2010 International Conference on Reconfigurable Computing

Parallel FPGA-based Implementation of Recursive Sorting Algorithms

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

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

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

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

***Abstract*—The paper describes the hardware implementation and optimization of parallel recursive algorithms that sort data using binary trees. Since recursive calls are not directly supported by hardware description languages, they are implemented using the model of a hierarchical finite state machine (HFSM). Parallel processing is achieved by constructing N binary trees (N>1) and applying concurrent sorting to N trees at the same time with the aid of N communicating HFSMs. The paper presents new results in: 1) parallel sorting algorithms; 2) FPGA-based parallel architectures; and 3) the analysis and comparison of alternative and competitive techniques for implementing parallel recursive algorithms. Experiments demonstrate that the performance of sorting operations is increased compared to previous implementations.**

Keywords-Circuit synthesis; Field programmable gate arrays; Finite state machines; Sorting; Tree data structures

I. INTRODUCTION

The main objective of this paper is to design, evaluate, and analyze hardware circuits that implement *parallel* recursive algorithms. Recursion is a powerful problem-solving technique [1] that may be applied to problems that can be decomposed into smaller sub-problems that are of exactly the same form as the original. Many examples that demonstrate the advantages of recursion are presented in [1-7]. An in-depth review and comparison of different approaches to hardware implementation appears in [7]. The advantages and disadvantages of recursive techniques in software are well known [1]. It has been shown [4] that recursion can be implemented in hardware more efficiently than in software. This is because any activation/termination of a recursive sub-sequence of operations can be combined with the execution of operations that are required by the respective algorithm. The number of states needed for the execution of recursion in hardware can be further reduced compared with software. Besides, such states are accumulated on stacks that can be constructed on built-in memory blocks, which are relatively cheap. The results obtained with some known methods for implementing recursive calls in hardware, reviewed in [7], have shown that many hardware circuits are faster than software programs executing on general-purpose computers.

Using and taking advantage of application-specific circuits in general and FPGA-based accelerators in particular have a long tradition in data processing [8]. The technique proposed in this paper can be used to design hardware circuits for run-time sorting and fast resorting for new incoming data items. Suppose, 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, 8-10] 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. The requirement of fast resorting is important, in particular, for the design of priority buffers (queues) that are needed for numerous practical applications. For example, in [11] a priority buffer (PB) stores pulse height analyzer events. Real-time embedded systems [12] employ a priority preemptive scheduling in which each process is given a particular priority (a small integer) when the system is designed. At any time, the system executes the highest-priority process. A smart agent scanning and selecting the data according to their priority is considered in [13]. A comprehensive method for managing priorities that requires fast resorting is described in [14].

The proposed technique is based on the use of tree-like data structures and it enables fast correction of the output sorted sequence after any change in the input. Besides, this paper concentrates on further improvements of recursive sorting algorithms through their parallelization that can be provided in hardware circuits. Recursion is supported using the model of a hierarchical finite state machine (HFSM) [4,5] and parallelism is achieved with the aid of communicating HFSMs implementing concurrently executing processes.

A brief summary of what is new is given below:

- Constructing and processing left and right sub-trees of certain tree nodes in parallel;
- Constructing and processing multiple (N>1) trees in parallel using N concurrent communicating HFSMs;
- FPGA-based architectures that enable the methods indicated above to be implemented in hardware;
- Experiments with the relevant FPGA-based circuits;
- In-depth analysis and comparison of alternative and competing techniques;
- Some improvements to sequential recursive sorting algorithms implemented in individual HFSMs.

The remainder of this paper is organized in five sections. Section II suggests new methods for parallel processing of binary trees for recursive sorting algorithms. Section III presents computational models and hardware architectures for the proposed methods. Section IV describes FPGA-based implementations and experiments. Section V analyzes the

978-0-7695-4314-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ReConFig.2010.30

IEEE computer society

results, compares different implementations, and suggests some improvements. The conclusion is in Section VI.

