2011 21st International Conference on Field Programmable Logic and Applications

Implementation in FPGA of Address-based Data Sorting

Valery Sklyarov, Iouliia Skliarova
DETI / IEETA
University of Aveiro
Aveiro, Portugal
e-mail: skl@ua.pt, iouliia@ua.pt

Dmitri Mihhailov, Alexander Sudnitson
Department of Computer Engineering
Tallinn University of Technology
Tallinn, Estonia
e-mail: d.mihhailov@ttu.ee, alsu@cc.ttu.ee

***Abstract*—The paper describes the hardware implementation and optimization of sorting algorithms that use data items as memory addresses with one-bit flags indicating presence of data. The proposed technique enables such type of address-based sorting to be applied either directly or through tree-walk tables permitting number of bits in sorted data items to be increased by constructing and traversing N-ary trees (N>2) composed of so called no-match and working nodes. The latter are organized in well balanced sub-trees of equal depth. It is allowed more than one data item to be assigned to leaves of working sub-trees and such sets of items are processed by fast acceleration circuits. Experiments and comparisons demonstrate that the proposed technique can be used efficiently in low cost FPGAs.**

Keywords-data sort; FPGA; N-ary tree; HFSM

I. INTRODUCTION

Using and taking advantage of FPGA-based accelerators have a long tradition in data processing [1] essential for vast variety of computational systems. Among numerous tasks that need to be solved, sorting is considered to be one of the most important [2]. Since it is time consuming for large volumes of data, acceleration is greatly required for many practical applications. A number of recent research works in this area are targeted to the potential of advanced hardware accelerators, which are analyzed in detail in [3]. Notable results have been achieved through applying parallelism, pipelining, non-sequential circuits and other techniques and building specialized blocks in hardware. A special attention has been paid to such competitive implementation platforms as: FPGAs, graphical processing units (GPU) and multi-core CPU. In [4] GPUs execute comparison-based parallel sort and the results are analyzed in competition with other known state-of-the-art implementations on GPUs: similar comparison-based sort [5], quicksort [6], bitonic sort [7] and some others. Up to 30% increase in performance is reported. In [8] FPGA, GPU, and multi-core CPU systems have been evaluated demonstrating distinctive features of different platforms and allowing an appropriate platform to be selected. The use of FPGAs, permitting design constraints of CPU and GPU with predefined architectures to be eliminated, is studied in a number of publications [*e.g.* 3, 8-13]. An interesting technique is described in [13], where a set of potential methods fairly unique to reconfigurable hardware are investigated.

FPGAs and GPUs are highly competitive. We believe that the winner can be determined by efficacy, pertinence and simplicity of the implementation technique. It is important to discover such methods that take advantage of the platform (due to its uniqueness) considering not only the number of the required operations but also efficiency of these operations in hardware circuits [13]. This paper presents a simple, but efficient sorting technique fairly unique to hardware and the method (called address-based sorting) that is particularly useful for FPGAs.

The remainder of this paper is organized in six sections. Section II describes traditional vs. stream based sorting and the role of FPGA accelerators. Section III introduces a core of address-based data sort. Section IV suggests N-ary incomplete trees (N≥2) with multiple items associated with leaves. Section V is dedicated to the details of implementation. Section VI presents experiments, comparisons, and discusses the results. The conclusion is given in Section VII.

II. FPGA-BASED ACCELERATORS AND THEIR PRACTICAL APPLICATIONS

We assume that input data are kept either in memory common to traditional computers (see Fig. 1a, where the given integers are shown in binary and decimal notations), or are represented in form of incoming streams (see Fig. 1b), dynamically generated from different sources, such as distributed sensors in networked embedded systems.

Suppose an FPGA-based hardware implements a sorter that takes input data either from memory (Fig. 1a) or from streams (Fig. 1b) and outputs a sorted sequence that can either replace the original (unsorted) data in memory, or be saved in a separate memory, or be transmitted to another device (see Fig. 1).

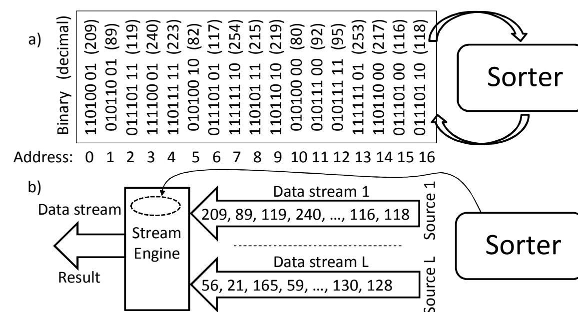

Figure 1. Input/output data and the role of sorter.

Physically sorters can be used differently, for example, they can be connected through a system bus of a general-purpose computer and access computer memory (that is a source of data) through allocated windows in memory space; or they can be seen as a standalone accelerator getting external packages of unsorted data and outputting sorted sequences; etc. In some practical applications data

978-0-7695-4529-5/11 $26.00 © 2011 IEEE
DOI 10.1109/FPL.2011.81

IEEE computer society

have to be resorted dynamically as soon as a new data package/item is received [14]. In all cases we have to make clear what is taken into account when we measure performance and resources. For example, if a sorter is operated as it is shown in Fig. 1a we eventually have to measure a time interval beginning from the first access to unsorted data in external memory until getting the final recorded sorted sequence. However, if input data are preliminary copied to FPGA memory and the result is also kept inside the FPGA then measurements are similar but since they are done inside the FPGA, the performance is undoubtedly better due to avoiding multiple input/output operations. In the proposed address-based data sort getting data can be combined with their processing. Thus, sorting and getting input data can be executed in parallel as shown in Fig. 2.

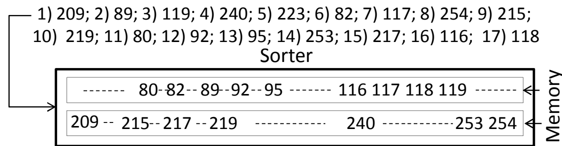

Figure 2. Combining input and positioning operations.

Input data in Fig. 2 are the same as in Fig. 1a and they arrive in a sequence indicated by numbers 1)…17). As soon as a new data item has arrived it is inserted into a proper memory address. Thus, if we look at memory we can see the sorted chain. Similar method is used when we need to output the sorted data (*i.e.* skipping unused spaces between data items and outputting can be done in parallel). If we are capable to process data within the same time interval that is needed for input and output, then the delay for exchanging data with memory becomes negligible and can be omitted when measuring performance.

III. A Core of Address-based Data Sort

The main idea is rather simple. As soon as a new data item is received, its value V is considered to be an address of memory to record a flag (1). We assume that memory is zero filled at the beginning. If floating point numbers need to be sorted then the considered above technique is used to sort integer parts of input data items and their fractional parts are associated with the relevant integers and sorted somehow differently applying any known method. Fig. 3 takes a simple example from [2] and shows how the sorting is done.

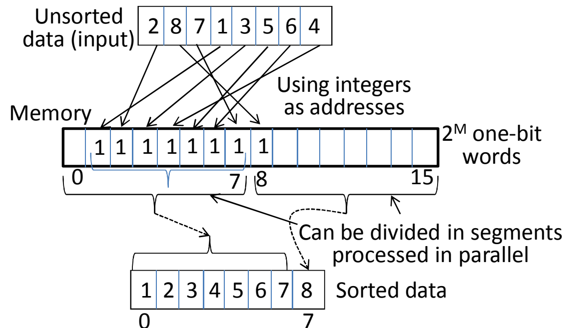

Figure 3. The core of address-based data sort.

As soon as all input data are recorded in memory in form of sequence 011111111 shown in Fig. 3, the sorted sequence can be immediately transmitted to any destination. The vector 011111111 has to be converted to integers and this can be done by a two-level combinational circuit, thus, a delay is minimal. Besides, the vector can be fragmentized in such a way that fragments (segments) are processed (converted to the respective numbers) in parallel. Fig. 4 gives an example of conversion for two segments (01111111 and 1) from Fig. 3. Each segment is composed of 8 consecutive bits of the vector 0,..,7 and 8,…,15 accordingly. Clearly, other sizes can be chosen. Eight bits (such as 01111111 from Fig. 3) are considered to be inputs of a two-level combinational circuit that makes the conversion and outputs 8 sorted integers with the number of valid outputs corresponding to the number of ones in the segment (*i.e.* 7 for our example). The sorted sequence from Fig 3, managed by the converter in Fig. 4, is transmitted either sequentially (1, 2, 3, …) or in parallel (*i.e.* all seven values are transmitted in one word within a single clock cycle). The method is obviously simple and effective but there are some problems discussed below.

First, the size of memory is large. Suppose, we need M-bit data to be sorted and M = 32...64. Thus, the number of one-bit words becomes $2^{32}...2^{64}$. Relying on the Moore law we can expect cheaper and larger memory to become available on the market but the required size ($2^{32}...2^{64}$) is still huge.

Second, if we sort M-bit data, many often, the number of input data items Q is significantly less than 2^M ($Q<<2^M$) especially for large values of M. Thus, we can expect a huge number of empty positions in memory space without data (some of such holes are easily visible in Fig. 2). This situation is somehow similar to the SAT problem where we want to consider a formula with Q clauses and M variables and $Q<<2^M$. Thus, we can apply some ideas inherited from the SAT such as the tree-walk tables proposed in [15].

Third, the circuit in Fig. 4 becomes very complicated for large segments. We can replace this circuit with a sorting network [3], but the latter requires lots of cascaded comparators having significant delays and huge hardware resources. We intend to use another way discussed below.

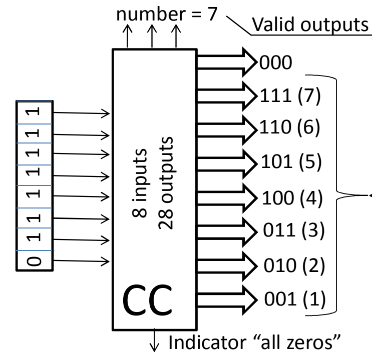

Figure 4. Using a combinational converter (CC).

IV. Tree-walk Tables and U-leaves Trees

Let us consider binary codes of input data (see Fig. 1a), use the converter for K less significant bits and sort the remaining M-K bits differently. The value K is limited by

the complexity of the circuit shown in Fig. 4 and in practice might be equal to 2...4. Thus, from 2^2=4 to 2^4=16 data items are sorted by the combinational circuit in Fig. 4. Note that the value of K can be significantly increased if address-based sort described in section III is applied for K less significant bits. We suggest to sort the remaining up to 2^{M-K} data items by an N-ary tree (N>2) described below.

A. N-ary Trees

Let as agree to consider such N-ary trees for which N is a power of 2, *i.e.* 2,4,8,16, etc. Suppose, G=$\log_2$N, and, thus, G is 1,2,3,4, etc. Let us now partition input data into N=2^G groups in such a way that assigning to the proper group or allocating a new group can be done immediately after receiving a data item. Since N=2^G, we can allocate a new group on the basis of G-bit segments within M-K bit words much like it is done in [15] for clause index walk.

Let us divide given (input) M-K bit data items into $\lceil$(M-K)/G$\rceil$ segments from left to right in such a way that the left segment contains less than G bits (if $\lceil$(M-K)/G$\rceil$>(M-K)/G)) and the remaining segments contain exactly G bits. For example, for G=K=2 any data item from Fig. 1a (where M=8) is composed of three 2-bit segments and they are 11 01 00 for the first data item 209. These three 2-bit segments will be associated with nodes of N-ary tree (4-ary tree for our example) and every non-leaf tree node has up to N=4 children (*i.e.* the tree will be $\lceil$(M-K)/G$\rceil$ deep: 3 for our example). The depth of the tree is equal to the number of edges and, thus, to the number of steps needed to execute forward traversal from the root to a leaf (one step for every edge). Clause index walk in [15] can be renamed now to *GLS index walk*, where GLS means a group of K less significant bits (for number 209 in Fig. 1a, K=2 and the value of GLS is 01). When we receive data, G segments are sequentially analyzed to allocate the existing group on the tree or to create a new one. If $\lceil$(M-K)/G$\rceil$>(M-K)/G then the number of sub-trees from the root is less than N=2^G. Suppose M=9, G=K=2. Thus, there are four segments (one 1-bit segment and three 2-bit segments) and there are just two sub-trees for the root.

Let us consider an example with integers shown in Fig. 1a. Six most significant bits (M-K = 6) divided into 3 segments will be used to build the 4-ary tree (N=4). The following steps have to be executed for each data item:

1) Take the most significant segment and allocate a new child node or use the existing child node of the tree;
2) Execute forward propagation step to the node taken in point 1);
3) Apply recursively the steps 1) and 2) to the remaining segments of the data item.

The first integer in Fig. 1a is 11010001_2 (209_{10}). In point 1, we take 6 most significant bits divided in three segments 11, 01 and 00. The first segment 11 has to be associated with a new node of the tree connected with the root by the rightmost edge (11). In point 2) the control flow is passed to the newly created node. Then the same steps have to be executed for the child node and for the second segment 01. Finally, the tree will include highlighted nodes shown in Fig. 5. We provide below additional details about unexplained records.

B. U-leaves N-ary Trees

If we implement the described above method then up to K items can be associated with each leaf. For example, when you consider the item 01110111_2 (119_{10}), a new node is added. Later on three new items (116, 117 and 118) will be associated with the same node that is underlined in Fig. 5.

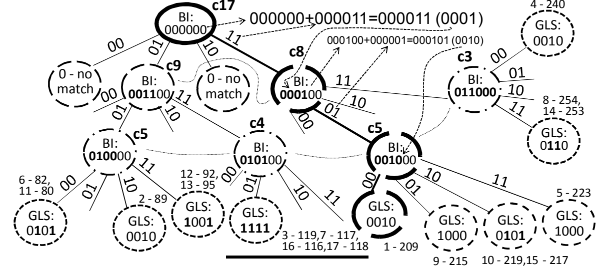

Figure 5. N-ary tree (N=4) that is built for data in Fig. 1a.

To represent multiple items associated with the same node a positional encoding with 2^K bits is used in such a way that in the code Bit_i=1 if and only if there exists an item associated with the considered node in which K less significant bits are equal to i_2 (*i.e.* binary code of i), else Bit_i=0. Finally four items 116 – 119 will be represented by the code 1111. Indeed, K=2 less significant bits of 116 are 00 and, therefore Bit_0 =1, two less significant bits of 117 are 01 and, thus, Bit_1=1, etc. Bits are numbered from 0 to 3 from right to left. Let us call *U-leaves tree* such a tree that has up to U=2^K (data) items associated with leaves. Thus, the considered N-ary tree is U-leaves tree and U=4.

Fig. 5 depicts the whole tree that is built for integers from Fig. 1a or Fig. 2 (non expanded nodes are either explicitly marked with *no-match* or the relevant edges are pendent). Data items associated with a leaf are shown in the form: <sequence>-<value>. For example, 6-82 (the bottom left leaf in Fig. 5) indicates that the value 82 is the sixth in the sequence shown in Fig. 2.

C. Representation of Trees in Memory

Clearly trees have to be coded in memory, which can be done using tree walk tables from [15]. Given a non-leaf node, the address of its leftmost child in the tree walk table is called the base index [15] of the node. The rest of the children are ordered sequentially, following the leftmost child. Therefore, to locate the i_{th} child, the index can be calculated by adding i to the base index. If a child is not associated with any GLS, a no-match (0) tag is stored in the entry. If for a node, all of its N children have no match, then the tree is not expanded and just a no-match tag is stored in the node itself. All other nodes (not containing no-match tags) are called *working nodes*. There is no cell in tree walk table for the root node and the first address (all zeros) is just considered to be the base index (BI) for children of the root. Other BIs are assigned sequentially for newly processed segments taking into account that every tree node necessitates a space in memory for N potential children. Thus, for our example, BI for the root is 000000; BIs within the way from the root to the first leaf (209) are 000100 and 001000 (they are selected sequentially with the increment N=4: *i.e.* 000000, 000100, 001000, etc.).

