding window for contiguous values is increased. This requires a complementary increase in the linear sorter depth to augment storage capabilities for unsorted tags. In general, the extra storage capacity should exceed CW extra nodes per linear sorter, for a total of $W \times CW$ sorter nodes in the ILS system. Because additional linear sorter nodes do not greatly affect performance, there is only an area overhead cost associated with this reduced latency benefit.

We chose the cases in which the contention buffer input size CW was twice the interleaved linear sorter system's width W. This buffer size is practical, because it avoids overflows when multiple values get sent to a contention buffer that is almost full. Table 4 presents the revised latency of the ILS system with respect to its width. The additional latency for large values of W is not as dramatic as was presented previously on Table 2 and, therefore, counters the reduced clock frequencies experienced in implemented systems, as Section 6 will describe in detail.

Table 5: ILS FPGA area.

Linear sorters, W	Total slices	Slices/node	Area overhead
1	278	17.4	2.3%
2	641	20.0	17.6%
4	1294	20.2	18.8%
8	2612	20.4	20.0%
16	5250	20.5	20.6%

6. Hardware Implementation

The hardware implementation for the interleaved linear sorter depends on the width W of the ILS, the tag width T, the data width D, and the depth N for each of the W linear sorters. Of those, only three affect the maximum clock frequency. The width W tends to dominate the frequency, the depth N only has a minor effect, and larger values of T create longer tag comparisons. Tag sizes of 32-bits decreased the frequency by 16% compared to an 8-bit tag system. The data and tag sizes directly influence the total area. The number of the linear sorters, sorters depth, data, and tag size are parameterized offline before synthesis.

We used Xilinx ISE 11.1 and EDK 8.2 to synthesize the ILS for a Virtex-5 FPGA. The area for a linear sorter node was fairly static. For an 8-bit data and 8-bit tag linear sorter node, 17 FPGA slices were needed. The total area was generally scaled by the total number of nodes $W \times N$. Table 5 shows ILS systems of different widths W and their respective areas, with each linear sorter being 16 nodes deep.

The area overhead comes from the extra implementation logic for interleaving among multiple linear sorters. Likewise, it also includes the adder/subtractor and counter necessary for detecting full conditions on the linear sorters. Storage for the data to be sorted utilizes the FPGA board's BlockRAMs.

6.1. Throughput. The maximum throughput was calculated as the product of the maximum frequency and the number of linear sorters in the ILS. This assessment includes the logic necessary to interleave the inputs to their corresponding linear sorter, which for large values of W became a performance bottleneck. Table 6 shows the implemented Virtex-5 frequencies for $W \in \{1, 2, 4, 8, 16\}$, with 8-bit tags and 8-bit data, averaged for linear sorters with depth $N \in \{1, 2, 4, 8, 16, 32, 64, 96, 126, 256\}$. Even though at $W = 16$ the ILS throughput per clock cycle of the ILS is high, the clock frequency drops to 40 MHz due to the large logic and routing delays at the input logic stage.

The frequency discrepancies for ILS systems of width $W = 8$ and $W = 16$ are the result of the underlying logic cell in the Virtex-5 board. The lookup table (LUT), is a 6-input component that implements function generators. In the first three cases in Table 6, only one LUT is needed to generate functions with one, two, or four inputs. When the ILS has eight different inputs, two 6-input LUTs have to be cascaded to generate functions. With sixteen inputs, four LUTs are needed. This cascading of LUTs doubles and quadruples the

Table 6: ILS frequency and throughput.

Linear sorters W	Frequency (MHz)	Maximum sorting throughput (millions/s)
1	299	299
2	275	550
4	275	1101
8	132	1058
16	40	645

processing delay of inputs, and decreases the clock frequency accordingly.

Using these frequencies, we can now show the maximum throughput for different ILS configurations. Figure 10 shows the maximum sorting rate for $W \in \{1, 2, 4, 8\}$ for varying linear sorter depths N with 8-bit datas and tags. The 16-wide ILS was omitted due to the poor performance that resulted from its low clock frequency. The frequencies diminish as a function on ILS depth because of fanout. The input signals in each linear sorter have to be forwarded to all sorter nodes, which consequently increases the routing delay of these signals. In turn, the ILS clock frequency is decreased, but the effect is minimal in comparison to the routing delay experienced from interleaving tags to their corresponding linear sorters.

