Processing Tree-like Data Structures for Sorting and Managing Priorities

Valery Sklyarov  
DETI/IEETA,  
University of Aveiro,  
Aveiro, Portugal  
skl@ua.pt

Iouliia Skliarova  
DETI/IEETA,  
University of Aveiro,  
Aveiro, Portugal  
iouliia@ua.pt

Dmitri Mihhailov  
Computer Dept.,  
TUT,  
Tallinn, Estonia  
d.mihhailov@ttu.ee

Alexander Sudnitson  
Computer Dept.,  
TUT,  
Tallinn, Estonia  
alsu@cc.ttu.ee

Abstract- The paper describes the hardware implementation and optimization of algorithms that sort data using tree-like structures combined with sorting networks. The emphasis is done on applications that require dynamic resorting for new incoming data items. Experiments and comparisons demonstrate that the performance is increased compared to other known implementations.

I. INTRODUCTION

Using and taking advantage of application-specific circuits in general and FPGA-based accelerators in particular have a long tradition in data processing [1]. A number of research works are targeted to databases that use the potential of advanced hardware architectures. For example, the system [2] solves a sorting problem over multiple hardware shading units achieving parallelization through using SIMD operations on GPU processors. The algorithms [3,4,5] are very similar in nature, but target SIMD instruction sets of PowerPC 970MP, Cell, and Intel Core 2 Quad processors.

The application of FPGAs was studied within projects [6,7] implementing traditional CPU tasks on programmable hardware. In [8] FPGAs are used as co-processors in Altix supercomputer to accelerate XML filtering. The advantages of customized hardware as a database co-processor are investigated in different publications (e.g. [1]).

Data processing has many practical applications [9,10] and one of them is in the scope of distributed embedded systems and networks, targeted to the design of priority buffers (queues). Such buffer is a device that stores an incoming (sequential) flow of instructions (or other data) and allows outputs to be selectively extracted from the buffer for processing. The following list exemplifies some typical selection rules:

- Each instruction (data item) is provided with an extra field indicating its priority. The selection mechanism has to be able to extract the instruction with the highest priority;
- The system has to be able to remove from the buffer all the instructions that are not longer required;
- The system has to be able to check if a particular instruction is in the buffer.

The technique proposed in this paper can be used to design hardware circuits for sorting and managing priorities with fast resorting for new incoming data items. Let us consider two examples. Suppose, it is necessary to sort arriving data items during run-time. The data items are received by portions and we need to resort them as soon as a new portion has arrived. The known methods, such as [1,10,11] cannot be used efficiently because all existing data have to be resorted from the beginning after any new portion has been received and the resorting takes relatively long time. In the second example, we would like to design a priority buffer with characteristics described above. Many known methods targeted to the design of such buffers are based on sort and shift algorithms [12]. Once again, the existing hardware accelerators for sorting are not efficient, because they require frequent resorting, starting from the beginning that is not appropriate for time critical systems.

This paper suggests a technique for sorting and managing priorities that takes into account potential frequent changes in input data. The technique is based on the use of tree-like data structures and it has the following distinctive features: a) fast correction of output sorted sequence after any change in the input; b) optimization of sorting algorithms through parallelization and balancing the initial tree; c) acceleration through combining tree-like data structures with sorting networks.

The remainder of this paper is organized in four sections. Section II suggests methods for processing tree-like data structures. Section III is dedicated to the relevant hardware architectures. Section IV describes implementations and experiments. The conclusion is given in Section V.

II. PROCESSING OF TREE-LIKE DATA STRUCTURES

Tree-like data structures are frequently explored for data processing [10,12,13]. Let us consider an example of using a binary tree for sorting data [13]. Suppose that the nodes of the tree contain four fields: a pointer to the left child node (an address of the left child node) LA, a pointer to the right child node (an address of the right child node) RA, a counter count and a value val (e.g. an integer or a pointer to a string). Such fields can be described by the following C++ structure template tree_node:

```
template <class T> struct tree_node {
    T val;                // Value of an item of type T
    int count;            // Number of items with value val
```
The memory is full the signal speed in S. Thus, buffering in internal memory is essential. If general the input transfer rate is not the same as the processing be resorted as fast as possible. When any item is sent to the any new data item has arrived, all available data in S have to be sorted. This allows various sequences of operations over a tree to be executed in parallel will also be explored.

A. General Functionality

Let us consider a device (called a sorter - S) that receives input data from outside and generates a sorted sequence that is sent to other external devices (see Fig. 1a).