Now let us consider how N-ary U-leaves trees can be represented in memory. You can clearly see that the rightmost G-bit segment in M-K code is used to distinguish children of a tree node. For example, the first two segments (11 and 01) in the code 11 01 00 01 (209_{10}) are used to find addresses (**0001**00 and **0010**00) of children at the first and at the second levels (see Fig. 5). The third (rightmost) segment (00) enables us to calculate the address (**0010**00+00=001000) of the leaf containing positional code of the last K bits. Thus, we do not need to allocate an additional space and the required size S of memory is 2^{M-K} words for the maximum number of leaves plus additional words for storing intermediate nodes on the way from the root to the leaves. If $\lceil(M-K)/G\rceil=(M-K)/G$ then at the first level the number of additional words is $N=2^G$; at the second level - N^2; etc. Finally, at the level of leaves the number of words is $N^{\lceil(M-K)/G\rceil}$. Totally we need:

$$S_{max} = \sum_{i=1}^{\lceil(M-K)/G\rceil} N^i * \max((M-K-G),U) \text{ bits}$$

For the considered example S_{max}=(4+16+64)*max(4,4) =336 bits. If $\lceil(M-K)/G\rceil>(M-K)/G$ then the number of children of the root is less than $N=2^G$ and S_{max} is calculated as a product of S_{max} for *any child* of the root and the number of children of the root. For example, if M=9, K=G=2, S_{max}=2*(4+16+64)*max(4,4) =672 bits. For the majority of practical cases S<<S_{max}, because, as a rule Q<<2^M (see section III).

Fig. 6 demonstrates step by step representation of the tree in memory for our example (N=4 word fragments needed for newly allocated nodes are separated by dashed horizontal lines). The first data item is 11010001_2 (209_{10}). We take the leftmost fragment 11 and calculate an address of the child node as *BI of the root + 000011* (see Fig. 5). At the resulting address 000011 we record the first available base index (address for the next 4-bit segment, *i.e.* 0001, which means the first four children). Then we continue similar operation until arriving to the leaf (see Fig. 5 and points 1, 2, 3 in Fig. 6). As soon as we arrive to the leaf with the address 0010000 we record U bits as 0010 because the most significant bit in U corresponds to value 11 of K bits, the next bit in U corresponds to value 10 of K bits, etc. and the value of K bits for 11010001_2 is 01.

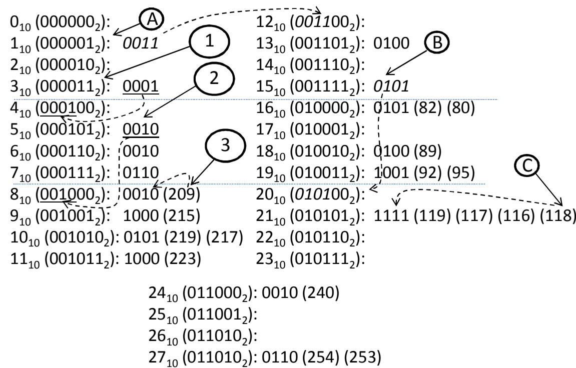

Figure 6. Representation of data from Fig. 1a in memory.

Similar steps have to be executed for all input data items (the steps 1-3 and A-C in Fig. 6 are shown for two integers 209 and 118). From Fig. 6 you can see that just 28 * 4 = 112 bits are needed to represent all data from Fig. 1a.

D. Sorting Data

Traversing the tree and sorting data can be done using the following recursive C function:

```
void sort (tree_node* node)  // state a0
{     static int level = 0;
      if (node!=0)                              {
        if (level==depth)
        {   if (node->l!=0)    // output data – state a1
            if (node->lm!=0)  // output data – state a2
            if (node->rm!=0) // output data – state a3
            if (node->r!=0)    // output data – state a4
        }
        else       {            level++;
                   if (node->l != 0)     sort(node->l);      // - state a5
                   if  (node->lm != 0)  sort(node->lm);  // - state a6
                   if (node->rm != 0)   sort(node->rm);  // - state a7
                   if (node->r != 0)     sort(node->r);      // - state a8
                              level--;     /* - state a9 */     }
                                        }
}                  // state a10
```

Where l, lm, rm, r are pointers to child nodes from left to right that are declared in the following structure:

```
struct tree_node
{              int value;
               struct tree_node *l, *lm, *rm, *r;   };
```

The number of steps from the root to leaves (the depth of the tree) is fixed. Comments in the function sort will be explained in the next section. Note that the function sort is recursive, but iterative technique can also be used with very similar results in performance and resource consumption.

V. IMPLEMENTATION DETAILS

All the proposed methods (see sections III and IV) have been implemented and tested in FPGAs. For methods of section IV the following two modules were developed: 1) for constructing N-ary U-leaves trees; and 2) for traversing the trees enabling the results of sorting to be produced. Synthesis and implementation in hardware were done using the model of a hierarchical finite state machine (HFSM) [16]. The complete design flow includes the following steps:

1) Describing the methods in high-level language (C in our case) and modeling in software;
2) Selecting the numbers G and K for the given M and representing trees with specified constraints in memory. Providing an interface with memory and implementing necessary operations;
3) Associating executable statements in C language (such as *output data* or *recursive calls*) with states of HFSM (see comments in the sort function above);
4) Optimization of HFSM using the methods [17];
5) Customizing VHDL templates [17], *i.e.* describing transitions between the states, operations in the states and recursive calls/returns using the methods [16,17];
6) Linking with the previously designed circuits for generating input data and presenting the results;

7) Synthesis and implementation of circuits using commercial tools (we used Xilinx ISE 11);
8) Uploading the generated bit-stream and testing in FPGA.

All the listed above steps are described in [17]. Optimization technique [17] enables the number of states (see comments in the program in section IV.D) to be reduced. Recursive calls/returns are implemented using stacks as HFSM memory [16]. Due to lack of space we cannot provide more details here. However, one feature should be highlighted. Let us look at Fig. 5. When we construct the tree we can think about potential parallelism at each level of the tree. Such levels are marked with dashed curves in Fig. 5. Suppose we would like to implement parallelism at the level of the bottom dashed curve. In such case all nodes with BI have to include counters (see letter c with a number in Fig. 5) that count the number of items associated with the relevant leaves. Such simple addition allows initial positions of data items associated with the corresponding sub-trees to be found. It simplifies parallel processing of tree branches.

Input data can be repeated, for example, 209, 89, 89,.... where the value 89 has appeared at least twice. In the current implementation repetitions are not taken into account. One potential method allowing the number of repeated values to be pointed out is the use of additional counters associated with the relevant data items. For example, instead of one-bit fields in codes of U-leaves, n>1 bits for counters can be allocated. However, the size of required memory will be increased and a constraint for the number of repeated items still exists. If the size of counters is n bits then the maximum 2^n repetitions can be indicated. Similar constrains exist for other known methods.

VI. Experiments and Results

The methods described above were tested in FPGA Spartan3E-1200E-FG320 (NEXYS-2 prototyping board of Digilent). For all experiments a random-number generator produced data items that were supplied to tested circuits. The generator and the circuits were built within the same FPGA. Synthesis and implementations of the circuits were done using Xilinx ISE and the sequence of steps described in section V.

Direct implementation of address-based method (see section III) permits *any set* of 18-bit data to be sorted (up to 2^{18} numbers). The following results permit the complexity and performance of the circuit to be evaluated:

- Number of slices (N_s): 326 (3%);
- Number of slice Flip Flops (N_{ff}): 57 (~0%);
- Number of 4 input LUTs (N_{LUT}): 578 (3%);
- Number of BRAMs (N_{BRAM}): 16 (57%);
- Maximum clock frequency (F): 155.376MHz;
- The number of clock cycles N_{in} needed to fill in BRAM is equal to Q assuming that each data item can be saved in BRAM during one clock cycle;
- The number of clock cycles N_{out} needed to sort data is equal to 2^{14} (one clock cycle is used to read 16-bit word from which up to 16 data items can be extracted during the same clock cycle).

The first implementation of the method based on tree-walk tables for N-ary U-leaves trees (see section IV) permits *any set* of 18-bit data to be sorted (up to 2^{18} numbers). The following results permit the complexity and performance of the circuit to be evaluated:

- N_s=562 (6%); N_{ff}=131 (1%); N_{LUT}=1048 (6%); N_{BRAM}=18 (64%); F=76.693MHz;
- The maximum number of clock cycles needed to sort data is 53 556. For different data sets the actual number of clock cycles varies from 7 000 to 53 556.

As you can see, the last implementation has bigger hardware resources and lower clock frequency. However, tree-walk tables permit more complicated data sort to be implemented in the same FPGA (the results of experiments are given below). Besides, direct use of address-based data sort requires resetting all one-bit fields to zero when repeating the sort with a new set of data (such resetting is not needed for tree-walk tables).

The second implementation of the method based on tree-walk tables for N-ary U-leaves trees (see section IV) enables us to sort sets of 20-bit data when $Q<2^M$ (in many cases Q is significantly less than 2^M). The following results permit the complexity and performance to be evaluated:

- N_s= 586 (6%); N_{ff}= 130 (1%); N_{LUT}= 1109 (6%); N_{BRAM}= 28 (100%); F=79.821 MHz;
- The actual number of clock cycles in different experiments varied from 8 500 to 86 000.

Note, that 20-bit data items cannot be sorted in a single FPGA Spartan3E-1200E-FG320 if we apply direct address-based technique described in section III. The maximum number of sorted data (with tree-walk tables) depends on a distribution of data within the interval from 0 to 2^{20}-1. This number is increased if there are many large clusters within the interval that are almost entirely filled in. For example, if all data are within the 20-bits interval 00----------------- and if N=4 then there is just one sub-tree (00) from the root and up to 2^{18} data items can be sorted. If all data are within the two 20-bits intervals 01------------------ and 10------------------ then there are two sub-trees from the root, etc. Some examples with 18-bit, 19-bit and 20-bit data produced by a random number generator that were implemented and tested in FPGA are given below:

- Q=100, N^{18}=7000-7700, N^{19}=8500-8900, N^{20}=9300-10000, where N^{18}, N^{19}, and N^{20} are numbers of clock cycles for 18-bit, 19-bit and 20-bit data accordingly;
- Q=300, N^{18}=16000-16600, N^{19}=19000-20000, N^{20}=22000-25000;
- Q=500, N^{18}=20000-24000, N^{19}=27000-29000, N^{20}=34000-36000;
- Q=1000, N^{18}=33000-36000, N^{19}=44000-47000, N^{20}=55000-58000;
- Q=2000, N^{18}=45000-47000, N^{19}=68000-70000, N^{20}=82000-83000;

Since the random number generator equally distributes data within the intervals 0,…,2^{19}-1 and 0,…,2^{20}-1, the examples above give the worst cases for 19-bit and 20-bit data. There is no worst case for 18-bit data because any set of data can be sorted.

Two strategies shown in Fig. 1 were considered. Data were sorted either based on their static representation (Fig. 1a) or sorted and resorted dynamically as soon as new data items are produced by a generator. Let us call the latter case dynamic sort and it is directly applicable to priority buffers [14] requiring fast ordering of new data that can

arrive at any time. Dynamic sort can also be implemented using binary trees with nodes containing three fields: a pointer to the left child node, a pointer to the right child node, and a value. The nodes are maintained so that at any node, the left sub-tree only contains values that are less than the value at the node, and the right sub-tree contains only values that are greater. Repeated values can be indicated by a counter associated with each node if required. In [18] sorting over binary trees was improved by applying optimization techniques and parallelization based on communicating HFSM. The results of experiments (presented in this paper and in [18]) demonstrate that the considered address-based sorting has advantages in performance and in the required hardware resources compared to data sort over binary trees. For example, the complexity of problems that can be solved in a single FPGA Spartan3E-1200E-FG320 was increased in more than 30 times.

The developed circuits are also faster than implementations in general-purpose software and in embedded to FPGA Power PC (the latter two cases were tested in HP EliteBook 2730p and PPC405 embedded to FPGA Virtex-4 FX12). It should be noted that quite complex problems can be processed in relatively simple and cheap FPGAs of Spartan-3 family. Performance is comparable with known results obtained for significantly more advanced FPGAs. Often the known methods were either modeled or just partially tested in available prototyping systems. Frequently, external onboard memories were used. Thus, the exact comparison in hardware is indeed difficult. All the considered circuits were entirely implemented in a single Spartan 3 FPGA and external resources were not used at all.

Besides, because there is no data dependency between tree branches, the algorithm permits individual sub-trees with any desired level of parallelism to be processed. Potential levels are shown by dashed curves in Fig. 5. For example, two sub-trees from the root can be processed in parallel at the first level. The counters c9 and c8 indicate the number of data items associated with the relevant sub-trees. For example c9 designates that there are 9 data items associated with the first sub-tree and they are the following: 82, 80, 89, 92, 95, 119, 117, 116, 118. Such counters permit positions in memory for parallel branches to be easily found. Indeed, the first sub-tree (with smaller data) requires 9 memory cells for sorted items. Thus, the first position in memory for keeping data of the second sub-tree is 10. Similarly, four sub-trees at the second level (see Fig. 5) can also be processed in parallel.

