Processing Tree-like Data Structures in Different Computing Platforms

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

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

Ramiro Oliveira
DETI,
University of Aveiro,
Aveiro, Portugal
ramiro@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 analyzes and compares three different computing platforms for processing tree-like data structures, namely: general purpose computers, embedded processors, and direct mapping of the relevant algorithms to hardware in application-specific circuits. Tree-based recursive data sorting is considered as a case study. The results demonstrate that application-specific hardware is the fastest and processor-based implementation is the slowest. This gives motivation for developing new optimization techniques in the scope of application-specific hardware circuits, which is especially beneficial for FPGA-based design.**

Keywords-Algorithms; Processing; Tree-like data structures; Computing platforms; FPGA

I. INTRODUCTION

Tree-like data structure can be seen as a widely used model for numerous computations, such as data sort [1], priority management [1,2], combinatorial optimization [3], etc. Using and taking advantage of application-specific circuits in general and FPGA-based accelerators in particular have a long tradition in data processing [4] and for solving problems with high computational complexity (*e.g.* [3]). A number of research works are targeted to the potential of advanced hardware architectures. For example, the system [5] solves a sorting problem over multiple hardware shading units achieving parallelization through using SIMD operations on GPU processors. The benefits of FPGAs were 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.* [4]).

The use of tree-like data structures can be explained on the following simple example [9] targeted to data sort. 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. Such a tree can easily be built and traversed either iteratively or recursively. Another example can be taken from combinatorial search algorithms [3,10,11]. Let us consider a search tree described in [11]. The root of the tree corresponds to the initial situation in solving a particular task (such as the Boolean satisfiability problem – the SAT). Edges of the tree lead to child nodes of the tree representing simplified situations. In case of the SAT problem [3] the root corresponds to the initial Boolean formula [3] and the other nodes represent simplifies Boolean formulas. Every pair of child nodes permits to remove one variable from the formula assigning it 0 for one child and 1 for another child.

It is known that processing tree-like data structures can be done in different computing platforms. The main objective of this paper is to compare the most widely used platforms, namely *general-purpose processors*; *embedded microprocessors*; and *application-specific hardware circuits* that make it possible direct mapping of the relevant algorithms to hardware to be provided. Recursive data sorting based on tree-like data structures is considered as a case study.

The remainder of this paper is organized in five sections. Section II describes the basic algorithms and their implementation in software. Section III is dedicated to hardware implementation of the basic algorithms. Section IV briefly characterizes the considered computing platforms. Section V is dedicated to experiments, and comparisons. The conclusion is given in Section VI.

II. THE BASIC ALGORITHM AND IMPLEMENTATION IN SOFTWARE

To process tree-like data structures a variety of techniques can be applied. We would like to compare alternative computing platforms through implementations of recursive algorithms because of their clarity and compactness. Although in software iterative algorithms over binary trees reveal slightly better performance, the implementation of recursive algorithms in hardware often gives the result comparable with iterative algorithms. Since forward and backward propagation steps needed for processing tree-like data structures are exactly the same for each node, a recursive procedure can be applied naturally. There are the following four basic modules that can be used for data sorting and some supplementary operations:

- Module M1 adds a new node to the tree;
- Module M2 outputs the sorted data from the tree;
- Module M3 extracts the smallest data item from the tree. Such operation is needed, in particular, for priority management. Alternatively the largest data item can be extracted;

978-1-4244-9503-0/11/$26.00 ©2011 IEEE

- Module M4 removes unneeded tree nodes that have already been extracted or on an external request. Such operation is also useful for priority management.

