Application-specific Hardware Accelerator for Implementing Recursive Sorting Algorithms

Dmitri Mihhailov [#1], Valery Sklyarov [*2], Iouliia Skliarova [*3], Alexander Sudnitson [#4]

[#] *CED, Tallinn University of Technology*
Tallinn, Estonia
[1] d.mihhailov@ttu.ee
[4] alsu@cc.ttu.ee

[*] *DETI/IEETA/HIPEAC, University of Aveiro*
Aveiro, Portugal
[2] skl@ua.pt
[3] iouliia@ua.pt

***Abstract*—The paper is dedicated to hardware accelerators for data sorting using tree-based recursive algorithms. Since recursive calls are not directly supported by hardware description languages, they are implemented using the model of a hierarchical finite state machine. The paper presents new results in: 1) computational models and hardware architectures; 2) optimization and parallel execution of recursive sorting algorithms; 3) the analysis and comparison of alternative and competitive techniques for implementation of recursive sorting algorithms both in hardware and software. Experiments with the proposed FPGA-based hardware accelerators demonstrate that the performance of sorting operations is increased compared to known implementations.**

I. Introduction

Recursive algorithms are frequently used in a wide range of practical applications [1] and the most often for various kinds of binary search. Let us consider an example of using a binary tree for sorting data [1]. 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. Methods [2] permit to process such a tree in hardware enabling incoming data items to be sorted. This is achieved with the aid of two recursive algorithms A_1 and A_2 in such a way that A_1 builds the tree and A_2 outputs the sorted data items from the tree. The algorithms A_1 and A_2 have two important features: 1) it is not necessary to rebuild the tree in order to insert new data items; and 2) the number of data items to be sorted might be unknown. Such features are required, in particular, for priority buffers (queues) that store an incoming (sequential) flow of data and allow outputs to be selectively extracted from the buffer for processing. Buffers of this type are required for such practical applications as [3,4] and they can be constructed in hardware on the basis of recursive sorting algorithms. These algorithms are also very common for other engineering applications both in hardware and in software [1,2].

978-1-4244-8982-4/10/$26.00 ©2010 IEEE

Different methods for implementation of recursive algorithms in hardware are considered in [2,5-10]. This paper evaluates, analyzes, and improves hardware circuits that implement recursive algorithms with primary objective to use such circuits as FPGA-based accelerators targeted to data sorting and associated problems such as the considered above priority buffers. Many examples that demonstrate the advantages of recursion are presented in [1,2,5-14]. An in-depth review and comparison of different approaches to hardware implementations appears in [14].

The remainder of this paper is organized in four sections. Section II suggests new parallel hardware-oriented algorithms for recursive data sorting and improves the known sequential algorithms. Section III is dedicated to models of hierarchical finite state machines (HFSM) that permit recursive algorithms to be implemented in hardware. Section IV presents the results of experiments, comparisons and analysis of the models, methods and implementations in hardware and software. The conclusion is given in Section V.

II. Recursive Data Sorting

A. A Known Technique

Algorithms for many computational problems are based on the generation and traversal of a binary tree where the recursive technique is very helpful. Let us assume that a tree for sorting has already been built. Now we would like to use the tree to output the sorted data. A known algorithm [2], mentioned in the introduction as A_2, solves this problem. Note that algorithm A_2 is sequential. We would like to accelerate sorting and this is achieved through parallelization considered below.