Preliminary tests on some examples demonstrate that parallel processing is faster that sequential and the hardware resources are increased insignificantly.

VII. CONCLUSION

The paper suggests address-based data sorting and sorting over N-ary trees (N>2) containing leaves with multiple data items (U-leaves). Two types of processing have been proposed: constructing/traversing the trees; and sorting leaf items by fast combinational circuits. Finally, very fast resorting for newly arriving data items has been achieved that is an important feature for data processing in stream applications. The results of experiments clearly demonstrate applicability and high efficiency of the proposed methods, which can easily be implemented in low cost widely available FPGAs.

ACKNOWLEDGMENT

This research was supported by the European Union through the European Regional Development Fund.

REFERENCES

[1] R. Mueller, J. Teubner, and G. Alonso, “Data processing on FPGAs”, Proc. VLDB Endowment 2(1), 2009.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stain, Introduction to Algorithms, 2nd edition, MIT Press, 2002.

[3] R. Mueller, Data Stream Processing on Embedded Devices, Ph.D. thesis, ETH, Zurich, 2010.

[4] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne, “High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs”, IEEE Symp. IPDPS, April, 2010.

[5] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore GPUs”, IEEE Symp. IPDPS, 2009.

[6] D. Cederman and P. Tsigas, “A practical quicksort algorithm for graphics processors”, Proc. 16th Annual European Symposium on Algorithms (ESA 2008), Sep. 2008, pp. 246–258.

[7] R. Baraglia, G. Capannini, F. M. Nardini, and F. Silvestri, “Sorting using Bitonic Network with CUDA”, 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR), Boston, USA, July, 2009.

[8] S. Chey, J. Liz, J.W. Sheaffery, K. Skadrony, and J. Lach, “Accelerating Compute-Intensive Applications with GPUs and FPGAs”, Proc. Symposium on Application Specific Processors, Anaheim, USA, June 2008, pp. 101-107.

[9] D.J. Greaves and S. Singh, “Kiwi: Synthesis of FPGA circuits from parallel programs”, Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008.

[10] S.S. Huang, A. Hormati, D.F. Bacon, and R. Rabbah, “Liquid Metal: Object-oriented programming across the hardware/software boundary”, European Conference on Object-Oriented Programming, Paphos, Cyprus, 2008.

[11] A. Mitra, M.R. Vieira, P. Bakalov, V.J. Tsotras, and W. Najjar, “Boosting XML Filtering through a scalable FPGA-based architecture”, Proc. Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, 2009.

[12] S. Rajasekaran and S. Sen, “Optimal and Practical Algorithms for Sorting on the PDM,” *IEEE Trans. Computers,* vol. 57, no. 4, pp. 547-561, Apr. 2008.

[13] R.D. Chamberlain and N. Ganesan, “Sorting on Architecturally Diverse Computer Systems”, Proc. 3rd Int’l Workshop on High-Performance Reconfigurable Computing Technology and Applications, November 2009.

[14] V. Sklyarov and I. Skliarova, "Modeling, Design, and Implementation of a Priority Buffer for Embedded Systems", Proc. 7th Asian Control Conference – ASCC’2009, Hong Kong, pp. 9-14, 2009.

[15] J.D. Davis, Z. Tan, F. Yu, and L. Zhang, “A practical reconfigurable hardware accelerator for Boolean satisfiability solvers”, Proc. 45th ACM/IEEE Design Automation Conf., pp. 780 – 785, 2008.

[16] V.Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control", IEEE Trans. on VLSI Syst., vol. 7, n. 2, pp. 222-228, 1999.

[17] V. Sklyarov, "Synthesis of Circuits and Systems from Hierarchical and Parallel Specifications", Proc. 12th Biennial Baltic Electronics Conf., Invited paper, Tallinn, Estonia, pp. 37-48, 2010.

[18] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, "Hardware Implementation of Recursive Algorithms". Proc. MWSCAS'2010, Seattle, USA, pp. 225-228, Aug. 2010.