The following C/C++ code fragments describe the primary operations of the modules M1-M4 (for the simplicity, exception handling is not shown).

```
// Module M1
tree_node* add_node (tree_node* node, int value)
{       if  (node == 0)
        {       node = new tree_node;
                node->value = value;
                node->c = 1; // setting counter to 1
                node->r = node->l = 0;          }
        else if (value == node->value)
                node->c++; // incrementing counter
        else if (value < node->value)
                node->l = add_node (node->l,value);
  // traversing the left sub-tree
        else
                node->r=add_node (node->r,value);
  // traversing the right sub-tree
        return node;                    }

// Module M2
void treesort (treenode *node)
{     if (node!=0)          // if the node exists
      {    treesort (node->l);  // sort left sub-tree
  // display value after any hierarchical return
           treesort (node->r); // now sort right sub-tree
      }
}

// Module M3
void extract_smallest_value (tree_node* node)
{     if (node != 0)
      {    while (node->l != 0)
             node = node->l;
           // send node->value
      }
}

// Module M4
void extract_from_tree (tree_node*& node, int value)
{       tree_node *temp_node;
        if (node != 0)          // verifying if node exists
           if (value > node->value)   // traversing the right sub-tree
              extract_from_tree (node->r,value);
           else if (value < node->value)  // traversing the left sub-tree
              extract_from_tree (node->l,value);
           else
           {    if ( (node->l == 0) && (node->r == 0) )
                // in this case the node has to be deleted
                {       delete node;
                        node = 0;               }
                else if (node->r != 0)
                {       // changing pointers for the right node
                        temp_node = node->r;
                        if ((node->l) != 0)
                            build_subtree (temp_node,node->l,
                                           node->l->value);
                        node->r = temp_node->r;
                        node->l = temp_node->l;
                        node->value = temp_node->value;
                        node->c = temp_node->c;
                        delete temp_node;               }
                else
                {       // changing pointers for the left node
                        temp_node = node->l;
                        node->r = temp_node->r;
                        node->l = temp_node->l;
                        node->value = temp_node->value;
                        node->c = temp_node->c;
                        delete temp_node;               }
           }
}
```

In this code tree_node is considered to be the following structure:

```
struct tree_node
{       int value; // node value
        int c;    // counter for repeated values
        struct tree_node* l; // pointer to the left sub-tree
        struct tree_node* r; // pointer to the right sub-tree
        // other fields if required
};
```

The build_subtree function has the following code:

```
tree_node* build_subtree (tree_node* node
                    tree_node* subnode, int value)
{   if (node == 0) node = subnode;
    else if (value < node->value)
      node->l = build_subtree (node->l, subnode, value);
    else
      node->r = build_subtree (node->r, subnode, value);
    return node;                                    }
```

The modules M1-M3 implement algorithms [9]. The module M4 is described on the basis of the algorithms [1,9]. All the modules have been verified in software.

III. Implementation in Hardware

The following two types of circuits have been designed, tested, and compared:

- Application-specific circuits that directly implement the given algorithms in hardware;
- Circuits which use an embedded Power PC processor executing programs for the given algorithm.

A. Application-specific Circuits

Recursion is applied in the same way as in C/C++ functions shown in section II. Fig. 1 presents a simple example. A tree built for a given sequence of input data items (19,17,11, 28,25,18,34,31) is depicted in Fig. 1(a).

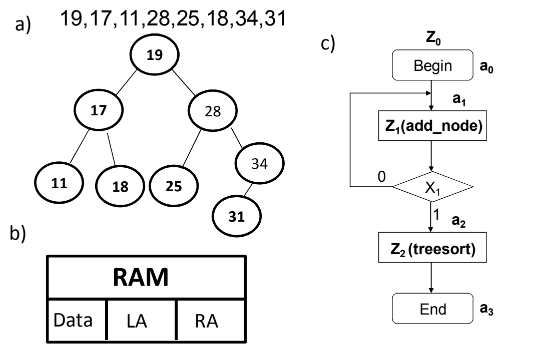

Figure 1. Binary tree for data sort (a); memory contents (b); top-level sorting algorithm (c)

We will assume that input data are stored in RAM along with the addresses of the left (LA) and right (RA) sub-trees (see Fig. 1(b)). Basic top-level algorithm for sorting is shown in Fig. 1(c) (the labels $a_0,\ldots,a_3$ will be discussed later). The module z_1(add_node) corresponds to C/C++ function *add_node* from the previous section. This module sequentially adds input data items to the tree while x_1=0. As soon as x_1=1, the module z_2(treesort) outputs the sorted data from the tree (z_2(treesort) corresponds to C/C++ function *treesort* from the previous section). The executed operations are shown in the functions *add_node* and *treesort* (see section II).

It is known that C/C++ functions from section II can be implemented more efficiently in hardware through the use of dual-port memories and algorithmic modifications. All necessary details can be found in [12]. The improvements permit to speed up the execution of the modules M1-M4.