For ILS systems with two, four, and eight linear sorters, the average speedups against a single linear sorter were 1.8, 3.7, and 3.5, respectively. The diminishing returns trend is evident for $W = 8$, which shows a maximum throughput comparable to an ILS with $W = 4$. All of the ILS implementations showed higher throughput when compared to a single linear sorter system.

The maximum throughput results in Figure 10 do not consider two important factors. First, interleaving contention for the same linear sorter results in an average latency that increases with W, as previously shown in Tables 2 and 4. For the average case, the maximum frequency needs to be normalized by one of these factors. Second, the input logic for a small ILS width W will result in fairly simple logic and minimal delay. It is unlikely that this ILS delay would limit a nontrivial system. Instead, it is most likely that logic delays elsewhere in the same system determine the critical path and consequently sets the maximum frequency. As such, we assumed a maximum frequency of 300 MHz, which was the highest frequency obtained for 16-bit comparisons. This eliminates some of the artificially high frequencies an isolated ILS system achieves.

We first evaluate the decreased performance experienced when normalizing our maximum throughput due to interleaving contention for the system's linear sorters. Figure 11 shows the average throughput of an 8-bit tag and 8-bit data interleaved linear sorter, normalized by the corresponding ILS latency in Table 2 using immediate contention resolution with a 300 MHz maximum frequency. The interleaved linear sorters with $W = 2$ show a speedup of 1.3 while the one with $W = 4$ a 1.8 speedup. The $W = 8$ ILS, unfortunately, falls short at a 1.4 speedup due to its lower clock frequencies, contention, and complex input logic.

By using delayed contention resolution with a buffer twice the size of the interleaving width, we subsidize timing penalties of conflicting requests by dispatching them at once. Table 4 specified the ILS latency for our different interleaving values. Implementation of ILS systems with width $W > 4$ was not practical due to extreme routing delays. Contention buffering requires each of the W tag inputs to be routed to any of the $2 \times W$ contention buffer cells. The routing is not only expensive, but it is also magnified for $W > 4$ due to the FPGA's logic cell features, which limit LUT inputs to 6 and require cascading for larger values. The average frequency reduction for an ILS system with width $W = 8$ was more than 45% (the average speedup using contention buffering), making this enhancement perform worse than the original solution.

Figure 12 shows improved results for ILS systems with delayed contention buffering. The difference is more pronounced for $W = 4$, when the system can reap all the benefits of buffering without paying the expensive routing delay price. Its average sorting rate was 633 million values sorted per second, compared to 518 million when using immediate contention resolution.

6.2. Virtex Implementation. The implementation of a 4-way interleaving ILS was compared against quicksort running in a MicroBlaze processor, both in a Virtex-2 Pro ML310 device. The clock frequencies for software and hardware implementations matched the bus frequency of 100 MHz.

Data to be sorted resided in BlockRAMs. To create the tags, a pseudorandom scheme was used. The size of our sliding window for tag generation was set at 64, meaning that two contiguous tags would not be more than 16 address spaces apart within the four BlockRAMs. Unsorted sets of 64 tags and data were written in random order to the BlockRAMs while ensuring the sliding window property. The same data was used for both the MicroBlaze and the interleaved linear sorter tests.

The MicroBlaze version ran a C program for quicksort. The unsorted data resided as an array of values in a single BlockRAM, and values were retrieved and written through the on-chip peripheral bus (OPB) BRAM interface controller. For a small dataset of 64 values, MicroBlaze took 49,982 clock cycles, which include bus arbitration and read and write requests over the OPB. The end result is a sorted set saved in BlockRAM.

Three scenarios were setup with the ILS system doing hardware-based sorting. In each scenario, the unsorted data was saved in four BlockRAMs, with each BlockRAM output connected to the corresponding interleaved linear sorter input. The ILS output logic was set to delete tags and values in strictly contiguous increments, acting as a sorter rather than a priority queue. This means the output will halt until the next tag in the sequence is sorted to the top of the queue, as was shown in Figure 6, ensuring that we will get the same end results as the MicroBlaze test. Setting the depth of the four linear sorters to 16 nodes prevents each individual linear

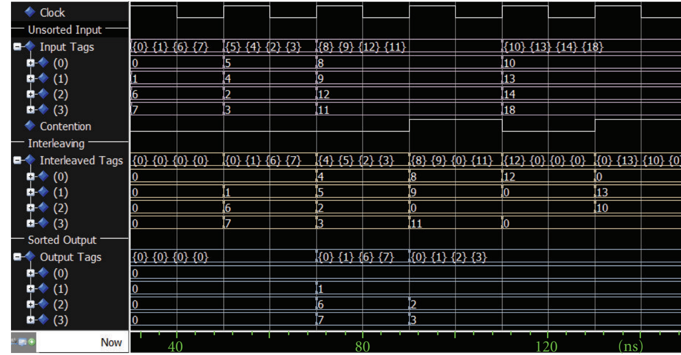


Figure 9: Simulation for interleaved linear sorter, including contention.