B. Parallel Implementations

Suppose input data items arrive in a sequence shown at the bottom of Fig. 1. In the first method that we propose (let us call it *Sp*), the left (such as the sub-tree with the root *b* in Fig. 1 and the right (such as the sub-tree with the root *c* in Fig. 1) sub-trees of the main tree root (*a* in Fig. 1) are built and traversed in parallel using the algorithms A_1 and A_2. There are two simultaneously functioning HFSMs that are a master and

a slave. The master HFSM takes the first incoming data item; builds the root of the tree; and activates the slave HFSM. After that the master and the slave HFSMs implement the algorithm A_1 for the left (see a fragment enclosed by a dashed curve in Fig. 1) and the right (see a fragment outside of the closed dashed curve in Fig. 1) sub-trees in parallel. As soon as the incoming flow of input items is ended, the master HFSM outputs the left sub-tree (the nodes *b*, *d*, *e*, *h*, *i* in Fig. 1); and the slave HFSM outputs the right sub-tree (the nodes *c*, *f*, *g*, *j*, *k*, *l* in Fig. 1). Thus, two algorithms A_2 are executed in parallel by two HFSMs.

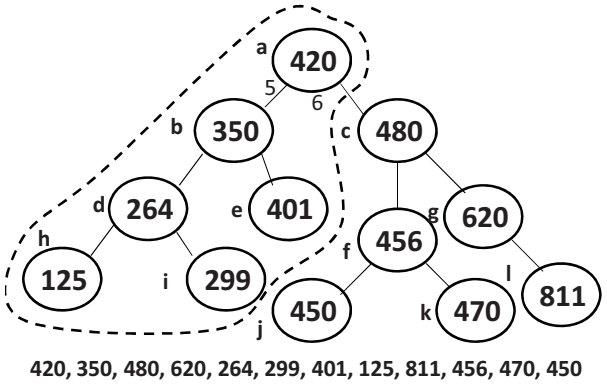

Fig. 1. Binary tree for data sort

By adding simple counters to the root that count the number of left and right descendant nodes during the construction of the tree, we can easily calculate the addresses for the sorted data in an output memory. If needed, more parallel branches can be introduced using cascade structures.

However, there is one significant limitation. If the tree is unbalanced, one HFSM will need significantly more time for data processing than the other. In the second method that we propose (let us call it *Spp*), the master HFSM activates the slave HFSM just if there is a sufficient number of processing steps. Each node of the tree is provided with two additional fields indicating the number of nodes in the left and in the right sub-trees accordingly. For the algorithm A_2 two HFSMs, the master ($HFSM_1$) and the slave ($HFSM_2$), begin their job at the same time. Both the master and the slave HFSMs repeat the same steps to remember the way from the root for further backward. In each tree node, the master HFSM evaluates the number of nodes for forward propagation to the left N_l and to the right N_r. The latter (N_r) includes the number of nodes in the right sub-tree of the current node and the number of nodes in all right sub-trees encountered at the previous propagation steps. As soon as N_l and N_r differ in some predefined value (normally either 0 or 1), the master HFSM takes responsibility for sorting of the last root and the left sub-tree; and instructs the slave HFSM to continue sorting with the remainder of the tree.

C. Improvements of Sequential Algorithms

The known algorithm [2] can be in hardware through the use of dual-port memories and algorithmic modifications. Let us use dual-port memories to store words for both left and right sub-trees of nodes and a buffer register to store the currently selected node in the format: *data+LA+RA* (LA is the address of the left sub-tree and RA is the address of the right sub-tree). The dual-port memory permits two words to be accessed simultaneously through LA and RA of the buffer register. Each word stores similar information to the buffer register (i.e. *data+LA+RA*) for the left and for the right nodes.

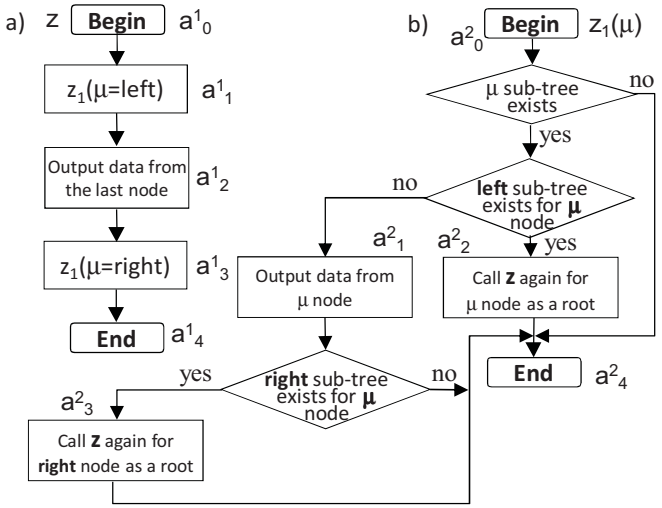

Fig. 2. Improvement of sequential algorithm: the top-level module (a) and the module $z_1(\mu)$ (b)