The number of receiving data items is generally unknown. A processing unit (see Fig. 1b) is responsible for four basic operations: a) receiving data while Enable signal is active; b) storing the received data in internal memory; c) sorting the data currently available in the memory; and d) outputting the sorted data on external requests. Thus, sorting has to be provided beginning with the first incoming data item. When any new data item has arrived, all available data in S have to be resorted as fast as possible. When any item is sent to the output it has to be removed from memory in Fig. 1b. In general the input transfer rate is not the same as the processing speed in S. Thus, buffering in internal memory is essential. If the memory is full the signal Enable becomes not asserted.

Such kind of processing differs from traditional data sorting when initial data are kept in memory and have to be sorted. In many cases the sorted sequence of data can be saved in the same memory. Thus, the number and values of data are fixed and cannot be changed after the relevant processing has started. We can call such sorting static as distinct from the considered sorting that is dynamic. In other words, we have to be able:

- To accommodate any incoming item in a proper position of the sorted sequence;
- To remove from the sorted sequence an item that is sent from sorter' output to external device;
- At any time the memory of size R has to accommodate up to R data items. The total number P of inputs can be significantly bigger than R (because items are periodically removed from the head of the sorted sequence).

Thus, the number and values of data are not fixed and they are changed during processing time. As a result the majority of known methods developed for data sort either cannot be used or are very time consuming.

We suggest to process data through their addresses in internal memory. Potential techniques can involve linked lists and binary trees. We base our approach on trees because they permit to execute the required operations faster through applying both sequential and parallel processing.

B. Sequential Sort

Two (following each other) sequential algorithms SA1 and SA2 have to be executed for data sort in [13]. The first one SA1 constructs the tree. The second algorithm SA2 outputs the sorted data from the tree. SA1 executes the following steps: 1) compare the new data item with the value of the root node to determine whether the item should be placed in the sub-tree headed by the left node or the right node; 2) check for the presence of the node selected by 1) and if it is absent, create and insert a new node for the data item and end the process; 3) otherwise, repeat 1) and 2) with the selected node as the root.

The following two C++ functions give additional details for sorting integers (i.e. in the structure above parameter $T$ is int):

```cpp
// SA1: Constructing the tree with the structure considered above
tree_node<T>* build_tree(tree_node* node, int v)
{
    if (node == 0)
    {   node = new tree_node;
        node->val = v;
        node->count = 1; // setting counter to 1
        node->lnode = node->rnode = 0;
    }
    else if (v == node->val)
        node->count++; // incrementing counter
    else if (v < node->val)
        node->lnode = build_tree(node->lnode, v);
        // traversing the left sub-tree
    else node->rnode = build_tree(node->rnode, v);
        // traversing the right sub-tree
    return node;
}
```

```cpp
// SA2: traversing the tree void tree_sort(tree_node* node)
{   if (node!=0)
    {   // if the node exists
        tree_sort(node->lnode); // sort left sub-tree
        tree_sort(node->rnode); // sort right sub-tree
        // display value after any hierarchical return
    }
}
```

Fig. 2 demonstrates an example assuming that the following
sequence of input data has been received before the request for sorting: 10, 7, 9, 12, 3, 15 (see Fig. 2a). Since there is no repeated data, \( count = 1 \) for all nodes.

Let us consider ascending sort: 3, 7, 9, 10, 12, 15 and suppose the sorted sequence has to be sent to output from left to right. Each word of memory has three fields: the value \( v \), an address of the left sub-tree LA, and an address of the right sub-tree RA. For example, Fig. 2c demonstrates the representation in memory of the tree shown in Fig. 2a. Suppose the first two data items (i.e. 3 and 7) have been sent to the output of the sorter and two new data items (e.g. 8 and 11) have arrived. As a result, two nodes of the tree (3 and 7) have to be removed (see Fig. 2d) and two new nodes with the value 8 and 11 have to be inserted (see Fig. 2b and 2e). Finally, P=8 arrived data can be temporarily saved in the memory containing R=6 words.

The build_subtree function is a simplified build_tree function with the following code:

```cpp
tree_node* build_subtree(tree_node* node, tree_node* subnode, int v)
{
    if (node == 0)
        node = subnode;
    else if (v < node->val)
        node->rnode = build_subtree(node->rnode, subnode, v);
    else
        node->lnode = build_subtree(node->lnode, subnode, v);
    return node;
}
```

Fig. 2. Pointer based dynamic data sort using binary trees.

The build_subtree function is a simplified build_tree function with the following code:

```cpp
tree_node* build_subtree(tree_node* node, tree_node* subnode, int v)
{
    if (node == 0)
        node = subnode;
    else if (v < node->val)
        node->lnode = build_subtree(node->lnode, subnode, v);
    else
        node->rnode = build_subtree(node->rnode, subnode, v);
    return node;
}
```

The methods [15] enable us to convert the considered above C++ functions to hierarchical graph-schemes (HGSs), which can be seen as constrained flow charts [16]. The set of HGSs is a synthesizable specification making it possible to design the sorter hardware based on the model of hierarchical finite state machine (HFSM) [16]. At the beginning the HGSs are optimized in such a way that permits to take into account particularities of the target hardware platform such as rational use of dual-port memories applying methods [17].