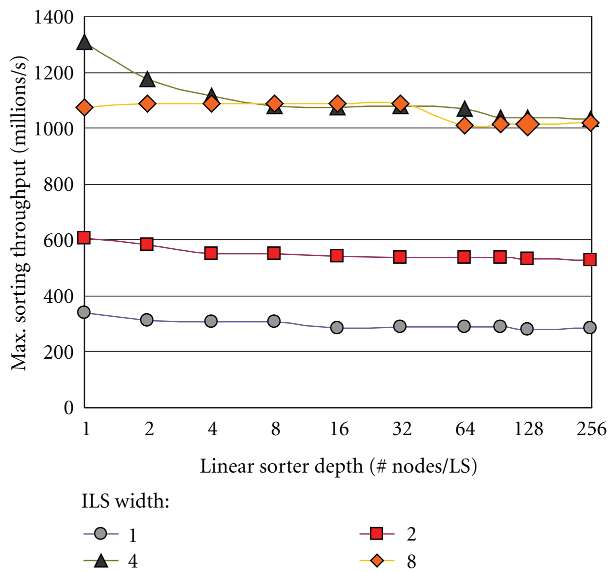


Figure 10: Maximum throughput for interleaved linear sorters.

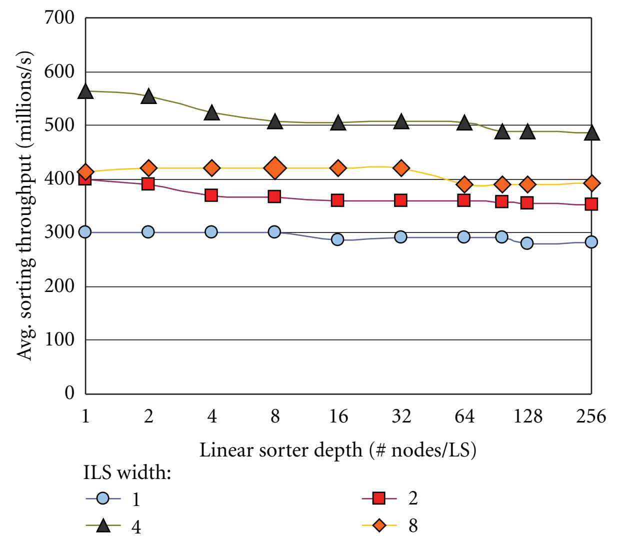


Figure 11: Normalized throughput for interleaved linear sorter system.

sorter from filling up while waiting for the input arrival of contiguous tags, even in the worst case scenario. This depth property further ensures that the input streaming remains not stalled due to lack of space in any of the linear sorters, preventing the system from going into a deadlock situation and allowing continuous output streaming.

In the first scenario, a MicroBlaze processor writes the unsorted data to the four OPB-based BlockRAMs and then sets a signal that starts the input streaming into the ILS. A simple hardware counter was used to drive the addresses of the BRAMs to cycle through all the unsorted values. The ILS output was then connected to the input ports of four secondary OPB BRAMs that hold the sorted values. Finally, the MicroBlaze reads back the sorted values through an on-chip peripheral bus (OPB) BRAM interface controller. Sorting with an ILS takes 2272 clock cycles, achieving a speedup of 22 over the MicroBlaze-only option.

The second scenario is set up in the same fashion as the first, but the results do not need to be read back into the MicroBlaze over the arbitrated bus since final storage happens in the ILS itself, whose linear sorters are deep enough to hold all the data. However, the MicroBlaze checks the ILS output to ensure all values have been retrieved before streaming the sorted results into the four OPB BlockRAMs, which takes a total of 732 clock cycles. Again, the end result is the sorted set saved in BlockRAM, but the speedup is magnified to 68 from the initial setup.

The third and final scenario involves a hardware-only approach with no MicroBlaze involvement. The output from the ILS is consumed by other hardware components as soon as it is ready (but still keeping the contiguous sorted output limitation). Under these circumstances, the interleaved linear sorting system takes only 30 clock cycles, a speedup of 1666 over the MicroBlaze quicksort.