At each recursive step up to three nodes can be processed within the same time slot. Thus, descendants of the child nodes can be analyzed to reduce the number of recursive calls/returns during the traversal procedure compared to the known method. The block diagram of the improved sequential algorithm is presented on Fig. 2.

III. Implementations of Recursive Algorithms

A. HFSM with Explicit Modules

There are a variety of synthesizable specifications for recursive algorithms and we will use hierarchical graph-schemes (HGS) [15] that can be seen as flow-charts with some predefined constraints. For example, Fig. 2 depicts the improved algorithms A_2 in form of HGSs. A HGS can easily be converted to a HFSM using the method [15] and then formally coded in a hardware description language such as VHDL. The coding is done using the template proposed in [2], which is easily customizable for the given set of HGSs. The resulting (customized) VHDL code is synthesizable and permits the hardware to be designed in commercially available CAD systems, such as Xilinx ISE.

The HFSM model, as describes in [2], has the following distinctive features. There are two stack memories with $\lceil \log_2 Q \rceil$ bits for modules (HGSs) and $\lceil \log_2 R \rceil$ bits for states (Q is the number of modules and R is the maximum number of states in one module). States in different modules can be assigned the same labels (i.e. the same codes). Each module

(HGS) is coded explicitly. That is why we will call this model *HFSM with explicit modules*.

B. HFSM with Implicit Modules

We propose a new model, which is called *HFSM with implicit modules*. It has a single stack of states and a state register. In this case states in different modules have to be assigned different labels (different codes). The stack is needed just to know which state has to be the target of the transition when a called module is terminated. In any module all necessary state transitions are realized through the register, much like it is done in a conventional FSM.

IV. Implementation and Analysis

A. 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. 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:

1) The known [2,7] A_{known} and the improved $A_{improved}$ (see Fig. 2) sequential algorithms implemented on the basis of HFSMs with explicit and implicit modules. Hardware circuits based on different HFSM models differ in the used resources and the maximum attainable clock frequency. However, there is no difference for these models in the number of clock cycles required for data sorting by the same algorithm (because this number depends only on sorting algorithms to be chosen). The results are presented in Table I. The columns A_{known} and $A_{improved}$ indicate the number of clock cycles for sorting; the *Data* column shows the number of data items that were sorted. An additional column *Left*/*Right* shows the number of nodes in the left and right sub-trees from the root, which is needed to examine the dependency of the results on the balance between the left and right sub-trees.

2) Two proposed parallel algorithms *Sp* and *Spp* (described in sub-section II.B) implemented with the aid of one master and one slave HFSM with implicit modules. Each HFSM (master and slave) executes the algorithm $A_{improved}$ and both HFSMs work in parallel. The results are presented in columns *Sp* and *Spp* of Table I.

3) The known [2,7] algorithm A_{known} described in C++ program and implemented in software. Other algorithms referenced in points 1 and 2 above are hardware-oriented and their advantages have appeared just in hardware. The same data (randomly generated) were used for the software implementations. The results were produced on HP EliteBook 2730p (Intel Core 2 Duo CPU, 1.87 GHz) computer. Table II provides sorting time per data item (in ns).