II. Parallel Processing of Binary Trees

Algorithms for many computational problems are based on the generation and traversal of a binary tree where the recursive technique is very practical. Suppose that the nodes of the tree contain three fields: a pointer to the left child node, a pointer to the right child node, and a value (*e.g.* an integer or a pointer to a string). 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 are indicated by a counter associated with each node. During the search for the proper place for a new data item we can:

1) Compare the new data item with the value of the root node to determine whether it 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) Repeat 1) and 2) with the selected node as the root.

Clearly recursion can naturally be applied in point 3. Let us assume that a sequence of input data is the following: 24, 17, 35, 30, 8, 61, 12, 18, 1, 25, 10, 15, 19, 20, 21, 40, 9, 7, 11, 16, 50. A tree for this sequence is shown in Fig. 1a.

Now we would like to use the tree to output the sorted data. Fig. 1b demonstrates a known algorithm [4,5] assuming that incoming data items are stored in RAM along with the addresses of the left (LA) and right (RA) sub-trees (Fig. 1c). All other details can be found in [4,5]. Let us call this known method *S1*.

In the first method (involving parallel processing) that we propose (let us call it *Sp1*), the left (such as the sub-tree with the root *17* in Fig. 1a) and the right (such as the sub-tree with the root *35* in Fig. 1a) sub-trees of the tree are traversed in parallel using the method *S1*. Intuitively we can guess that the result would depend considerably on the balance between the left and right sub-trees of the root. We would like to eliminate such dependency, and this is achieved by the second method below that uses parallel processing of two or more non-linked sub-trees.

In the second method that we propose (let us call it *Sp2*) N trees (N>1) are built and then processed in parallel. Let us consider an example for N=3. In this case the 1st, the 4th (N+1), the 7th (2N+1), etc. incoming data items are included into the 1st tree. Consequently, the 2nd, the 5th (N+1)+1, the 8th (2N+1)+1, etc. incoming data items are included into the 2nd tree and the 3rd, the 6th (N+1)+2, the 9th (2N+1)+2, etc. incoming data items are included into the 3rd tree. Fig. 2 shows N=3 trees that are built for our example (see the sequence of input data items given above and Fig. 1a).

Let us now compare Fig. 1a and 2. The maximum depth of the tree in Fig. 1a is 5 and the maximum depth of the trees in Fig. 2 is 3. Since the time needed for sorting depends on the depth of the trees [4] we expect that the performance of sorting algorithms that process data represented by a set of trees (see Fig. 2) is better.

Suppose a set of trees (such as that are shown in Fig. 2) is built and each tree is stored in the relevant processing memory (in a form shown in Fig. 1c). Since N=3, there are totally N=3 blocks of processing memory. To output the sorted data the following method is proposed:

1) The trees in N=3 processing memories are traversed in parallel and there are also N=3 dual-port output memories. Data items from the tree *n* (1≤*n*≤N) are saved in the output memory *n* applying the known method *S1* [4,5] for a single tree and using the first port. In other words, the output memory *n* is used to store the sorted data from the tree *n*.
2) This second point is executed in parallel with point 1) above but the second port for the output memories is used. At the beginning, all the addresses point to the first cell of the corresponding output memories. When all N addresses contain data items the smallest one (or the greatest one) is extracted and the address of the appropriate memory (from which the data item is extracted) is incremented. Whether the smallest or the greatest item is extracted depends on the chosen sorting strategy (*i.e.* either ascending or descending).
3) Points 1) and 2) above are repeated until all data items are sorted.

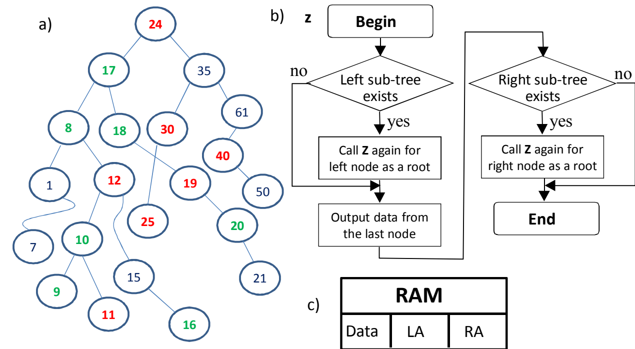

Figure 1. Binary tree for data sort (a); recursive algorithm for data sort (b); contents of memory (c)

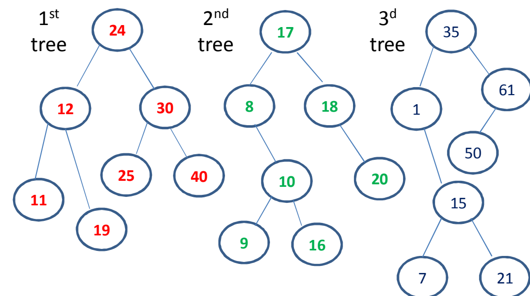

Figure 2. Sorting data based on N=3 trees that are built and processed in parallel

Let us consider an ascending sort for Fig. 2:

- At the first step we propagate on N=3 trees (see [4,5] for details) as follows: 24-12, 17-**8** - *save*, 35-**1** – *save*. The word *save* indicates storing data items 8 and 1 in the

second and the third output memory blocks. An item is saved when there is no left sub-tree.

- At the second step we propagate once again as follows: 12-**11** - *save*, 8-10, 1-15. Since all N=3 output memories contain data items at the first address for the second port, these data items (*i.e.* underlined and saved above 11, 8, and 1) are compared; the smallest item (*i.e.* 1) is sent from the output memory 3 to the external output; and the address of the second port for the output memory 3 is incremented (see Fig. 3a).
- At the third step we propagate once more as follows: 11-**12** - *save*, 10-**9** - *save*, 15-**7** – *save*. Since all N=3 output memories contain data items at the last (updated in the previous point) addresses for the second port these data items (*i.e.* 11, 8 and 7) are compared; the smallest item (*i.e.* 7) is sent from the output memory 3 to the external output; and the address of the second port for the output memory 3 is incremented (see Fig. 3a).

Subsequent steps are similar and they are shown in Fig. 3b. In Fig. 3 each output data item is supplied with a pair (x,y), where x is the step number, and y is the sequence number of the output data item.

As a result, the following sorted sequence of items is built in the output memory: 1, 7, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 24, 25, 30, 35, 40, 50, 61. The algorithm requires just 22 steps and this is approximately half that for the single tree depicted in Fig. 1a. Thus, we can reduce time for both the construction of the trees (see Fig. 2) and the output of data from the trees (see Fig. 3).

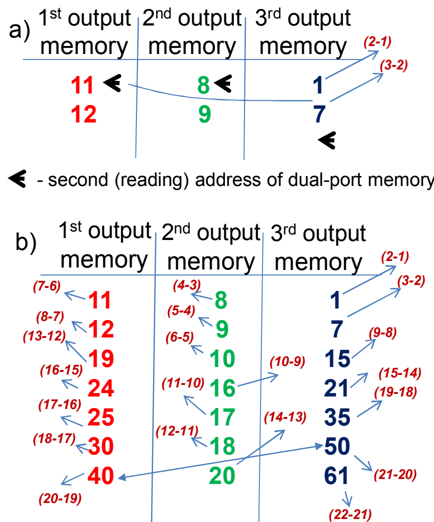

Figure 3. Parallel data sorting using algorithm *Sp2*

III. Computational Models and Hardware Architecture

The methods of parallel sorting described in section II were implemented using the following models:

- Parallel processing is done in concurrent communicating HFSMs (the details are given below);
- Each individual HFSM implements the known algorithm *S1*. Hierarchical graph-schemes (HGS) [4,5] have been used to describe *S1* (all necessary details can be found in [4,5]). A HGS can easily be converted to a HFSM and then formally coded in a hardware description language. The coding is done using the VHDL templates proposed in [4], which are easily customizable for the given set of HGSs. The resulting (customized) VHDL code is synthesizable and permits hardware circuits to be designed in commercial CAD systems.

All the VHDL projects that were developed are parametric (generic) and can easily be customized for a different number of parallel processes (concurrent HFSMs).

At the top level (at the level of the main root of the tree) the method *Sp1* is implemented using two simultaneously functioning HFSMs where one is a master and the other a slave. The master HFSM builds the tree, outputs the left sub-tree, and activates the slave HFSM, which works in parallel with the master HFSM. The slave HFSM outputs the right sub-tree. By adding a simple counter to the root that counts the number of left and right nodes during the construction of the tree, we can easily calculate the addresses for the sorted data in the output memory. Thus, processing both sub-trees can be done simultaneously. Naturally, more parallel branches can be introduced using cascade structures of more than two HFSMs that are activated for different sub-trees on certain paths from the main root.

The top-level hardware architecture that implements the algorithm *Sp2* is shown in Fig. 4. This architecture can be customized for different values of N.

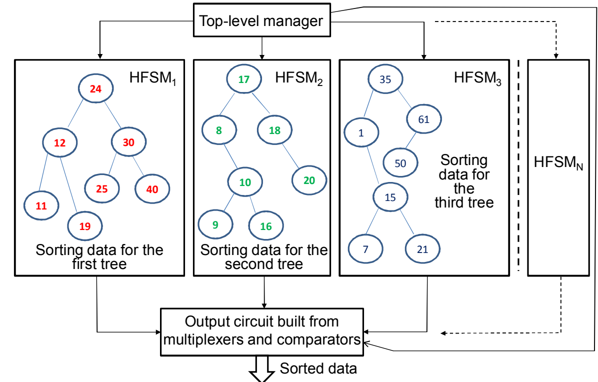

Figure 4. Top-level arcitecture that permits the algorithm Sp2 to be implemented

The top-level manager - TLM (top-level HFSM) receives input data items and distributes them between N concurrent HFSMs in such a way that the first data item is supplied to the first HFSM; the second data item is supplied to the second HFSM, etc. until the Nth HFSM. Data item N+1 can be supplied once again to the first HFSM as soon as the first HFSM completes the previous job (*i.e.* as soon as the first data item is allocated in the first tree), etc. Suppose N=3 (see Fig. 2). In this case, for the example that we considered in the previous section, three trees will be constructed in processing memories (by $HFSM_1$, $HFSM_2$ and $HFSM_3$) as

shown in Fig. 4. For example, the tree of $HFSM_1$ is stored in the 1st processing memory as follows: (*24 – address of 12 – address of 30*); (*12 – address of 11 – address of 19*); (*30 – address of 25 – address of 40*); etc. Any leaf of the tree is coded as: (*node value – no left node – no right node*). A similar technique is used if there is just one child node. The absence of a node is indicated by a specially allocated code.

As soon as the trees are constructed, the TLM instructs the output circuit to generate the sequence of sorted data (see the bottom part of Fig. 4). The output circuit is built from multiplexers and comparators and it implements the method shown in Fig. 3.

IV. FPGA-BASED IMPLEMENTATIONS AND EXPERIMENTS

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. A random-number generator produces 2^{11} items of data with a length of 14 bits (*i.e.* values in an interval between 0 and 16383). Values greater than 9999 are removed leaving 1200-1300 items available for further processing. These items are sorted using the following parallel FPGA-based implementations:

1. The known algorithm *S1* [4] briefly characterized in section II (see Fig. 1). The results are presented in Table I, where: the column *S1* indicates the number of clock cycles required for sorting; the *Data* column points to the number of data items that were sorted.
2. The proposed method *Sp1* (see section II and Table I) for two HFSMs: master and slave.
3. The proposed method *Sp2* (see section II) with different numbers N of trees (see columns of Table I for *Sp2* with different values of N: N=2, N=3 and N=4).

TABLE I. THE RESULTS OF EXPERIMENTS FOR POINTS 1-3

Data	*S1*	*Sp1*	*Sp2*			*Left/Right*
			N=2	*N=3*	*N=4*	
1286	5143	6424	2581	1751	1362	1/1284
1278	5111	6124	2591	1731	1370	53/1224
1248	4991	5524	2525	1700	1324	143/1104
1211	4843	5129	2474	1667	1298	185/1025
1216	4863	4749	2462	1701	1296	266/949
1248	4991	4579	2522	1689	1326	332/915
1203	4811	3714	2420	1675	1290	460/742
1228	4911	3499	2490	1661	1295	528/699
1212	4847	3279	2440	1684	1291	556/655
1230	4919	3101	2479	1666	1302	623/606
1305	5919	3533	2622	1799	1411	742/562
1259	5035	3727	2541	1708	1329	822/436
1230	4919	3629	2507	1679	1325	799/430
1304	5215	3853	2632	1780	1414	849/454
1276	5103	4167	2573	1775	1399	963/312
1225	4899	4101	2479	1671	1312	958/266
1225	4899	4185	2479	1663	1305	986/238
1199	4795	4354	2432	1675	1281	1051/147
1309	5235	5175	2660	1808	1397	1288/20
1204	4815	4816	2444	1698	1302	1203/0

Table I summarizes the results of experiments with the algorithms *S1*, *Sp1* and *Sp2*. An additional column *Left/Right* shows the number of nodes in the left and right sub-trees from the root. It permits to examine dependency of the results on the balance between the left and right sub-trees.

Table II presents the maximum clock frequency (*F*) and FPGA resources (the number of slices - *Slices*, the number of LUTs – *LUTs*, the number of block RAMs - *BRAMs*) needed for different implementations indicated in points 1-3 above. Note, that all the circuits were physically implemented and tested in the prototyping board NEXYS-2. Three lines at the bottom of Table II will be explained in section V.

The algorithm *S1* was also described in C++ and implemented in software. Other algorithms (*Sp1*, *Sp2*) are hardware-oriented and their advantages have appeared just in hardware. The same data were used for the software implementations. The results were produced on HP EliteBook 2730p (Intel Core 2 Duo CPU, 1.87 GHz) computer. Table III indicates times per sorted data item for software and for hardware in ns.

TABLE II. IMPLEMENTATION DETAILS

Algorithm	*F (MHz)*	*Slices*	*LUTs*	*BRAMs*
S1	101.36	714	1391	5
Sp1	102.6	1115	2203	8
Sp2 (N=2)	90.37	1297	2480	8
Sp2 (N=3)	87.52	1963	3736	12
Sp2 (N=4)	83.18	1707	3209	12
Sp3 (N=2)	77.00	1667	3203	10
Sp3 (N=3)	69.34	2518	4814	15
Sp3 (N=4)	73.32	2140	4088	12

TABLE III. THE RESULTS IN HARDWARE AND SOFTWARE

Data	*S1 (ns)*	*Sp1 (ns)*	*Sp2(ns)*			*Software (ns)*
			N=2	*N=3*	*N=4*	
1286	39.5	48.7	22.3	15.6	12.7	147.2
1278	39.5	46.7	22.5	15.5	12.9	151.1
1248	39.5	43.1	22.5	15.6	12.8	153.6
1211	39.5	41.3	22.7	15.7	12.9	167.3
1216	39.5	38.1	22.5	16.0	12.8	154.1
1248	39.5	35.8	22.5	15.5	12.8	148.6
1203	39.5	30.1	22.4	15.9	12.9	155.6
1228	39.5	27.8	22.5	15.5	12.7	151.5
1212	39.5	26.4	22.4	15.9	12.8	148.5
1230	39.5	24.6	22.4	15.5	12.7	155.2
1305	44.7	26.4	22.3	15.8	13.0	147.8
1259	39.5	28.9	22.4	15.5	12.7	149.1
1230	39.5	28.8	22.6	15.6	13.0	150.0
1304	39.5	28.8	22.4	15.6	13.0	148.7
1276	39.5	31.8	22.4	15.9	13.2	148.0
1225	39.5	32.7	22.5	15.6	12.9	151.0
1225	39.5	33.3	22.5	15.5	12.8	148.8
1199	39.5	35.4	22.5	16.0	12.8	157.6
1309	39.5	38.6	22.6	15.8	12.8	149.6
1204	39.5	39.0	22.6	16.1	13.0	157.0

Experiments in software were done as follows. All C++ output operators were removed and statements that permit measuring the execution time were inserted just before and immediately after the execution of the recursive sorting procedure. The time T_{ns} in nanoseconds was measured in software as an average of one hundred executions of the algorithm over the same set of data. Table III indicates values T_{ns}/<*number of sorted data items*>. For hardware these times are calculated as follows: (<clock period based on frequency from Table II>*<number of clock cycles from Table I>)/ <number of data items>.

Fig. 5 permits to compare acceleration of resorting for a new portion (that includes from 10 to 120 data items) comparing with complete resorting of 2^{12} data items. Such resorting, in particular, is important for management of priorities considered in [14].

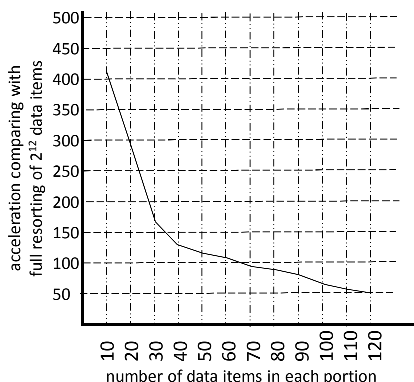

Figure 5. Resorting data items compared with full sort

V. Analysis of the Results and Comparisons

A. *Performance*

As can be seen from Table III, the performance of sorting operations in FPGA is approximately 3.75 times faster than in software for the known algorithm *S1* and approximately 11.7 times faster than in the software for the proposed algorithm *Sp2 (N=4)*. Table I compares different parallel sorting methods in terms of clock cycles. In particular, we can conclude (see also Table II) that it is important to deeper optimize circuits for *Sp2* to increase *F*. The proposed method (*Sp2: N=4*) gives better performance than *S1* in approximately 3 times. It is difficult to draw any particular conclusion for the *Sp1*, because the results depend considerably on the balance between the left and right sub-trees of the root. For example, in the first line of Table III (see also Table I), where the tree is completely unbalanced, the *Sp1* gives the worst result. However, in case of a well-balanced tree, the results of *Sp1* are significantly better than for *S1* (see lines in the middle of Table III). In the best case the performance of *Sp1* is 6.3 times better than for software, 1.6 times better than for the *S1* and still 1.9 times worse than for *Sp2* (*N=4*). We believe that additional research efforts are needed to explore the potential for parallel sorting and the results look promising. There are at least two ways for potential improvements. *Sp1* can be modified in such a way that the dependency of the results on the balance between the left and right sub-trees is either decreased or completely eliminated. Indeed, simultaneously functioning HFSMs (see section III) might implement a more intelligent strategy. For example, the first HFSM could implement the algorithm *S1* and when the HFSM detects that for the current node the left and the right sub-trees are well balanced, the second HFSM is activated. In this approach each node of the tree would have to include an additional field indicating the number of child nodes on the left and on the right (such fields can easily be filled in during the construction of the tree). The second way assumes improvements to the sorting algorithm, such as *S1*.

B. *Potential improvements*

Let us consider the basic ideas that make it possible for *S1* to be improved. The embedded dual-port memory blocks (available for different FPGAs) permit two cells (such as that shown in Fig. 1c) to be accessed simultaneously through LA and RA. Each cell holds *data+LA+RA*. Initially the input buffer register is loaded with *data+LA+RA* for the root of the sorting tree. Firstly, we examine the left sub-tree. If the left sub-tree (node) exists then it is checked again to determine whether the left node also has either left or right sub-trees (nodes). 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 this case the node in the input buffer register holds the second smallest value and the relevant data value is sent to the output. Secondly, we perform similar operations for right nodes. The dual-port RAM allows LA and RA (for each word in the dual-port RAM) to be examined independently. Finally, compared with the *S1* in Fig. 1, for some sub-trees of the tree we are able to check more than one node at the same time (*e.g.* a left node and a right node or a left node and a left node of the first left node), which reduces the processing time.

The improved algorithm *S1* has been implemented in FPGA and was used in the parallel sort described in section III for three values of N (let us call the relevant parallel algorithms *Sp3 (N=2)*, *Sp3 (N=3)* and *Sp3 (N=4))*. Recursion is applied in much the same way as in Fig. 1. The maximum clock frequency (*F*) and FPGA resources for the algorithms are given in three bottom lines of Table II. Table IV summarizes the results.

An analysis of Table IV shows that the *Sp3 (N=2* and *N=3)* is better than *Sp2*, but *Sp3 (N=4)* gives nearly the same results as *Sp2*. We think that the *Sp3* is better for deeper trees (the less N the deeper the tree). Thus, *Sp3* can be tested additionally for significantly larger number of data items. This is a direction for future work. We tested also that the considered algorithms can be improved through combining tree-like structures with sorting networks [15] using two potential techniques that are: 1) processing groups of items instead of individual items; and 2) using multiple

trees. The experiments in such direction are almost finished but the relevant work is outside of the scope of this paper.

TABLE IV. PARALLEL SORTING USING THE IMPROVED ALGORITHM S1

Data	*Sp3 (clock cycles)*			*Sp3(ns per data item)*		
	N=2	*N=3*	*N=4*	*N=2*	*N=3*	*N=4*
1286	1826	1314	1300	18.4	14.7	13.8
1278	1803	1303	1301	18.3	14.7	13.9
1248	1743	1272	1275	18.1	14.7	13.9
1211	1707	1251	1235	18.3	14.9	13.9
1216	1698	1239	1243	18.1	14.7	13.9
1248	1752	1284	1278	18.2	14.8	14.0
1203	1653	1240	1225	17.8	14.9	13.6
1228	1741	1262	1252	18.4	14.8	13.9
1212	1688	1253	1235	18.1	14.9	13.9
1230	1727	1266	1252	18.2	14.8	13.9
1305	1843	1341	1338	18.3	14.8	14.0
1259	1765	1288	1281	18.2	14.8	13.9
1230	1722	1261	1260	18.2	14.8	14.0
1304	1837	1337	1329	18.3	14.8	13.9
1276	1772	1306	1298	18.0	14.8	13.9
1225	1727	1264	1253	18.3	14.9	14.0
1225	1721	1261	1253	18.2	14.8	14.0
1199	1692	1228	1223	18.3	14.8	13.9
1309	1863	1351	1334	18.5	14.9	13.9
1204	1671	1231	1222	18.0	14.7	13.8

C. *Resource consumption and summary*

From Table II we can see that parallel implementations of recursive sorting algorithms support a generally smaller maximum clock frequency (*F*) and consume more resources. In spite of the continuous growth in the number of FPGA slices and embedded blocks, additional optimization of the designed circuits would be desirable and can be identified for future work. It is important to note that the proposed hardware implementations are faster than similar software implementations in spite of the chosen FPGA being very cheap and not very fast. The use of significantly more advanced and faster FPGAs available on the market (*e.g.* Virtex-5/6 families) would permit even faster recursive sorting algorithms to be implemented (*i.e.* to gain even more advantages over software and known hardware implementations). Currently the considered algorithms are planned to be implemented on the basis of XUPV5-LX110T development system (XC5VLX110T FPGA of Virtex-5 family) with 64-bit wide 256Mbyte external memory. Availability of such memory will permit significantly more complicated tree-like data structures to be processed. Considerable improvements have also been achieved in HFSM models and methods of synthesis, which permit additional optimization. We do hope that the results of the in-depth experiments with physical implementations in FPGA circuits presented in the paper will give assistance to deciding what is more important and which implementation should be chosen for a particular system.

VI. CONCLUSION

The paper suggests new hardware-oriented parallel implementations of recursive sorting algorithms and clearly demonstrates the advantages of the innovations proposed based on prototyping in FPGA and abundant experiments. Recommendations and discussions are also presented. Obviously, the results of the paper are not limited to just recursive sorting. They have a wide scope and can be applied effectively to numerous systems that implement recursive algorithms over tree-based structures.

ACKNOWLEDGEMENTS

This research was supported by the European Union through the European Regional Development Fund (Tallinn University of Technology).

The authors would like to thank Ivor Horton for very useful comments and suggestions.

REFERENCES

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

[2] T. Maruyama, M. Takagi, T. Hoshino, "Hardware implementation techniques for recursive calls and loops", Proc. 9th Int. Workshop on Field-Programmable Logic and Applications - FPL'99, Glasgow, UK, 1999, pp. 450-455.

[3] T. Maruyama, T. Hoshino, "A C to HDL compiler for pipeline processing on FPGAs", Proc. IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'2000, CA, USA, 2000, pp. 101-110.

[4] V. Sklyarov, "FPGA-based implementation of recursive algorithms," Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs, vol. 28/5-6, 2004, pp. 197–211.

[5] V. Sklyarov, I. Skliarova, B. Pimentel, "FPGA-based Implementation and Comparison of Recursive and Iterative Algorithms", Proc. 15th Int. Conference on Field-Programmable Logic and Applications - FPL'2005, Finland, 2005, pp. 235-240.

[6] S. Ninos, A. Dollas, "Modeling recursion data structures for FPGA-based implementation", Proc.18th Int. Conference on Field Programmable Logic and Applications – FPL'08, Heidelberg, Germany, 2008, pp. 11-16.

[7] I. Skliarova, V. Sklyarov, "Recursion in Reconfigurable Computing: a Survey of Implementation Approaches", Proc. 19th Int. Conference on Field Programmable Logic and Applications – FPL'2009, Prague, Czech Republic, 2009, pp. 224-229.

[8] R. Mueller, J. Teubner, G. Alonso, "Data processing on FPGAs", Proc. VLDB Endowment 2(1), Aug., 2009. pp. 910-921.

[9] R.D. Chamberlain, N. Ganesan, "Sorting on Architecturally Diverse Computer Systems", Proc. 3rd Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications, Portland, USA, Nov, 2009, pp. 39-46.

[10] X. Ye, D. Fan, W. Lin, N. Yuan, P. Ienne, " High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs ", Proc. of the 24th IEEE Int Symp. IPDPS, Atlanta, USA, April, 2010.

[11] R.A. Mewaldt, C.M.S. Cohen, W.R. Cook, *et al*., "The Low-Energy Telescope (LET) and SEP Central Electronics for the STEREO Mission", Space Science Rev., 136, 2008, pp. 285–362.

[12] S.A. Edwards, "Design Languages for Embedded Systems", Computer Science Technical Report CUCS-009-03, Columbia University, May, 2003.

[13] H.T. Sun, "First Failure Data Capture in Embedded System", Proc. of IEEE IIT, 2007, May 17-20, Chicago, USA, 2007, pp. 183-187.

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

[15] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd edn., Addison-Wesley, 1998.