6.3. Superscalar Processor Implementation. The greatest benefit of our interleaved linear sorter system is achieved on a hardware-only computational platform. Even though it is difficult to find applications that can fit this sorting scheme, the speedups attained in these systems make our approach

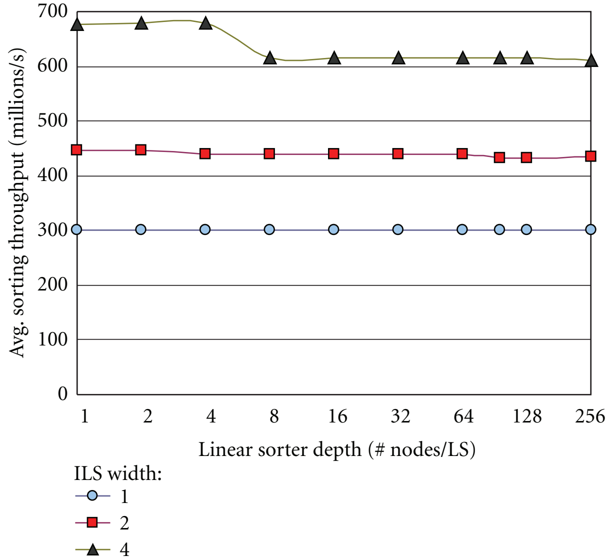


Figure 12: Improved normalized throughput for ILS system with delayed contention buffering.

an attractive option. Therefore an ideal target application is one that receives an unsorted streaming input of values, and transforms them into a streaming output of sorted data for other hardware components. One such application is a reconfigurable superscalar processor.

The configurable interleaved linear sorter was implemented as a component in an FPGA-based configurable superscalar processor [21, 22], which required resorting of processed results to allow precise interrupts. The superscalar processor utilizes register renaming and out-of-order execution to increase performance by exploiting instruction-level parallelism. One of the key components is its heterogeneous collection of functional units, which provide specialized processing for different instruction types at reduced execution latencies. Within the realm of reconfigurable computing, these functional units can also provide custom execution of instructions. The chronological sequence of events for an instruction execution starts with instruction retrieval, decoding, elimination of false data dependencies through register renaming, and storage in the reservation stations. These waiting instructions contain only true data dependencies, and once their operands are ready from the execution of previous instructions, they can be immediately dispatched in out-of-order fashion. Performance is thus increased when executing multiple instructions simultaneously in the parallel functional units.

To service interrupts from external sources like I/O and network, the processor saves its state, services the interrupt, and then restores state to continue execution. The superscalar's in-flight instructions, variable number of functional units, and out-of-order execution make restoration particularly difficult, so a reorder buffer is utilized to track in-order completion of instructions. The reorder buffer was an ideal target application for an interleaved linear sorter, as it had streaming input of instructions, streaming output of executed results, and a sliding window that set a maximum latency between contiguous tags depending on the number of in-flight instructions. Additionally, the processor's reconfigurable and superscalar nature also demanded increased throughput and performance while requiring flexibility for different number of instruction streams. The interleaved linear sorter system met all these throughput, streaming, and adaptability requirements.

Table 7: Superscalar and reorder buffer comparison.

	Min. delay	Max. frequency
Interleaving 1		
Superscalar processor	15.3 μs	65 MHz
Interleaved linear sorter system	3.612 μs	277 MHz
Interleaving 2		
Superscalar processor	16.5 μs	61 MHz
Interleaved linear sorter system	4.746 μs	211 MHz
Interleaving 4		
Superscalar processor	17.8 μs	56 MHz
Interleaved linear sorter system	6.941 μs	144 MHz

When used as the reorder buffer for the reconfigurable superscalar processor, the interleaved linear sorter system surpassed the throughput requirements of the system. The amount of interleaving was determined by a reconfigurable parameter of the processor, which controlled the memory banks available for parallel instruction retrieval. Table 7 shows the implementation delay and performance for a superscalar processor with an instruction issue width of four and four parallel functional units.

In all the interleaving cases in Table 7, the ILS system experiences less delay than that of the register renaming process in the superscalar processor. On average, the ILS delay was 34% of the register renaming delay, with 26% of that delay due to logic and 74% due to routing.