Table III presents the maximum attainable clock frequency (F) in MHz and FPGA resources (the number of slices – S and the number of block RAMs - B) needed for different implementations indicated in points 1 and 2. Here in the columns $HFSM_{LUT}$ the stacks are built from LUTs and in the columns $HFSM_{BRAM}$ the stacks are built from the embedded in FPGA block RAMs.

TABLE I. THE RESULTS OF EXPERIMENTS FOR POINTS 1 AND 2

Data	A_{known}	$A_{improved}$	Sp	Spp	Left/Right
1211	4843	3373	5129	2701	185/1025
1216	4863	3393	4749	2546	266/949
1248	4991	3486	4579	2584	332/915
1203	4811	3350	3714	2618	460/742
1228	4911	3432	3499	2702	528/699
1212	4847	3350	3279	2506	556/655
1230	4919	3470	3101	2588	623/606
1305	5919	3596	3533	2814	742/562
1259	5035	3496	3727	2746	822/436
1230	4919	3419	3629	2602	799/430
1304	5215	3610	3853	2796	849/454
1276	5103	3564	4167	2734	963/312
1225	4899	3417	4101	2538	958/266
1225	4899	3420	4185	2542	986/238
1199	4795	3319	4354	2398	1051/147

TABLE II. COMPARISON OF THE RESULTS OF IMPLEMENTATIONS IN HARDWARE AND IN SOFTWARE

Data	A_{known}	$A_{improved}$	Sp	Spp	Software
1211	39.5	33.6	41.1	22.1	167.3
1216	39.5	33.7	37.9	20.7	154.1
1248	39.5	33.7	35.6	20.5	148.6
1203	39.5	33.6	30.0	21.5	155.6
1228	39.5	33.7	27.7	21.8	151.5
1212	39.5	33.4	26.3	20.5	148.5
1230	39.5	34.0	24.5	20.8	155.2
1305	39.5	33.3	26.3	21.3	147.8
1259	39.5	33.5	28.7	21.6	149.1
1230	39.5	33.5	28.6	20.9	150.0
1304	39.5	33.4	28.7	21.2	148.7
1276	39.5	33.7	31.7	21.2	148.0
1225	39.5	33.7	32.5	20.5	151.0
1225	39.5	33.7	33.2	20.5	148.8
1199	39.5	33.4	35.2	19.8	157.6

TABLE III. IMPLEMENTATION DETAILS

Implementation	HFSMLUT			HFSMBRAM		
	F	S	B	F	S	B
HFSMe A_{known}	101	714	5	70	149	7
HFSMi A_{known}	111	597	5	97	131	6
HFSMe $A_{improved}$	83	790	6	69	197	7
HFSMi $A_{improved}$	63	672	6	62	232	6
HFSMe Sp	103	1115	8	67	586	10
HFSMi Sp	103	986	8	68	574	10
HFSMe Spp	101	1218	8	71	692	10
HFSMi Spp	101	1194	8	78	626	10

B. Analysis of the results and comparisons

1) *Performance:* Tables I and II permit the performance of different algorithms to be compared. From the results in the columns A_{known} and $A_{improved}$ you can see that the improved sequential algorithm is faster than the known sequential algorithm for all lines. It is difficult to draw any particular conclusion for the algorithm Sp, because the results depend

considerably on the balance between the left and right sub-trees of the root. If the tree is completely unbalanced, the results of *Sp* are not good at all. However, in case of a well-balanced tree, the results from the algorithm *Sp* are significantly better than the results of the known algorithm. Besides, *Sp* gives base for *Spp* that provides the best performance and the results of *Spp* do not depend on the balance between the left and right sub-trees of the root.

As can be seen from Table II, the hardware implementations are faster than software implementations for all the experiments even though the clock frequencies of the FPGA and the PC differ significantly. In spite of a significantly lower clock frequency, the performance of sorting operations in FPGA for the algorithm *Spp* is more than 7 times faster than for software implementations. We believe that the results of *Spp* would be even better if more than one slave HFSM would be introduced. This is a direction for future work.