The designed application-specific circuits are based on a hierarchical finite state machine (HFSM) with a simple datapath. HFSM can be built from C/C++ functions (such as *add_node* and *treesort*). The datapath is the same as in [13]. Fig. 2 demonstrates how the function *add_node* can be converted to specification in form of flow-chart that is needed for synthesis of HFSM. Other functions (*extract_smallest_value*, *extract_from_tree*, *build_subtree*) are implemented similarly. The details can be found in [9].

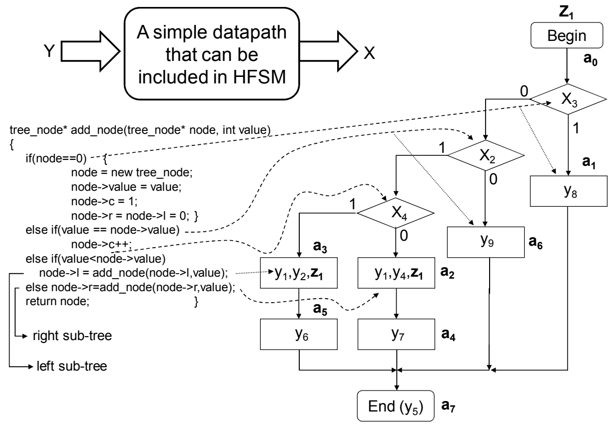

Figure 2. Convertion of the C/C++ function *add_node* to a flow-chart that can be further used for synthesis of an HFSM

Symbols x_2, x_3, x_4 and $y_1,\ldots,y_9$ in Fig. 2 represent accordingly logical conditions and operations in the relevant C/C++ functions (the correspondence is shown by dashed arrow lines). The HFSM (see Fig. 3) analyzes the logical conditions x_2, x_3, x_4 and generates signals $y_1,\ldots,y_9$ in accordance with the flow-chart.

It is known [14] that flow-charts (such as that is shown in Fig. 2) can be converted to HFSM through applying the following sequence of steps [15]:

1) Marking the given flow-chart with labels that will be further considered as the HFSM states. For example the labels a_0-a_3 in Fig. 1 and a_0-a_7 in Fig. 2 are HFSM states. Transitions between the states are described in point 2.
2) Customizing the proposed HDL templates for an HFSM combinational circuit (CC) that can be also combined with the relevant datapath (see Fig. 3). All the details for templates are given in [15], where it is also explained how stack memories shown in Fig. 3 are used.
3) Synthesis of HFSM circuits from the customized templates with the aid of commercially available computer-aided design tools, such as the ISE of Xilinx.

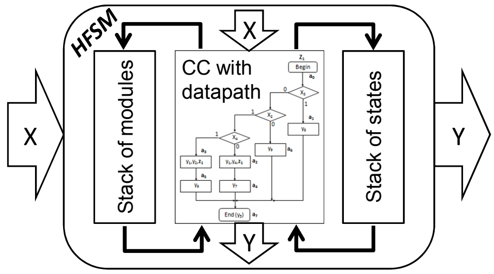

Figure 3. Implementation of the algorithm by an HFSM

B. Using Embedded Power PC Processor

Programs for Power PC processor are developed using embedded development kit (EDK) from Xilinx as it is shown in Fig. 4. Input needed for EDK is very similar to C/C++ functions described in section II. EDK outputs low-level program that can be executed in Power PC.

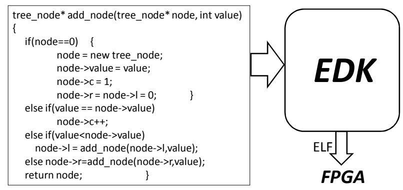

Figure 4. Implementation of the C/C++ functions in Power PC processor

IV. COMPUTING PLATFORMS

Three different computing platforms described below have been analyzed. A random-number generator produces items of data that have to be sorted and the results on different computing platforms are compared.

A. General Purpose Computers

The considered in section II C/C++ functions have been tested on HP EliteBook 2730p (Intel Core 2 Duo CPU, 1.87 GHz) computer. Statements that allow the execution time to be measured were inserted just before and immediately after the execution of the sorting procedure that is composed of the functions *add_node* (this function is sequentially executed while randomly generated data items are available – see Fig. 1(c)) and *treesort*.

B. Application-specific Hardware Circuits

Application-specific hardware circuits were developed on the basis of HFSM using the technique considered in section III.A. Traversing tree-like data structures 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. Data in any memory cell are coded as it is shown in Fig. 1(b). The absence of a node is indicated by 0 because zero address is used just for the root node and can easily be recognized. PM is based on HFSM and it builds the tree from incoming data through creating pointers between the data items and outputs the sorted sequence from the tree.

C. PowerPC

The C/C++ functions considered in section II have been converted to the set of instructions for PowerPC PPC405 processor embedded to FPGA Virtex-4 FX12 available on prototyping board FX12 of Nu Horizons. Synthesis and implementations were done using Xilinx ISE and Xilinx EDK.

V. Experiments and Results

Initial data are generated randomly in the PC computer and then are used in computations within the platforms described in subsections IV.A, IV.B, IV.C (see Fig. 5). For software implementations C/C++ programs take data items directly from a random generator in the PC computer and produce the results of sorting on PC monitor screen.

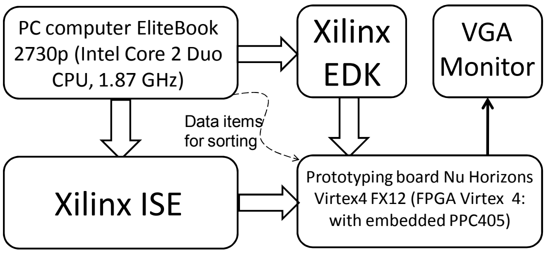

Figure 5. Experiments in different computing platforms

Xilinx tools ISE and EDK are used to design FPGA circuits as it is shown in Fig. 5. In hardware implementations the FPGA receives and stores data through RS-232 port available on the prototyping board FX12. The time of data transfer is not taken into account. The results of sorting are shown on a VGA display (see Fig. 5) directly connected to the prototyping board. An example is shown in Fig. 6.

Tables I, II, III present the results for different computing platforms (general-purpose computer – Table I, embedded to FPGA PowerPC – Table II, application-specific FPGA circuits based on HFSM model – Table III). The work frequency for FPGA was set to 200 MHz. The number of data items in Tables I, II and Table III is different because the implementations of the improved algorithms in FPGA require embedded memory blocks and the selected FPGA does not have sufficient number of such blocks. Using external memory permits the number of data items to be easily increased. Besides, we can see the following tendency in Table III: the more the number of data items, the better time per data item is achieved.

If we compare the results we can conclude that application-specific HFSM-based FPGA circuits are undoubtedly the fastest. PowerPC-based implementations are the slowest. These results give well-founded motivation for exploring and optimization of application-specific hardware circuits targeted to processing tree-like data structures. Such hardware circuits are especially useful and advantageous for FPGA-based applications because FPGA can easily be customized for the required types of data and data sizes. Note that similar results were received for implementation of iterative algorithms over tree-like data structures.

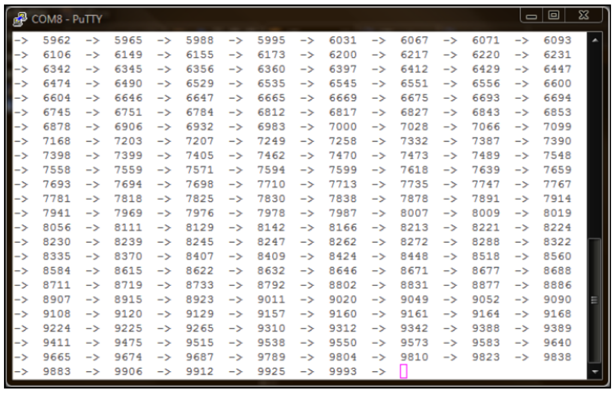

Figure 6. Example of results generated by hardware circuits

TABLE I. General-Purpose computer

Number of data items	5000	10000	20000	30000	40000	50000
Time(ms)	1.84	2.90	5.60	7.90	10.4	12.0
Time per data item (μs)	0.368	0.29	0.28	0.263	0.259	0.24

TABLE II. PowerPC

Number of data items	5000	10000	20000	30000	40000	50000
Time (s)	0.17	0.27	0.56	0.83	1.09	1.25
Time per data item (μs)	34	27	28	27.7	27.3	25

TABLE III. Application-specific, HFSM-based FPGA circuits for the best improved algorithm

Number of data items	1210	1236	1249	1265	1320	1518
Time(μs)	16.3	16.6	16.8	17.0	17.6	19.7
Time per data item (ns)	13.5	13.4	13.4	13.3	13.3	12.98

Resource consumption for application-specific hardware circuits is quite reasonable. For example, the circuit for Table III is built on 1556 FPGA slices and the used FPGA has totally 5472 slices.

There are a number of ways permitting additional improvements of application-specific circuits to be provided. The most important of them are briefly characterized below.

The paper [16] describes the hardware (FPGA-based) implementation and optimization of *parallel* recursive algorithms that sort data using binary trees. Recursive calls are supported using the HFSM model. 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 demonstrates that for N=4 parallel algorithms permit to improve performance in approximately 3 times comparing with the considered in this paper sequential algorithm. Preliminary results for N>4 demonstrate potentiality for additional speed-up.

The paper [17] suggests multilevel models for data processing and shows advantages of such models on examples of data sorting. An integration of three different techniques is discussed, namely graph walk [3], tree-like structures, and sorting networks. The relevant implementations were done on the basis of HFSMs and verified in commercially available FPGAs. Experiments and comparisons demonstrate that the results enable the performance of processing for different types of data to be increased compared to known implementations over tree-like structures. For example, when trees (such as described above) and sorting networks are combined, an additional acceleration of sorting is provided in an average 1.6/3.1/4.7 times for different values of k: k=2/k=3/k=4 respectively, where 2^k is the number of items processed by sorting networks. 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 required hardware resources for application-specific circuits can also be decreased. One potential way is to consider the most appropriate HFSM model. For example, paper [18] demonstrates that the use of HFSM with implicit modules instead of the HFSM with explicit modules permits the needed hardware resources to be reduced. Additional optimization can be achieved applying the advanced technique described in [15].

VI. Conclusion

Experiments with three widely used computing platforms (general-purpose processor, embedded PowerPC processor, and direct mapping of the relevant algorithms to hardware) for processing tree-like data structures clearly demonstrate advantages of application-specific circuits and give well-founded motivation for the development of new optimization methods in this area, which is especially beneficial for FPGA-based design. Advanced technique, which can be applied to both processing tree-like structures (*e.g.* parallel processing, combining different methods) and HFSM synthesis and optimization (such as using implicit modules) permits even better results for application-specific circuits to be received.

Acknowledgment

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, C. Stain, *Introduction to Algorithms*, 2nd edition, MIT Press, 2002.

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

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

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

[5] N.K. Govindaraju, J. Gray, R. Kumar, D. Manocha, “GPUTeraSort: High performance graphics co-processor sorting for large database management”, Proc. of 2006 ACM SIGMOD Int'l Conference on Management of Data, Chicago, IL, USA, pp. 325-336, 2006.

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

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

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

[9] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[10] J. Gu, P.W. Purdom, J. Franco, B.W. Wah, “Algorithms for the Satisfiability (SAT) Problem: A Survey”, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 35, pp. 19-151, 1997.

[11] A. Zakrevskij, Y. Pottosin, L. Cheremisinova, *Combinatorial Algorithms of Discrete Mathematics*, TUT Press, 2008.

[12] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson, "Hardware Implementation of Recursive Algorithms", Proc. of the 2010 IEEE International 53rd Midwest Symposium on Circuits and Systems - MWSCAS 2010, Seattle, USA, August 2010, pp. 225-228.

[13] 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.

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

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

[16] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson, "Parallel FPGA-based Implementation of Recursive Sorting Algorithms", Proc. of ReConFig’2010, Cancun, Mexico, 13-15 December, 2010.

[17] V. Sklyarov, I. Skliarova, D. Mihhailov, A. Sudnitson, "Multilevel Models for Data Processing", Proc. of GCC’2011, Dubai, February 2011.

[18] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson, "Application-specific hardware accelerator for implementing recursive sorting algorithms", Proc. of FPT’2010, Beijng, China, December 2010.