6.4. Comparison to Other Sorters. To compare our performance with that of a state-of-the-art Batcher odd-even sorting network implementation, we used the results from [20], which were also implemented in a Xilinx Virtex 2 device. The Batcher odd-even method took 95 ns to sort thirty-two 16-bit numbers. We set the ILS system to also sort thirty-two 16-bit tag values, taking 123 ns. While the mergesort performed better for a static data set, it still suffers from the detriments of sorting networks, namely the need to resort the full set of data upon a single new insertion and a large area implementation.

7. Conclusions

Linear sorters overcome the latency disadvantages of sorting networks and systolic sorters. Their regular structure makes them highly configurable and a fitting solution for streaming data. Nevertheless, their single-output nature limits their

throughput. We have presented an implementation using interleaving of linear sorters that alleviates this limitation, using a delayed contention buffering scheme to maintain a low sorting latency. An ILS system of width 4 showed, on average, a 1.8 speedup over a regular linear sorter and a speedup of 68 against an embedded MicroBlaze processor. In an all hardware-implementation without the need of bus requests, like our superscalar processor implementation, this speedup became 1666. The versatility of the ILS system also allows designers to easily configure the number of sorting nodes per linear sorter and the number of linear sorters in the system to best match system bandwidth and area requirements. This configurability makes ILS systems easily adaptable to a variety of applications, particularly those that require high throughput for sorting streaming data.

References

[1] K. Batcher, "Sorting networks and their applications," in *Proceedings of the AFIPS Spring Joint Computer Conference*, vol. 32, pp. 307–314, ACM, 1968.

[2] E. W. Dijkstra, "A heuristic explanation of Batcher's Baffler," *Science of Computer Programming*, vol. 9, no. 3, pp. 213–220, 1987.

[3] J. Ortiz and D. Andrews, "A configurable high-throughput linear sorter system," in *Proceedings of the 7th IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, Atlanta, Ga, USA, 2010.

[4] D. E. Knuth, *The Art of Computer Programming 3. Sorting and Searching*, Addison-Wesley Longman, Amsterdam, The Netherlands, 2nd edition, 1998.

[5] M. de Prycker, *Asynchronous Transfer Mode: Solution for Broadband ISDN*, Ellis Horwood, Upper Saddle River, NJ, USA, 1991.

[6] R. O. Onvural, *Asynchronous Transfer Mode Networks: Performance Issues*, Artech House, Norwood, Ma, USA, 2nd edition, 1995.

[7] R. Kannan, "A pipelined single-bit controlled sorting network with O(Nlog2 N) bit complexity," in *Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '97)*, vol. 1, pp. 253–260, April 1997.

[8] J. Martínez, R. Cumplido, and C. Feregrino, "An FPGA-based parallel sorting architecture for the Burrows wheeler transform," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '05)*, pp. 7–13, September 2005.

[9] N. Tabrizi and N. Bagherzadeh, "An ASIC design of a novel pipelined and parallel sorting accelerator for a multiprocessor-on-a-chip," in *Proceedings of the 6th International Conference on ASIC (ASICON '05)*, vol. 1, pp. 46–49, October 2005.

[10] D. Castells-Rufas, M. Monton, L. Ribas, and J. Carrabina, "High performance parallel linear sorter core design," GSPx, The International Embedded Solutions Event, September 2004.

[11] C. Leiserson, "Systolic priority queues," in *Proceedings of the Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pp. 199–214, 1979.

[12] B. Parhami and D. M. Kwai, "Data-driven control scheme for linear arrays: application to a stable insertion sorter," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 1, pp. 23–28, 1999.

[13] Y. Zhang and S. Q. Zheng, "Design and analysis of a systolic sorting architecture," in *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pp. 652–659, October 1995.

[14] M. Bednara, O. Beyer, J. Teich, and R. Wanka, "Hardware-supported sorting: design and Tradeoff analysis," in *System Design Automation: Fundamentals, Principles, Methods, Examples*, p. 97, 1979.

[15] C.-S. Lin and B.-D. Liu, "Design of a pipelined and expandable sorting architecture with simple control scheme," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '02)*, vol. 4, pp. 217–220, 2002.

[16] R. Perez-Andrade, R. Cumplido, F. M. Del Campo, and C. Feregrino-Uribe, "A versatile linear insertion sorter based on a FIFO scheme," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI: Trends in VLSI Technology and Design (ISVLSI '08)*, pp. 357–362, April 2008.

[17] C. Y. Lee and J. M. Tsai, "A shift register architecture for high-speed data sorting," *Journal of VLSI Signal Processing*, vol. 11, no. 3, pp. 273–280, 1995.