2) *Resource consumption:* As can be seen from Table III, the circuits in which the stacks are built from block RAMs consume the least number of FPGA slices and they require 1 or 2 additional block RAMs. However, the maximum achievable clock frequency for such circuits is lower than for similar implementations with the stacks built from CLBs (from LUTs). If you compare the HFSM models described in section III (compare the lines of Table III marked with $HFSM_e$ and $HFSM_i$), you can see that HFSM with implicit modules requires less hardware resources.

3) *Clarity of the specifications and potential for optimization:* Let us compare the characteristics of the models in section III. Obviously, the $HFSM_i$ with implicit modules is less resource consuming. Another advantage is that there is an opportunity to apply known optimization methods that have been developed for conventional state machines. The HFSM model with explicit modules ($HFSM_e$) is not so well suited for such optimization, mainly because states in different modules can be assigned the same codes. However, the $HFSM_i$ also possesses disadvantages, namely that modules become implicit and cannot be updated and refined easily. Although the HGSs for the $HFSM_i$ are the same and all features are supported, modularity, hierarchy and recursion become less clear at the implementation level. Thus, the $HFSM_e$ is better in terms of clarity, reuse and the ease with which modifications can be made. The $HFSM_i$ is better in terms of resources, performance and the potential for optimization. Finally, it is up to the designer to decide what is more important and which model should be chosen for a particular system.

V. Conclusion

The paper suggests new hardware-oriented parallel algorithms and improvements of known sequential algorithms for recursive data sorting, and clearly demonstrates the advantages of the innovations proposed based on prototyping in FPGA and abundant experiments. Recommendations and discussions are also presented.

The presented results demonstrate good capabilities of FPGAs for accelerating recursive algorithms over tree-like data structures. The use of significantly more advanced and faster FPGAs available on the market (e.g. Virtex-5 family) would permit even faster recursive sorting (i.e. to gain even more advantages over software). Preliminary test of an interface between the FPGA-based prototyping board (NEXYS-2 of Digilent) and a PC computer permits to conclude that FPGA can be efficiently used to accelerate software programs that process tree-like data structures.

Acknowledgment

The authors would like to thank Ivor Horton for very useful comments and suggestions. This research was supported by the European Union through the European Regional Development Fund.

References

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

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

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

[4] H.T. Sun, “First Failure Data Capture in Embedded System”, Proceedings of IEEE IIT, 2007, May 17-20, Chicago, USA, pp. 183-187, 2007.

[5] T. Maruyama, M. Takagi, and 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.

[6] T. Maruyama and 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.

[7] V. Sklyarov, I. Skliarova, and 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.

[8] G. Ferizis and H. ElGindy, “Mapping recursive functions to reconfigurable hardware”, Proc. 16th Int. Conference on Field Programmable Logic and Applications – FPL’06, Madrid, Spain, 2006, pp. 283-288.

[9] S. Ninos and 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.

[10] S.A. Edwards, “The Challenges of Synthesizing Hardware from C-Like Languages”, IEEE Design & Test of Computers, vol. 23, issue 5, September-October 2006, pp. 375-386.

[11] J.V. Nobble, “Recurses!”, Computing in Science & Engineering, May/June 2003, vol. 5, issue 3, pp. 76-81.

[12] G. Stitt and J. Villarreal, “Recursion flattering”, Proc. 18th ACM Great Lakes symposium on VLSI – GLSVLSI’08, FL, USA, 2008, pp. 131-134.

[13] R. Rugina and M. Rinard, ”Recursion unrolling for divide and conquer programs”, Proc. 13th Int. Workshop on Languages and Compilers for Parallel Computing - LCPC’2000, NY, USA, 2000, pp. 34-48.

[14] I. Skliarova and 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.

[15] V. Sklyarov, “Hierarchical Finite-State Machines and their Use for Digital Control”, IEEE Transactions on VLSI Systems, 1999, vol. 7, no. 2, pp. 222-228.