Fig. 3 demonstrates a potential improvement for the function tree_sort. An address of the current sub-tree root is kept in a register. At the beginning left sub-trees of the current root are examined. If a left sub-tree (node) exists then it is checked again, i.e. to determine whether the left sub-tree also has either left or right sub-trees (nodes). Two addresses can be verified in parallel in dual-port memories [17]. If there is no sub-tree from the left node, then the value of the left node is
the leftmost data value and can be output as the smallest. In the last case the node addressed by the register holds the second smallest value and the relevant data value is sent to the output. Then similar operations are executed for right nodes.

Analogous optimization technique can also be applied to the other C++ functions when they are converted to HGSs.

The considered tree-like data structures possess the following important feature: it is not necessary to rebuild the existing tree when a new portion of input data items arrives. Thus, resorting for newly arriving items can be done very fast.

**C. Parallel Sort**

In the considered parallel algorithm (let us call it PA), N trees (N>1) are built and then processed in parallel [18]. Let us consider an example for N=2. In this case the 1st, the 3rd (N+1), the 5th (2N+1), etc. incoming data items are included into the 1st tree. Consequently, the 2nd, the 4th (N+1)+1, the 6th (2N+1)+1, etc. incoming data items are included into the 2nd tree. Fig. 4 shows N=2 trees that are built for our example (see the sequence of input data items given above and Fig. 2a).

**D. Balancing**

The tree that is built by the algorithm SA1 can be balanced differently. For example, Fig. 5a shows a completely unbalanced tree that has just one linear branch. In the worst case, in order to add a new node we have to perform d steps where d is the depth of the tree. To reduce d the tree can be balanced at run-time. Fig. 5b demonstrates the result of balancing for the tree in Fig. 5a allowing d to be decreased from 14 to 3. Balancing allows inserting new nodes faster.

**E. Using Sorting Networks**

Some of the most efficient traditional approaches to sorting are also appropriate for hardware circuits [1]. One of such approaches is based on sorting networks [3,10] not requiring control flow of instructions or branches. Besides, sorting networks are parallel in nature. However, they are only suitable for relatively short sequences of data whose length is known a priori [1].

We suggest combining tree-like structures with sorting networks for data processing. The basic idea is the following. Suppose we need to sort m-bit data items. Let us apply the considered above algorithms for (m-k) most significant bits and sort the remaining k bits using sorting networks. In this case, the algorithms will sort up to 2*m groups of data items and up to 2*k data items within each group will be sorted by the networks. Such technique is additionally explained on an example shown in Fig. 6 for m=5, k=2 and the following sequence of inputs (that is the same as in Fig. 2a): 10(01010), 7(00111), 9(01001), 12(01100), 3(00011), 15(01111), where the binary code of the relevant decimal value is shown in parenthesis.

For this example, m=3 most significant bits arrive in the following sequence: 010, 001, 011, 000. The relevant bits are shown in *italic* in Fig. 6 and in the text. The tree for this sequence is shown in Fig. 6a. There is one less significant (k=2)-bit value associated with the group 000 namely 11 (3). Thus, an additional sort is not required. However, there are two values associated with the group 010: 2 (10) and 1 (01). The values 2 and 1 are written near the node 010 in Fig. 6a. By analogy (k=2)-bit values associated with each group are shown.
near the other nodes. All associated values are ordered in ascending sequence using sorting networks (see Fig. 6a where a trivial network is chosen). Thus, the groups are sorted using tree-like structures and the values associated with the groups are sorted using networks. Comparators needed for the networks are represented in Fig. 6a through the known Knuth notation [9]. The results of sorting are shown in Fig. 6b.

III. HARDWARE

A. Hardware Implementation of Sequential Sort

Sorting is provided by a processing module (PM) interacting with memory that keeps incoming data items that are received and stored sequentially by incrementing the memory address for any new item. An example is shown in Fig. 7a. The absence of a node is indicated by 0 because zero address is used just for the root and can easily be recognized.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ....
24 17 35 30 8 61 12 18 1 25 10 15 19 21 40 9 7 11 16 50
```

Values, associated with each group, are coded using positional technique in memory that has 2^k 1-bit words. For example, the values 2,1 associated with the root in Fig. 6 are coded as: <address 0> 0; <address 1> 1; <address 2> 1; <address 3> 0. The memory has two ports with different sizes for addresses and data. The first port is used to store individual bits in accordance with the considered above example. The second port permits to read the recorded 1-bit values as a single binary vector V (V=0110 for our example). The V is assigned to k-bit values associated with a node to inputs of a predefined sorting network. It is done through the following expression:

```
for all i within the range 0,...2^k-1 do if V(i) = 0 then 2^k-1 else i
```

Thus, if V(i) is 0, the relevant input of the sorting network is assigned the maximum value, else the actual value. The number of values that have to be taken from the network is determined with the aid of a counter that indicates the number of associated values for each group.

D. Dynamic Memory Allocation and De-allocation

The considered technique permits dynamic memory allocation/de-allocation to be provided. Indeed, data are saved in memory (see Fig. 1b) through the addresses of the nodes (see Fig. 2c, 2d, 2e). An availability of any memory cell is indicated by a one-bit flag. When a new node is saved in the memory the relevant flag is set to 1. When a data item is sent to the outputs on a request the relevant flag is set to 0. Thus the memory is filled in and freed dynamically, which is somehow similar to a circular buffer. However, the buffer permits to allocate cells sequentially from one side and free cells sequentially from the other side. In the considered memory cells can be taken and freed on any arbitrary address and permission for such address is determined just by the value of the flag. As soon as the memory is completely filled, the processing module sets disable signal (i.e. Enable=0 in Fig. 1a), which indicates that the sorter cannot accept input items. Thus, the sorter can be seen as a device that takes input data items, provides their buffering, and outputs the sorted sequence.

IV. IMPLEMENTATION AND EXPERIMENTS

Firstly, we simulated the sequential and parallel sort in software (in C++) running on HP EliteBook 2730p (Intel Core 2 Duo CPU, 1.87 GHz) tablet PC. The performance was evaluated through counting the number of clock cycles.

Secondly, the synthesis and implementation of the circuits from the specification in VHDL were done in Xilinx ISE 11 for FPGA Spartan3E-1200E-FG320 of Xilinx available on NEXYS-2 prototyping board of Digilent Inc.

For both types of experiments a random-number generator produces up to 2^12 data items with a length of 14 bits that are supplied in portions to the actual and simulated circuits that have to sort the previously received portions of data and to sort them for a new portion as soon as possible. Besides, both types were evaluated without and with sorting networks for different values of k.

Fig. 8a, 8b permit to compare acceleration of sorting for a new portion (that includes from 10 to 120 data items) comparing with sorting all 2^12 data items in
simulated/physical circuits respectively. Fig. 8a and 8b clearly demonstrate that the results of simulation are very similar to the results for physical circuits.

![Image](image_url)

**Fig. 8.** The results of experiments

**TABLE I.** The results of experiments

<table>
<thead>
<tr>
<th>Number of data items</th>
<th>Sequential sort</th>
<th>Parallel sort (N=2)</th>
<th>Parallel sort (N=4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4019</td>
<td>16075</td>
<td>5622</td>
<td>4047</td>
</tr>
<tr>
<td>3985</td>
<td>15939</td>
<td>5553</td>
<td>4013</td>
</tr>
<tr>
<td>3945</td>
<td>15779</td>
<td>5541</td>
<td>3969</td>
</tr>
<tr>
<td>3969</td>
<td>15875</td>
<td>5531</td>
<td>4001</td>
</tr>
<tr>
<td>3979</td>
<td>15915</td>
<td>5587</td>
<td>4041</td>
</tr>
<tr>
<td>3963</td>
<td>15851</td>
<td>5540</td>
<td>3989</td>
</tr>
<tr>
<td>4010</td>
<td>16039</td>
<td>5622</td>
<td>4037</td>
</tr>
<tr>
<td>3977</td>
<td>15907</td>
<td>5553</td>
<td>4013</td>
</tr>
<tr>
<td>4048</td>
<td>16191</td>
<td>5706</td>
<td>4072</td>
</tr>
<tr>
<td>3988</td>
<td>15951</td>
<td>5555</td>
<td>4014</td>
</tr>
</tbody>
</table>

When we combined tree-like structures and sorting networks for sequential and parallel algorithms we were able to provide additional acceleration on an average 1.6/3.1/4.7 times for different values of k: \( k = 2 \). Regardless of parallelization of 2\( k \) operations, acceleration cannot be equal to 2\( k \) because the use of sorting networks gives significant delay in getting the results due to long paths in the relevant combinational circuits.

The considered technique is useful for different practical applications, such as priority buffers described in [19].

V. CONCLUSION

The paper suggests methods for the design and implementation in hardware of a device that receives input data items and executes sorting of these items dynamically in such a way that permits the accumulated data to be sorted properly when new items arrive. To satisfy the considered requirements data are stored in internal processing memory and relationships between data items are set through their addresses (pointers). The applicability and advantages of the proposed technique are demonstrated through simulation in software, prototyping in FPGA, and experiments.

**ACKNOWLEDGMENT**

This research was supported by the European Union through the European Regional Development Fund and the Estonian Doctoral School in Information and Communication Technology.

**REFERENCES**