[18] A. A. Colavita, A. Cicuttin, F. Fratnik, and G. Capello, "SORTCHIP: a VLSI implementation of a hardware algorithm for continuous data sorting," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 6, pp. 1076–1079, 2003.

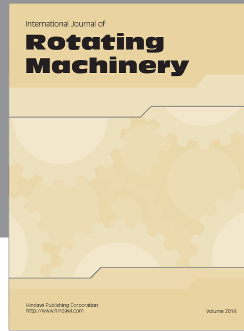
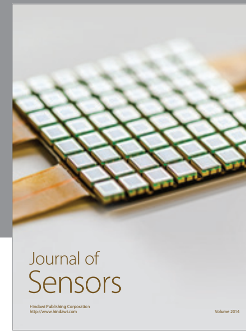
[19] R. Marcelino, H. Neto, and J. Cardoso, "Sorting units for FPGA-based embedded systems," in *Proceedings of the IFIP 20th World Computer Congress, TC10 Working Conference on Distributed and Parallel Embedded Systems (DIPES '08)*, vol. 271, pp. 11–22, Springer, Milano, Italy, September 2008.

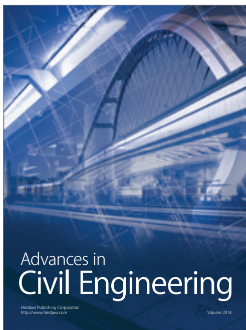
[20] L. Ribas, D. Castells, and J. Carrabina, "A linear sorter core based on a programmable register file," in *Proceedings of the 19th Conference on Design of Circuits and Integrated Systems (DCIS '04)*, pp. 635–640, 2004.

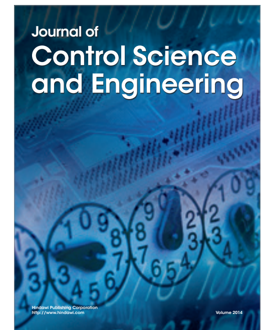
[21] J. Ortiz, "A reconfigurable superscalar processor architecture for FPGA-based designs," in *Proceedings of the International Conference on Computer Design (CDES '09)*, pp. 211–217, July 2009.

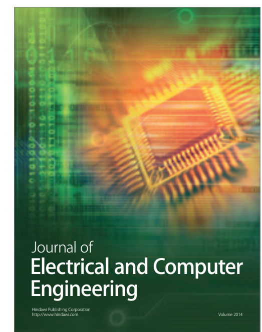
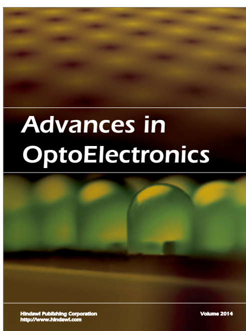
[22] J. Ortiz, *Synthesis techniques for semi-custom dynamically reconfigurable superscalar processors*, Ph.D. dissertation, University of Kansas, Department of Electrical Engineering and Computer Science, 2009.

Journal of
Engineering

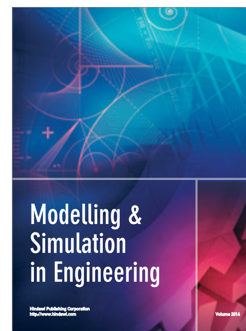
The Scientific
World Journal

International Journal of
Rotating
Machinery

Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

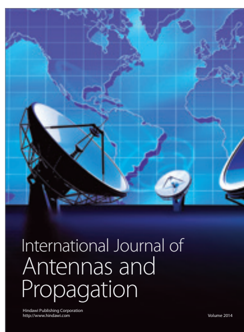
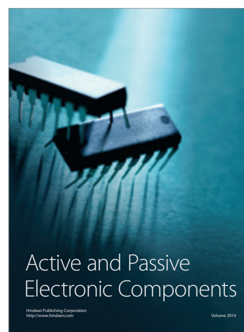
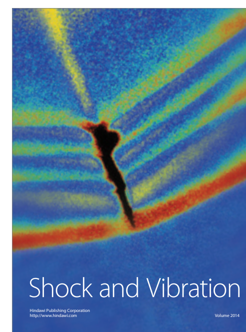
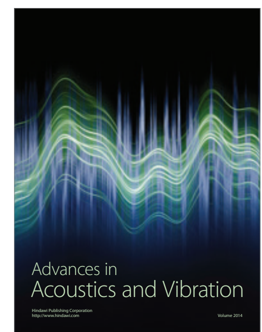
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration