Software/Hardware Solutions for Information Processing in All Programmable Systems-on-Chip

Valery Sklyarov*. Iouliia Skliarova*
João Silva**

**Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, 3810-193 Aveiro Portugal (Tel: 351-401-539; e-mail: skl@ua.pt, iouliia@ua.pt)*
*** Institute of Telecommunications, University of Aveiro, 3810-193 Aveiro Portugal (e-mail: jpss@ua.pt)*

Abstract: Algorithms in computer engineering and applied informatics often require extraction of data with some desired properties from large sets. Such tasks appear within different clustering algorithms in the scope of data mining, in classification of objects in accordance with given criteria, in knowledge acquisition obtained from controlled environments, in statistical analysis and in other areas. Many of these tasks involve widely used data processing techniques that are sorting and searching and for numerous practical applications, especially in real-time and high-performance systems, speeding-up is important. The paper suggests effective portable solutions that enable fast parallel information processing to be implemented in all-programmable systems-on-chip that combine multi-core computations with programmable logic interacting through multiple high-performance interfaces. Acceleration is achieved with parallel networks for data sorting created in programmable logic and enabling software running in multi-core processing units to be speeded-up, which is demonstrated in numerous practical examples fully implemented and tested in commercial microchips.

Keywords: parallel processing, microsystems, information retrieval, data processing, performance evaluation, multiprocessing systems.

1. INTRODUCTION

Data extraction and ordering are required in many algorithms of control engineering and applied informatics. Some common problems are listed below (see also Fig. 1):

1) Extracting the maximum/minimum (sorted) subsets from the given set;
2) Extracting subsets with such values that fall within an interval bounded by the given maximum and minimum;
3) Encountering the most repeated value or a set of the most repeated values;
4) Computing medians;
5) Solving problems indicated in points 1)-4) for tables (for values in rows/columns of the tables).

All these problems can efficiently be solved applying data sorting that is one of the most common types of computations (Knuth, 2011) required in different information processing systems. Let us describe practical applications where solutions of the problems listed above are needed. Clustering is a data mining activity that permits a given set of objects with similar properties to be grouped (Kovacs et al., 2007). Hierarchical clustering methods represent a major technique allowing the desired set to be built through searching common attributes and combining objects with such attributes (Serban et al., 2008). For instance, clinical investigators, health professionals and managers are often interested in clustering patients into clinically meaningful groups according to their expected length of stay (Garg et al., 2011). Similar problems arise in statistical data manipulation (Salter-Townshend et al., 2012; Sklyarov et al., 2014a), in classification (Chrysos et al., 2013; Abdelhamid et al., 2015) and in many other areas.

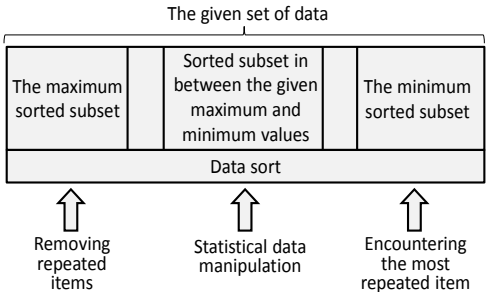

Fig. 1. Common problems that frequently need to be solved in algorithms of control engineering and applied informatics.

In (Baker et al., 2006) one common task is explained on an analogy of a shopping activity. A basket is a set of items purchased at one time. A frequent item is an item that often occurs in a database. A frequent set of items often occur together in the same basket. A researcher can request a particular support value and find the items which occur together in a basket either a maximum or a minimum number of times within the database. Similar problems appear to determine frequent inquiries at the Internet, customer

transactions, credit card purchases, etc., producing very large volumes of data in the span of a day. Sorting is involved in many known algorithms from this area (*e.g.* Sun, 2011a; Sun et al., 2011b; Wu et al., 2014; Firdhous, 2010). We believe that fast solvers for the problems listed above would be very helpful in service oriented, knowledge-based systems that use advanced search, data analytics and prediction tools such as in (Borangiu et al., 2014). Data need also to be processed in many other environmental, medical, and biological applications. Let us consider some examples. Applying the technique (Zmaranda et al., 2013) in real-time systems requires knowledge acquisition from a controlled environment (*e.g.* plant). For example, signals from sensors may be filtered and analyzed to prevent error conditions. To provide more exact and reliable conclusions, it is necessary to order and examine a combination of different values. Similar tasks appear in monitoring thermal radiation from volcanic products (Field et al., 2012), filtering and integration of information from a variety of different sources in medical applications (Zhang et al., 2014). Since many such systems are real-time, performance is important and hardware accelerators may provide significant assistance for software.

This paper suggests effective portable software/hardware designs executing different types of sorting and merging methods in recently appeared on the market Zynq-7000 devices from Xilinx (Xilinx, 2014a). These devices are based on Xilinx all programmable system-on-chip (APSoC) architecture, which combines the dual-core ARM® Cortex™–A9 central processing unit with Xilinx programmable logic (PL) appended with on-chip memories (OCM), high-performance (HP) interfaces, a rich set of input/output peripherals, and a number of embedded to the PL components, such as digital signal processing (DSP) slices. APSoC devices permit complete solutions to be realized on a single microchip running software that may be enhanced with easily customizable hardware. Various advantages of APSoC platform are summarized in (Santarini, 2014; Santarini, 2015). Interactions between the ARM-based processing system (PS) and PL are supported by nine on-chip Advanced eXtensible Interfaces (AXI): four 32-bit general-purpose (GP) ports; four 32/64-bit HP ports and one 64-bit accelerator coherency port (ACP) (Xilinx, 2014a). Different interfaces for hardware/software communications in Zynq-7000 devices were compared and analyzed in detail in (Silva et al., 2015). We will use these results for selecting the most appropriate hardware/software system architecture.

The remainder of the paper is organized in five sections. Section 2 discusses the background and potential applications of the proposed technique in the scope of applied informatics and control engineering. Section 3 introduces the adopted computation methods. Section 4 is dedicated to data processing in multi-core APSoC-based software/hardware systems. Section 5 describes implementation details and the results of experiments and comparisons. The conclusion is given in Section 6.

2. BACKGROUND AND POTENTIAL APPLICATIONS

Combining capabilities of software and hardware permits many characteristics of developed applications to be improved. The earliest work in this direction was done at the University of California at Los Angeles (Estrin, 1960). The idea was to create *Fixed + Variable* structure computer and to augment a standard processor by an array of reconfigurable logic assuming that this logic can be utilized to solve some processor tasks faster and more efficiently. Such combination of flexibility of software and speed of hardware was considered as a new way to evolve higher performance computing from any general-purpose computer. The level of technology in 1959-1960 was not sufficient for this method to be put in practice. Today Zynq-7000 architecture enables the ideas from (Estrin, 1960) to be realized in a wide scope of engineering designs (see examples in Santarini, 2014; Santarini, 2015). The basic reasons for choosing a software/hardware platform based on the Zynq-7000 APSoC are listed below:

1) ARM has become the standard embedded processing architecture for about anything that is not a PC (Santarini, 2014).

2) Newer more advanced microchips, such as Ultrascale Multiprocessing SoC, combine a quad-core 64-bit ARM Cortex-A53 application processor, a 32-bit ARM Cortex-R5 real time processor, and an ARM Mali-400 MP graphics processor together with 16 nm logic on a single chip (Santarini, 2015). This permits advanced portable systems on a single microchip possessing computational resources comparable with that of a PC and with significantly lower power consumption to be developed. Easily scalable designs (such as that described in the paper) can be implemented in currently available and future microchips that are faster, have more advanced resources, and consume less power.

3) Multicore on-chip architectures provide a way around the implementing limits of Moore's low (Bertels et al., 2010; Santarini, 2015). Today the Zynq SoC is at the heart of many of the world's newest and most innovative automotive, medical and security vision products (Santarini, 2014). It is very appropriate for numerous designs described, for example, in (Bertels et al., 2010; Kestur et al., 2010; Krueger et al., 2011). Thus, the proposed solutions may be adopted and used in a wide range of practical applications.

Let us look at Fig. 1 and the problems 1)-5) listed in section 1. These problems can be solved applying a technique described below:

1) We consider network-based data sorting as a core method for different problems. Much like (Mueller et al., 2012) to speed up the sorting, we suggest augmenting software running in the PS with HP hardware accelerators. However, the proposed methods differ from (Mueller et al., 2012) and they permit better solutions to be found applying such a network, which runs in parallel with data transfers, occupies significantly smaller hardware resources, and enables wider parallelism to be achieved in the dual-core PS and PL. Besides, the proposed solutions are scalable and may be used in a PS with more than two cores.

2) The problems 1)-5) are mainly based on extraction of data with certain characteristics, which is done using sorted sets. This is significantly easier and faster than solving similar tasks for unsorted subsets.

Since data sorting is the core method, we will analyze it with more detail. Performance is important for many information processing systems. The analysis presented in (Sklyarov et al., 2014a) enables us to conclude the following:

- The known even-odd merge and bitonic merge networks (Baddar et al., 2011) are the fastest and enable the best throughput to be achieved. However, they are very resource consuming and can only be built in the APSoC PL for sorting very small data sets. For example, in (Mueller, 2010) any set is composed of only 8 items.

- Pipelined solutions permit even faster circuits than in the point above to be designed. Usually pipelining can be based on flip-flops from APSoC PL slices used for the network, and resource consumption is almost the same as in the point above. Once again, in practice, only very small data sets can be processed in APSoC PL. Besides, the bottleneck is in fact not in the hardware sorter but in communications between software and hardware (Silva et al., 2015).

- To use even-odd merge and bitonic merge circuits for larger data sets, the following two methods are most commonly applied: a) large data sets are sorted in host computers/processors based on sorted blocks produced by hardware accelerators (see, for example, Mueller et al., 2012; Chamberlain et al., 2009); b) sorting networks for large sets are segmented such that any segment can be processed easily and the results are handled sequentially to form the sorted set like for example, in (Zulada et al., 2012; Gapannini et al., 2012). Both methods involve intensive communications between the PS and PL. The necessity in frequent data exchanges reduces potential benefits from the fastest burst mode, limiting burst sizes.

- The existing even-odd merge and bitonic merge circuits are not very regular (compared, for example, to the even-odd transition network (Kipfer et al., 2005)). The routing overhead may be considerable in the PL, increasing the occupied resources.

- It is shown in (Sklyarov et al., 2014a) that very regular even-odd transition networks with two sequentially reusable vertical lines of comparators are more practical because they operate with higher clock frequency, provide sufficient throughput, and enable a significantly larger number of items to be processed in the PL.

- Experiments that were done give additional motivation to apply the methods (Sklyarov et al., 2014a) which finally have been chosen as a base for sorting networks in the PL of APSoC. Two novel solutions compared to (Sklyarov et al., 2014a) are proposed: a) the number of combinational levels in the sorting network is reduced from 2 to just 1, permitting clock frequency in the PL to be increased; b) data processing is executed during data transmission, which enables throughput to be increased.

Let us discuss now effectiveness and applicability of the technique outlined in points 1)-2) of this section in the scope of applied informatics and control engineering.

One common problem is clustering objects in accordance with their attributes. Different methods have been proposed for solving this problem and many of them may recur to sorting and searching as frequently used operations (*e.g.* Sun, 2011a; Wu et al., 2014; Firdhous, 2010). For example, in CPES (Clustering with Prototype Entity Selection) method (Kovacs et al., 2007), a fitness function $f(x_i)$ is proposed to decide if given objects can be clustered and it is computed on the basis of the Euclidean distance $d(x_i,x_j)$:

$$d(x_i, x_j) = \sqrt{|x_{i1} - x_{j1}|^2 + ... + |x_{ip} - x_{jp}|^2}$$

where p is the number of attributes for objects x_i and x_j, $x_{i1},\ldots,x_{ip}$, $x_{j1},\ldots,x_{jp}$ are attributes for objects x_i and x_j. Generally, such two objects x_i and x_j are chosen for clustering for which the value of the fitness function f is higher. Sorting the Euclidean distances and the relevant fitness functions permits finding solutions with the method referenced above faster. Besides, a number of support functions can be suggested. For example, let attributes be associated with rows of a matrix μ and objects be associated with columns of the matrix μ. Intersection of a row r and a column c is marked with value 1 if an object in column c has an attribute in the row r. We assume that all unmarked positions in the matrix μ are zeros. Discovering and sorting Hamming weights for all the rows allows frequencies of attributes in different objects to be found. This simplifies allocating candidates for merging in clusters. A similar technique is considered in (Baker *et al.,* 2006). Thus, it is important to make sorting built-in much like it is done for such operations that compute Hamming weights of binary vectors, *e.g.* POPCNT (population count) (Intel, 2007) and VCNT (Vector Count Set Bits) (ARM, 2013). Similar proposals were made in (Arnold et al., 2014).

The following problem that requires fast sorting is described in (Sklyarov et al., 2013a). Suppose there are predefined values $\alpha_1,\ldots,\alpha_Q$ and we would like to discover how many values $\alpha_q \in \{\alpha_1,\ldots,\alpha_Q\}$ can be found in a given set. Let us consider a set of data items $I_0,\ldots I_{N-1}$. The result $R(\alpha_q)$ of comparing $\alpha_q \in \{\alpha_1,\ldots,\alpha_Q\}$ with all the items $I_0,\ldots I_{N-1}$ is a binary vector. The Hamming weight of the vector $R(\alpha_q)$ is equal to the number of items with the value α_q. Sorting the results $R(\alpha_1),\ldots,R(\alpha_Q)$ gives the distribution of data items with the values from $\{\alpha_1,\ldots,\alpha_Q\}$ in the set $I_0,\ldots I_{N-1}$. Such a problem appears in pattern recognition, image and signal processing.

The algorithm (Abdelhamid, 2015) discovers rules associated with a set of classes and it has been tested on a real world application data set related to website phishing. The experimental results show the effectiveness of this algorithm in which the classifier sorts classes within each rule based on their frequency. Thus, sorting is also needed.

In (Mueller, 2010) small even-odd merge and bitonic sorting networks were used to implement a median operator over a count-based sliding window. Such an operator is commonly needed to eliminate noise in sensor readings (Rabiner et al., 1975) and in data analysis (Tukey, 1977) that are tasks often solved in control engineering. These methods (Mueller, 2010) were also applied to wireless sensor networks.

The method proposed in (Batista et al., 2014) evaluates systematically the possible behaviors of a closed-loop system by analyzing its time response. This permits various techniques to be applied for solving the problems commonly encountered in the networked control systems. The proposed technique (Batista et al., 2014) is based on the Monte Carlo method coupled with a sorting algorithm (Sedgewick, 1978) and a gradient search (Yuan, 2008).

The software/hardware solutions proposed in this paper are faster. They combine a multi-core processor with hardware accelerators running in parallel. Effectiveness of hardware/software solutions is underlined in (Goodman, 2011), addressing the importance of portable computing hardware environments to handle massive data. According to Goodman, data miners and statisticians should collaborate and thus common design techniques are promising. We believe that sorting networks are one of such design techniques, which is also stated in (Mueller, 2010).

3. METHODS

Fig. 2 outlines the basic architecture of the proposed hardware accelerator for data sorting. The core component is an iterative even-odd transition network. As distinct from (Sklyarov et al., 2014a), the same comparator/swapper is reused for even and odd levels and switching between the even and odd levels is done by multiplexers that are not shown in Fig. 2. The maximum number of clock cycles τ_{max} for sorting N data items is equal to τ_{max}=N (Kipfer et al., 2005).

The primary idea is to enable data transfers and sorting in parallel. This is the main distinctive feature as compared to all the methods published and referenced above.

Let L be the number of items that have to be sorted and N be the number of items in one sorting block. The maximum delay from the beginning of source data transfer to the end of the result transfer is $\lceil L/N \rceil \times N+2\times N = (\lceil L/N \rceil+2)\times N$. Indeed, N clock cycles are needed to transfer the initial block of data from memory to the input register, N clock cycles - to transfer the sorted block of data from the output register to memory, and N clock cycles for sorting each block in the iterative network. Comparison/swapping is done for all N data items in parallel and we need N iterations at maximum. Beginning from the second block, sorting is done in parallel with data transfer to the input register. Thus, receiving input items and sorting require N clock cycles. As soon as N items are sorted, they are copied in parallel to the output register. Hence, 2×N clock cycles are needed to prepare the first sorted block in the output register. In subsequent N clock cycles: a) N data items are transferred to the input register; b) N previously transferred data items are sorted in the network; and c) N previously sorted data items are transferred from the output register to the PS (memory). Clearly, three operations a), b), and c) are executed at the same time, i.e. in parallel. Transferring N data items from the memory to the sorter needs N clock cycles. Transferring N data items from the sorter to the memory needs also N clock cycles. Therefore, if sorting can be done even without any delay, then only the data transfer requires L+N clock cycles. Indeed, N clock cycles are needed to get the first block and subsequent blocks can be received in parallel with transferring the results. For a simplification, we ignore additional clock cycles that are needed between burst packages (they are negligible and appear in any method). The difference between the described unrealistic case and the proposed solution in Fig. 2 is approximately N clock cycles that are required for the first sorting. Thus, the proposed solution is indeed very fast.

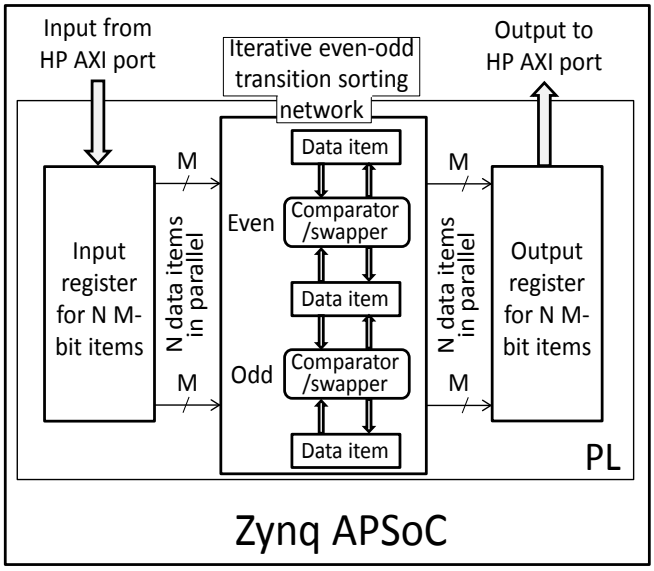

Fig. 2. Basic architecture of the hardware accelerator for data sorting.

The next distinctive feature is an opportunity to process data in the network with the maximum speed of data transfer. This can be done thanks to the minimal delay in comparators/ swappers for the network in Fig. 2. Indeed, all signals propagate through only one level of comparators/swappers at different iterations. The number of levels with comparators/swappers in combinational networks (Knuth, 2011; Baddar et al., 2011) is equal to $\lceil \log_2 N \rceil \times (\lceil \log_2 N \rceil+1)/2$ and, even for small values of N, let us say 128, the delay in (Knuth, 2011; Baddar et al., 2011) is 28 times bigger. Possible pipelining permits this delay to be reduced (Mueller et al., 2012). However, there is another very serious problem with the networks (Baddar et al., 2011). The proposed technique permits significantly more complicated sorters to be implemented with the same hardware resources. Indeed, for N=128 the number of comparators/swappers in Fig. 2 is N-1=127 and the number of comparators/swappers for the best network from (Baddar et al., 2011) is 1,471. This means that the required hardware resources for the networks (Baddar et al., 2011) are more than 10 times larger. Sorting large data sets is done by merging sorted blocks (in which the given large set is decomposed) in software.

Some additional improvements can be done taking into account an opportunity to transfer 64 bits in parallel. Any individual transfer in Fig. 2 is done for M=32 bit data items. AXI HP and AXI ACP interfaces in (Xilinx, 2014a) enable 64 bits to be transferred in one transaction. Thus, two M=32-bit data items can be parked in one AXI word, enabling the number of data transfers to be reduced by a factor of 2. However, in this case the iterative networks from (Sklyarov et al., 2014a) can directly be used, which enable sorting to be done in N/2 clock cycles at maximum (instead of N clock cycles in Fig. 2). Each method has advantages and disadvantages. For the method (Sklyarov et al., 2014a), the maximum number of data items (that can be processed in the PL) is reduced compared to Fig. 2, and the delay in the iterative network is increased (because signals propagate through two levels of comparators instead of one level in Fig. 2). This does not allow the maximum clock frequency for the PL to be used. Hence, the speed of transfers is decreased for 2×L items (L for reading and L for writing). However, the number of data transfers (when we pack 2 items in one AXI word) is also decreased and this is an advantage. In section 5 we will compare these two methods. Transferring 2×32 bit items can also be practical for the architecture in Fig. 2 using FIFOs on inputs and outputs. Although there is no speed-up in data processing in the PL, communication overheads are reduced by a factor of 2 and a shared memory (DDR/OCM) can be used by the PS for solving other problems in parallel.

Let us discuss now how the hardware accelerator shown in Fig. 2 can be used for solving problems 1)-5) listed in section 1. Extracting the maximum/minimum (sorted) subsets (see point 1) is done trivially, copying the required number of items either from the beginning or from the end of the sorted set. Since we need just to read data from memory, this process is very fast in the PS software. Alternatively, the maximum and the minimum sorted subsets may be accumulated in the PL and this can be done even faster. Extracting subsets with such values that fall between the given maximum and minimum can also be done in software applying, for example, a binary search tree (Cormen et al., 2009) to the sorted set of data.

The most repeated value can easily be encountered in a parallel hardware circuit that receives the sorted set on inputs and finds the most repeated item in ξ-1 clock cycles, where ξ is the number of repetitions of the most frequent item (Sklyarov et al., 2014b). Additional details will be given in subsection 5.1.

Computing medians on the basis of the results of sorting is described in (Mueller, 2010). Similar problems 1)-5) for tables can easily be solved taking input data from the selected rows/columns. Besides, the proposed methods may widely be used to solve other problems of control engineering and applied informatics discussed in the previous section.

4. PARALLEL DATA PROCESSING IN AN APSOC-BASED SOFTWARE/HARDWARE SYSTEM

We will analyze below the following four designs for solving the sorting problem:

1) A single core implementation where software in the PS and hardware in the PL operate sequentially. Analysis of such design permits communication overheads to be evaluated easier.

2) A single core implementation where software in the PS and hardware in the PL operate in parallel. The PS and PL frequently share the same memory, which may lead to performance degradation. The considered design permits to evaluate potential advantages/drawbacks.

3) A dual-core implementation where software in the PS and hardware in the PL operate sequentially. This permits comparison of dual-core and single-core solutions taking into account communication overheads between software and hardware.

4) A multi-core implementation where software in the dual-core PS and hardware in the PL operate in parallel allows the highest level of parallelism to be examined and evaluated.

Designs 3) and 4) permit merge operations in the PS to be parallelized in different cores. Designs 2) and 4) permit merge operations to be executed in the PS concurrently with sorting blocks in the PL. Access to memories can be done in lite and burst modes. The latter is faster (Silva et al., 2015), especially for transferring large data sets and it will be used in all the proposed designs. Note that higher parallelism requires more sophisticated interactions between the processing units that execute parallel operations. Besides, the used memories often have to be shared between the processing units. Potentialities for APSoC standalone applications are limited and applications running under operating systems (such as Linux) involve additional delays caused by the relevant programs of the operating systems. Furthermore, the programs allocate memory spaces and the size of available memory for data sorters is reduced. Consequently, more constraints are introduced. So, the results of the designs listed above need to be carefully evaluated and compared and they cannot be predicted in advance.

4.1 Single Core Implementation

Fig. 3 shows the proposed hardware/software architecture, which includes hardware in the PL synthesized from specifications in VHDL in Xilinx Vivado 2014.4 design suite and software written in C language and developed in Xilinx Software Development Kit SDK 2014.4.

The PL reads blocks of data from the chosen memory, sorts them by the iterative network, and copies the sorted blocks to the same location in the memory. Note that on-chip cache may be extensively used by other software programs running, for example, under Linux operating system. The available space for application-specific software and hardware is almost always unknown. However, as soon as the cache is filled up, an on-chip controller selects another available memory. We found that the use of cache memory is more efficient for standalone applications rather than for Linux applications.

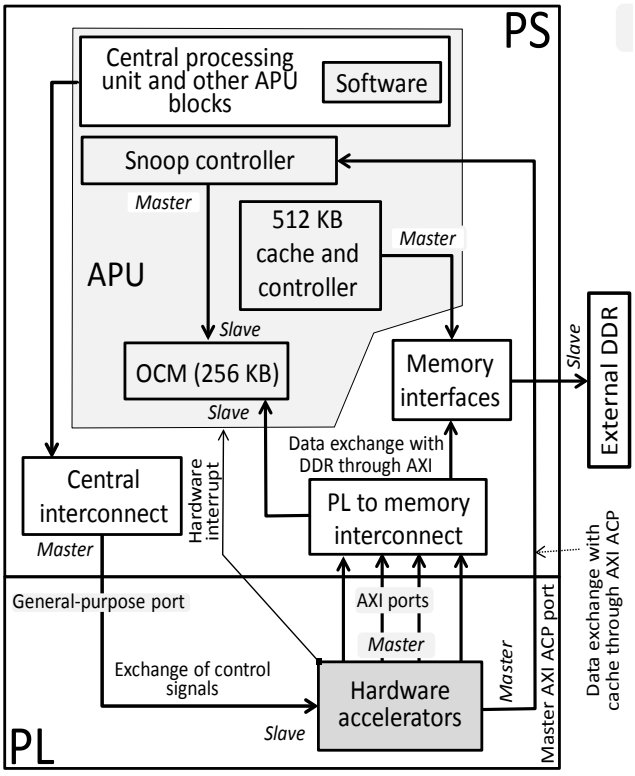

Fig. 3. Hardware/software architecture for a single core implementation.

As soon as all sorted blocks are ready and copied to memories, the PL forms an interrupt to the PS, indicating that further processing (*i.e.* merging) can be started. The PS reads the sorted subsets from memory and merges them in software, producing the final sorted set.

4.2 Single Core Implementation with Parallel Operations

Fig. 4 shows the proposed hardware/software architecture. $\lceil L/N \rceil$ blocks with up to N M-bit data items are copied from the chosen memory to the PL, sorted, and the sorted blocks are transferred back to the memory. As soon as the first two blocks are sorted and transferred, the PL generates an interrupt, indicating that the first two blocks can be merged in software of the PS. Further merging in software and sorting the remaining blocks in hardware are done in parallel. The number of currently sorted blocks is periodically updated through a GP port. As soon as the PS finishes merging, it checks the number of newly available blocks from the PL through a GP port. If a new pair of blocks is available, a new merge operation is started, otherwise either a merge of the previously merged blocks is initiated (if such blocks are ready) or software is suspended until blocks for merging from the PL become available. The latter situation (although supported) actually never occurs because hardware is faster than software even taking into account the communication overheads. Thus, the PS and the PL run in parallel until the final result of sorting is produced. Memories may be shared but such sharing is minimized through potential invocation of different memories (DDR, OCM and cache). Sorting of blocks in the PL is finished much earlier than merging the sorted blocks in the PS.

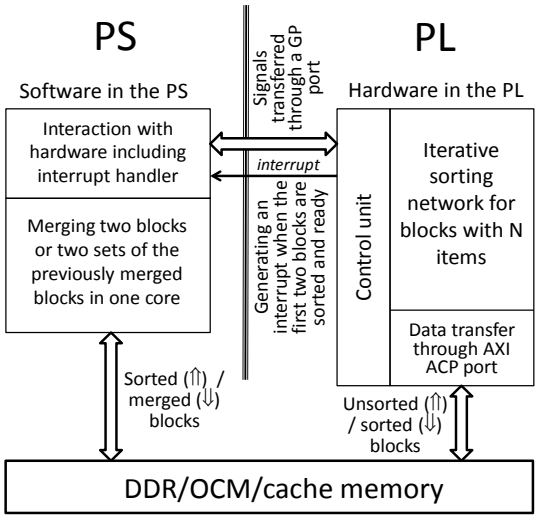

Fig. 4. Hardware/software architecture for single core implementation with parallel operations in the PS and PL.

4.3 Dual-core Implementation

We consider here a dual-core project running under Linux. A similar project may be used for more than two cores as soon as they become available in APSoCs (Santarini, 2015). Hardware for the project is the same as in sections 4.1 and 4.2. There are 4 threads in software executed in the processing cores of the PS such that two processing cores may be active at the same time (*i.e.* in parallel). The first thread is responsible for transferring unsorted subsets from the PS to the PL and sorted subsets from the PL to the PS. Finally, $\lceil L/N \rceil$ sorted subsets will be ready for the PS and they are divided into two halves. The second and the third threads activate the functions (*halfMerger*) that merge the first and the second halves of the sorted subsets, creating two large blocks of data that are further merged in the function *finalMerger* activated in the last (fourth) thread. Two functions *halfMerger* may run in different cores in parallel. Multiple threads are managed by the operating system. In this type of implementation, hardware and software operate sequentially, i.e. at the beginning software is suspended, waiting until all the blocks have been sorted in hardware.

4.4 Dual-core Implementation with Parallel Operations

Fig. 5 shows the proposed hardware/software architecture. $\lceil L/N \rceil$ blocks with up to N M-bit data items are copied from the chosen memory to the PL, sorted, and the sorted subsets are transferred back to the memory. As soon as the first two blocks are sorted and transferred, the PL generates an interrupt, indicating that the first two blocks can be merged in the software of the PS running in one core. At the beginning, software running in the second core checks availability of the sorted blocks through a GP port. As soon as such blocks are available, merging is started in parallel with merging in the first core. Subsequent operations are similar to those in section 4.2, i.e. as soon as any core finishes the merging, it checks the number of newly available blocks from the PL through a GP port. If a new pair of blocks is available, a new merge operation is started, otherwise either a merge of the

previously merged blocks is initiated or software is suspended until blocks for merging become available. Thus, software in two cores of the PS and hardware in the PL may run in parallel until the final result of sorting is produced. A similar project can be implemented for more than two cores.

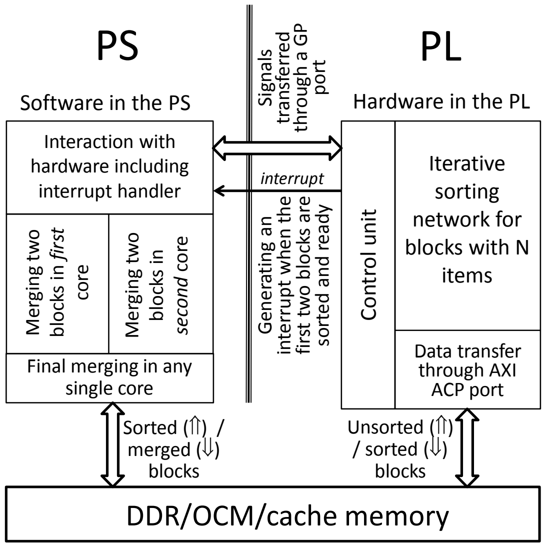

Fig. 5. Hardware/software architecture for multi-core implementation with parallel operations in the PS and PL.

5. EXPERIMENTS AND COMPARISONS

5.1 Implementation Details and Experimental Setup

All the designs described in points 4.1-4.4 have been implemented and tested in two prototyping systems: ZyBo with the Xilinx APSoC xc7z010-1clg400C (Digilent, 2014) and ZedBoard with the Xilinx APSoC xc7z020-1clg484c (Avnet, 2014). Two different APSoCs were chosen not for comparison between them. Our aim was just to show that the results are applicable to APSoCs with different complexity, including the cheapest and the less advanced microchip xc7z010-1clg400C. Interactions between hardware and software were done through Xilinx IP cores (Xilinx, 2014b). The number of data items in the initial (unsorted) set varies from N (*i.e.* from the size of one block) to L=33,554,432 (*i.e.* up to more than 33 million of 32-bit data items). Fig. 6 shows the organization of the experiments.

We have used a multi-level computing system. Initial unsorted data are either generated randomly in the software of the PS with the aid of C language rand function or prepared in the host PC. In the last case, data may be randomly generated by the rand or other functions or copied from benchmarks. Sorting is done completely in APSoC using architectures from section 4. The results are verified in software.

Standalone software applications have been created and uploaded to the PS memory from SDK. Interactions are done through the SDK console window. An example of interactions for a project from section 4.1 is shown in Fig. 7 for ZedBoard and N=256.

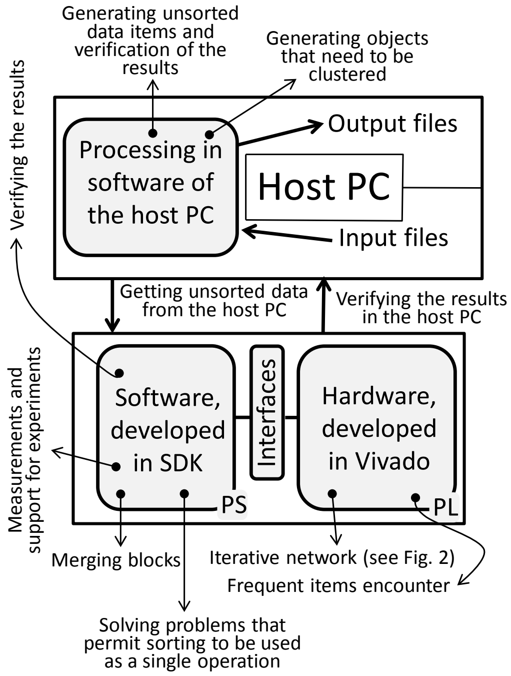

Fig. 6. The experimental setup.

Sorting Blocks	Values	Speed Up
1	256	10.10
2	512	6.46
256	65536	2.14
4096	1048576	1.76
16384	4194304	1.67
131072	33554432	1.56

Fig. 7. The results of experiments and comparisons (speed-up is measured compared to software only data sorter running in the PS and using C function qsort).

The measurements include all the involved communication overheads. The number of blocks varies from 1 to 131,072, N=256, and, thus, the number of 32-bit data items ranges from 256 to more than 33 million. For all the experiments, AXI ACP port was used for transferring blocks between the PL and memories.

The developed software and hardware may also solve tasks of higher hierarchical levels. As examples, we considered creating objects in software for further clustering and frequent items encountering in hardware (see Fig. 6). Attributes of any individual object are generated randomly in software within a given range. Objects and attributes are associated with rows and columns of the matrix μ, which is built in accordance with the rules given in section 2. Clearly, the Hamming weight of any row r of the matrix μ indicates how many times the attribute associated with r appeared in different objects (associated with columns). Two tasks are solved in the PL: 1) calculating the Hamming weights using the methods and tools from (Sklyarov *et al.*, 2013a); and 2) sorting the Hamming weights with the aid of the methods described above. The sorted values are used to simplify solving different problems from the scope of data mining.

Encountering the most frequent item is entirely done in hardware. Suppose we have a set of N sorted data items which might include repeated items and we need the most frequently repeated item to be found. This problem is solved in a hardware circuit shown in Fig. 8 (Sklyarov et al., 2014b) where N-1 comparators (Comp) form a binary vector. The most frequently repeated item can be discovered if we find the maximum number of consecutive ones in the vector and take the item from any input of the comparators that is used to form the sub-vector with the maximum number of successive ones. The binary vector that represents the result of comparison is saved in the feedback register *R*. The right-hand circuit in Fig. 8 implements the method described in (Sklyarov et al., 2014b), which enables the same combinational unit (such as that composed of AND gates in Fig. 8) to be reused iteratively in each subsequent clock cycle. This forces any intermediate binary vector that is formed on the outputs of the AND gates to be stored in the register *R*. Hence, any new clock cycle reduces the maximum number of consecutive ones O_{max} in the vector by one and as soon as all outputs of the AND gates are set to 0, we can conclude that $O_{max} = \xi+1$, where ξ is the number of the last clock cycle. Indeed, when there is just one value 1 in the register *R*, all the outputs of the AND gates are set to 0 and an additional clock cycle is not needed to reach a conclusion. The index of the single 1 in the register is the index (position) of the first value 1 (from the top) in the set with O_{max}. The feedback from the outputs of the AND gates enables any intermediate binary vector to be stored in the register *R*. The circuit in Fig. 8 is very simple and fast. It is composed of just N-1 AND gates, the register *R*, and minimal supplementary logic. Thus, the maximum attainable clock frequency is high.

5.2 Experimental Comparison of Software Only and Hardware/Software Sorters

Table 1 presents the result of our experiments that permit communication overheads to be estimated. We consider standalone applications and the following three types of data sorters:

- Software only sorters (see the row SO) where sorting is completely done in the software of the PS by C language qsort function. Initial data are taken from memory and the sorted data are saved in the same memory.
- Hardware only sorters (see the row HO) where sorting is completely done in the hardware of the PL without transmitting data items between the PS and PL. Initial data are taken from the PL registers and the sorted data are saved in the PL registers. We assumed that data items in the registers are ready before sorting and the results are not copied to anywhere. This case does not reflect reality but it is useful because it permits potentialities of hardware to be estimated.
- Hardware/software sorters (see the row HS) where unsorted blocks are copied from memory to the PL in AXI ACP burst mode and the sorted blocks are copied from the PL to memory in AXI ACP burst mode. The PS participates only in data transfers and does not execute merging. This case permits sorting in hardware plus communication overheads to be evaluated. The memory is shared between the PS and PL and will be used later on for subsequent merging of the sorted blocks in the PS.

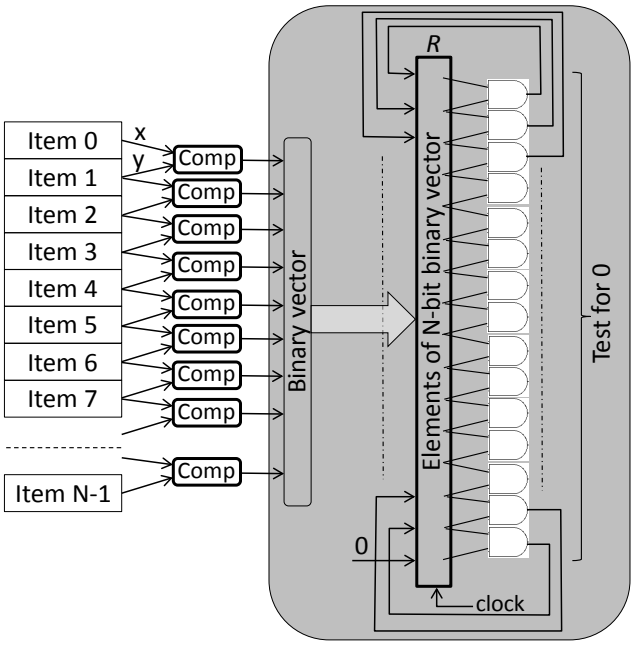

Fig. 8. Most frequent data item computation in a given sorted set of N data items.

Table 1. The results of experiments with one block of size N of 32-bit data items.

N		32	64	128	256	512
SO in µs	ZyBo	12.9	29.4	52.2	160.7	418.4
	Zed	12.0	28.2	51.6	160.5	417.7
HS in µs	ZyBo	2.7	3.6	5.6	-	-
	Zed	2.6	3.5	5.5	15.8	35.3
HO in µs	ZyBo	0.32	0.64	1.28	-	-
	Zed	0.32	0.64	1.28	2.56	5.12
Acc with CO (A_CO)	ZyBo	4.8	8.2	9.3	-	-
	Zed	4.6	8.1	9.4	10.2	11.8
Acc without CO (A)	ZyBo	40.3	45.9	51.7	-	-
	Zed	37.5	44.1	51.2	62.7	81.5

The rows *Acc with CO* (A_CO) and *Acc without CO* (A) show accelerations of HS and HO sorters compared to SO sorters (i.e. communication overheads - CO are either taken, in the row A_CO, or not taken, in the row A, into account). The clock frequency of the PS is 650 MHz for ZyBo and 666 MHz for ZedBoard (the rows Zed). The clock frequency for the PL was set to 100 MHz. The values in Table 1 are average times spent for sorting from 10 examples of randomly generated data. The iterative sorter in ZyBo for N=256 cannot be implemented because of the lack of hardware resources. We implemented the iterative sorter for N = 1,024 in ZedBoard but the remaining resources are not sufficient to provide support for interactions with the PS. We compared the results for two types of sorters: 1) from Fig. 2; and 2) from (Sklyarov et al., 2014a). The circuit from (Sklyarov et al., 2014a) has longer propagation delay. Thus, the clock frequency in the PL is decreased. Besides, the circuit in Fig. 2 enables resources to be reduced in almost

twice. Thus, the value N for the same PL resources is increased also by a factor of 2. Hence, larger blocks are sorted in hardware and the depth of merging in software is reduced. Since software is always slower (see conclusion below), the overall performance of HS solver is increased. From the results in Table 1 we can conclude the following:

1) Communication time (for data transfer between the PS and PL) is significantly larger than the time for sorting data in the PL by iterative networks. This is easily seen comparing values in the rows HS and HO and taking into account that no merging (or additional sorting operations) is done in software for the considered examples. Clearly, communication overhead is equal to HS time minus HO time. For example, for N=32, interactions between the PL and memory for ZyBo require additional time that is equal to 2.7 μs - 0.32 μs ≈ 2.4 μs. So, using faster but significantly more resource consuming sorting networks does not make any sense. Indeed, any additional acceleration in hardware (allowing the value 0.32 μs in the example above to be reduced) does not permit overall performance of HS sorters to be increased.

2) The row A makes sense only for evaluating hardware capabilities but the results in this row are not very important for practical applications requiring sorting large data sets because data items have to be transferred to/from internal registers and our experiments have shown that it takes significantly more time than sorting in the PL. Sorting larger data sets entirely in FPGA is possible but only in advanced devices, such as (Xilinx, 2014c, d).

3) The row A_CO shows actual accelerations in APSoC. It is clearly seen that the larger the size of the blocks, the higher is the acceleration. Taking into account the results of point 1), we can conclude that better acceleration can be achieved by increasing the size of blocks.

Let us now sort blocks in the PL and merge the sorted blocks in the PS. Since the results of Table 1 are very similar for ZyBo and ZedBoard, we will study the case with N=256 only for ZedBoard. Fig. 9 permits all the results of HS projects with different architectures to be compared.

We can see that dual-core implementations in HS sorters where software runs sequentially with hardware are the fastest. This is because hardware operates considerably faster than software, even including communication overheads. Hence, merging in software can be faster when both cores are involved. Sorting in hardware is completed much sooner than merging in software and although additional parallelism might give advantages, they do not appear in our projects. We explain this situation because of additional efforts done by the operating system (all multi-core implementations are executed under Linux) to support parallelism in software and in hardware, causing larger delays than sorting blocks in hardware. From the results of experiments we found the following: for the same N, the larger the number L of data items, the smaller is the acceleration. We can explain this by the necessity of more extensive data processing in software that is slower than hardware. Indeed, the size N of one block is the same, but the number of merges in software is larger and deeper for larger values of L. Thus, hardware processes blocks very fast (see Table 1) but merging in software becomes slower for larger data sets.

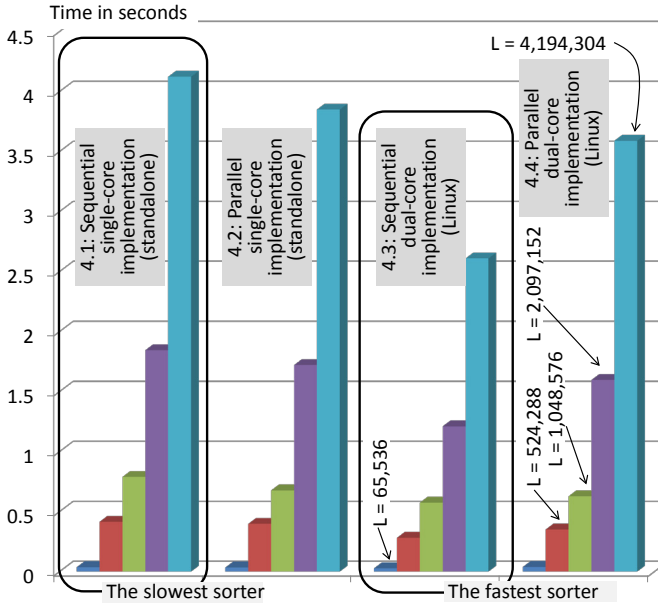

Fig. 9. The results of projects for architectures proposed in sections 4.1-4.4.

Let us discuss now potential future improvements. Fig. 10 shows the consumed time for sorting blocks of size N in SO using C function qsort and in HO using the iterative network from Fig. 2. The difference between SO and HO is very large (see the dotted area). Thus, future efforts have to be made to enlarge blocks processed in the PL.

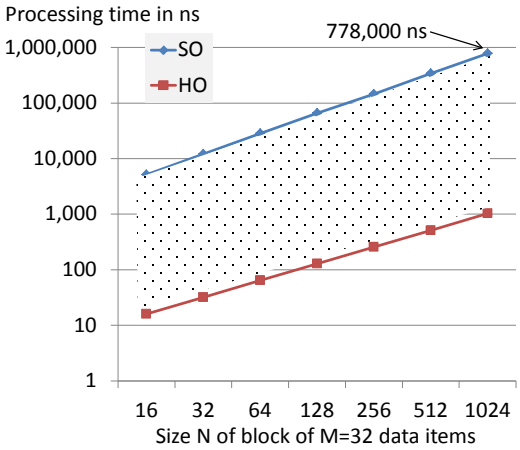

Fig. 10. Sorting blocks in software only (SO) and in hardware only (HO).

For example, we found that merging two blocks (N=256) in hardware using embedded to the PL block RAM takes 512 clock cycles. Subsequent merging of two new blocks with N=512 requires additional 1024 clock cycles. Thus, sequential sorting of 1024 data items for N=256 in the network of Fig. 2 and further partial merge in embedded to the PL block RAM would need

4×N+2×N+4×N=10×N=2,560 clock cycles or 25,600 ns assuming frequency 100 MHz. Potential parallel execution of different operations (i.e. *sorting+partial merging*) would permit the number of clock cycles indicated above to be additionally reduced. If we compare 25,600 ns with 778,000 ns for SO sorter (see Fig. 10), we can see that acceleration is by a factor of 30. The involved communication overheads significantly reduce this value, but the time of merging in the PS is also decreased. Thus, this technique can be additionally explored in the future research.

5.3 Discussion of the Results

It was demonstrated in sections 1, 2 that sorting is widely required for different applications in the scope of applied informatics and control engineering. It is clearly shown in subsection 5.2 that HS sorters are faster than SO sorters. We found also that AXI ACP port is faster than HP AXI port for standalone applications. The difference is the largest for small data sets and almost does not exist for large data sets. For projects running under Linux, the speed of data transfer is almost the same for AXI ACP and HP AXI. Both standalone and Linux projects are important and frequently practiced. The choice of a particular type depends on many factors, such as the complexity of the designed system, performance requirements, needs for device drivers available for operating systems and other factors. The results of experiments demonstrate that for both types of projects, HS solutions are always faster. One additional issue that should be addressed is the power consumption. We compared the power consumption in software only systems that use the ARM processor and the proposed hardware/software systems. Power estimation tools available for Xilinx Vivado allow necessary measurements to be done. We found that in the last case the power consumption is slightly increased but the difference is less than 5%. Note that the power consumption in new ARM-based microchips is decreased by a factor of 2-5 (Santarini, 2015). The results reported in (Kestur et al., 2010) show that FPGAs offer 2.7 to 293 times better energy efficiency compared to CPU and GPU for solving linear algebra problems. The FPGA-accelerated architectural simulation platform developed in (Krueger et al., 2011) to accurately evaluate the power and performance of the green wave design for seismic modeling also demonstrates reduction in power consumption. In (Mueller et al., 2012) power consumption was compared for a hardware/software data sorter and microprocessor-based data sorters. It was shown on examples that power consumption in FPGA hardware/software data sorters is lower. Taking into account the results reported, the proposed APSoC solutions are power efficient. Besides, much like (Bertels et al., 2010), our main objective is acceleration of algorithms in software/hardware data sorters compared to software only data sorters.

Our suggestion is that additional acceleration may be achieved by:

1) Increasing the size of the blocks sorted in hardware such that parallel merge of the sorted blocks (in software) and sorting of blocks (in hardware) can be done more efficiently.
2) Designing multi-core standalone applications. We have developed multi-core applications running under Linux operating system where programs with multiple parallel threads have been tested. However, potentialities for multi-core standalone applications may also be studied in depth.
3) Interaction between software and hardware through more than one HP AXI port. Zynq APSoCs permit up to five such ports to be used: four AXI HP ports and one AXI ACP port.
4) Parallel merge in more than two cores.

5.4 Comparison of the Results with Existing Solutions

There are three major procedures in software/hardware data sorters common to the proposed and existing solutions: 1) pre-processing in hardware; 2) communication between hardware and software; 3) post-processing in software. Let us compare these procedures in the proposed designs and in the best known alternatives. Pre-processing is aimed at sorting blocks of data that are further handled in software. There are two major characteristics of hardware sorters: a) the delay time from supplying unsorted inputs to producing the sorted outputs and b) the used resources. The best known solutions are based on Batcher's even-odd merge and bitonic merge networks (Baddar et al., 2011). If a pipeline is used (Mueller et al., 2012), then the delay is the same as in the networks described in the paper. However, hardware resources in the proposed networks are significantly decreased. For example, for N=16, they are decreased by a factor of 4.2 and for N=512 by a factor of 19. The details can be found in (Kipfer et al., 2005; Mueller et al., 2012; Sklyarov et al., 2014a). Our experiments have shown the following results: the best network from (Baddar et al., 2011) can be implemented in the PL of ZedBoard for N=64 and the proposed network can be built for N = 1,024. The network described in (Mueller et al., 2012) was built in FPGA for only N = 8.

Let us compare now post-processing in software. The FPGA accelerator in (Mueller et al., 2012) uses Batcher's networks (Baddar et al., 2011). The sorted blocks are then merged in software. For sorting L elements, $\lceil \log_2 L \rceil - \lceil \log_2 N \rceil$ merge levels are needed (Mueller et al., 2012). Since the value N is increased (from N = 8 in the referenced publication to N = 512 in the proposed network), the number of merge levels in software is decreased, for example, for $L = 2^{16}$ it is decreased from $\lceil \log_2 2^{16} \rceil - \lceil \log_2 8 \rceil = 13$ to $\lceil \log_2 2^{16} \rceil - \lceil \log_2 512 \rceil = 7$. We made experiments in the PS and merging blocks with N = 8 to sort L = 512 items requires more than 100,000 clock cycles while the same operation in the PL sorting network requires approximately 500 clock cycles. The clock frequency of the PS is 666 MHz and the clock frequency of the PL is 100 MHz. However, even taking into account the difference in the clock frequency, it is easily seen that the actual acceleration in the proposed solutions is significant. Communication between hardware and software involves additional time, which has been optimized (see section 3 and Fig. 2).

Finally, we compared the results of (Mueller et al., *2012)* shown in Fig. 29 and our results for the same values of L (from 256 to 16M data items) that are chosen in (Mueller et al., 2012) and found that sorting throughput in the proposed solutions has been increased by a factor from 1.6 (for the largest value of L) to almost 10 (for the smallest value of L). Comparison with other hardware/software data sorters referenced in section 2 has also demonstrated performance increase of the proposed solutions in all design cases.

6. CONCLUSIONS

Sorting and searching are common operations in different types of data processing required in various computing systems. Frequently, performance and portability are important and we have studied how these requirements can be satisfied using all programmable systems-on-chip from the Xilinx Zynq-7000 family. Four proposed architectures have been implemented and evaluated. They involve different iterative networks for sorting blocks of data in hardware and subsequent merge of the sorted blocks in software. The projects make use of parallelism including multi-core capabilities and data processing in hardware during data exchange. Two types of sorters have been developed that are standalone and running under the Linux operating system. The results of the paper demonstrate that hardware/software solutions are always the fastest. It is also proved that the larger the blocks processed in hardware, the better is the achieved acceleration. It is shown how to use the proposed solution in the scope of control engineering and applied informatics, including examples that were tested.

ACKNOWLEDGEMENTS

This work was supported by National Funds through FCT - Foundation for Science and Technology, in the context of the project PEst-OE/EEI/UI0127/2014.

REFERENCES

Abdelhamid N. (2015). Multi-label rules for phishing classification. *Applied Computing and Informatics*, Elsevier, vol. 11, pp. 29–46.

Aj-Haj Baddar S.W., Batcher K.E. (2011). *Designing Sorting Networks. A New Paradigm*. Springer.

ARM, Ltd. (2013). *NEON™ Version: 1.0 Programmer's Guide*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html.

Arnold O., Haas S., Fettweis G., Schlegel B., Kissinger T., Lehner W. (2014). An application-specific instruction set for accelerating set-oriented database primitives. *SIGMOD Conf.*, pp. 767-778.

Avnet, Inc. (2014). *ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide. Version 2.2*. http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.

Baker Z.K., Prasanna V.K. (2006). An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems, *Proc. 14th Annual IEEE Symp. on FCCM*, Napa, USA, pp. 67-75.

Batista A.P., Jota F.G. (2014). Effects of Time Delay Statistical Parameters on the Most Likely Regions of Stability in an NCS. *CEAI*, vol.16, no.1, pp. 3-11.

Bertels K., Sima V., Yankova Y., *et al.* (2010). HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms. *IEEE Micro*, vol. 30, no. 5, pp. 88-97.

Borangiu A., Popescu D. (2014). Digital Signal Processing for Knowledge Based Sonotubometry of Eustachian Tube Function. *CEAI*, vol. 16, no. 3, pp. 56-64.

Chamberlain R.D., Ganesan N. (2009). Sorting on Architecturally Diverse Computer Systems, *Proc. 3rd Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications – HPRCTA'09*, USA, pp. 39-46.

Chrysos G., Dagritzikos P., Papaefstathiou I., and Dollas A. (2013). CART: A Parallel System Implementation of Data Mining Classification and Regression Tree (CART) Algorithm on a Multi-FPGA System, *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 47.1-47.25.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009). *Introduction to Algorithms*. 3rd ed. MIT Press, Cambridge.

Digilent, Inc. (2014). *ZyBo Reference Manual*. http://digilentinc.com/Data/Products/ZYBO/ZYBO_RM_B_V6.pdf.

Estrin G. (1960). Organization of Computer Systems – The Fixed Plus Variable Structure Computer, *Proc. of Western Joint IRE-AIEE-ACM Computer Conference*, New York, pp. 33-40.

Field L., Barnie T., Blundy J., Brooker R.A., Keir D., Lewi E., Saunders K. (2012). Integrated field, satellite and petrological observations of the November 2010 eruption of Erta Ale, *Bulletin of Volcanology*, vol. 74, no. 10, pp. 2251–2271.

Firdhous M.F. (2010). Automating Legal Research through Data Mining, *Int. Journal of Advanced Computer Science and Applications*, vol. 1, no. 6, pp. 9-16.

Gapannini G., Silvestri F., and Baraglia R. (2012). Sorting on GPU for large scale datasets: A thorough comparison, *Information Processing and Management*, vol. 48, no. 5, pp. 903–917.

Garg L., McClean S., Meenan B.J., Millard P. (2011). Phase-Type Survival Trees and Mixed Distribution Survival Trees for Clustering Patients' Hospital Length of Stay. *Informatica*, vol. 22, no. 1, pp. 57-72.

Goodman A. (2011). Perspective Emerging Topics and Challenges for Statistical Analysis and Data Mining, *Statistical Analysis and Data Mining*, vol. 4, pp. 3-8.

Intel, Corp. (2007). *Intel® SSE4 Programming Reference*. http://home.ustc.edu.cn/~shengjie/REFERENCE/sse4_instruction_set.pdf.

Kestur S., Davisz J.D. Williams O. (2010). BLAS Comparison on FPGA, CPU and GPU. *Proc. of IEEE Computer Society Annual Symposium on VLSI*, Lixouri, Greece, pp. 288-293.

Kipfer P., Westermann R. (2005). *GPU Gems*, chapter 46. *Improved GPU Sorting*. http://http.

developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.

Knuth D.E. (2011). *The Art of Computer Programming. Sorting and Searching*, vol. III. Addison-Wesley.

Kovacs E., Ignat I. (2007). Clustering with Prototype Entity Selection Compared with K-means, *CEAI*, vol. 9, no. 1, pp. 11-18.

Krueger J., Donofrio D., Shalf J., *et al*. (2011) Hardware/Software Co-design for Energy-Efficient Seismic Modeling. *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, pp. 1-12.

Mueller R. (2010). *Data Stream Processing on Embedded Devices*. Ph.D. thesis, ETH, Zurich.

Mueller R., Teubner J., Alonso G. (2012). Sorting Networks on FPGAs, *The Int. Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1-23.

Rabiner L.R., Marvin R. Sambur M.R., Schmidt C.E. (1975). Applications of a nonlinear smoothing algorithm to speech processing. *IEEE Trans. on Acoustics, Speech and Signal Processing*, vol. 23, no. 6, pp.552-557.

Salter-Townshend M., White A., Gollini I., Murphy T.B. (2012). Review of Statistical Network Analysis: Models, Algorithms, and Software. *Analysis and Data Mining*, vol. 5, pp. 243-264.

Santarini M. (2014). Products, Profits Proliferate on Zynq SoC Platforms, *XCell Journal*, issue 88, pp.8-15.

Santarini M. (2015). Xilinx 16nm UltraScale+ Devices Yield 2-5X Performance/Watt Advantage. *XCell Journal*, issue 90, pp. 8-15.

Sedgewick, R. (1978). Implementing quicksort programs, *Communications of the ACM*, pp. 847-857.

Serban G., Câmpan A. (2008). Hierarchical Adaptive Clustering. *Informatica*, vol.19, no 1, pp. 101-112.

Silva J., Sklyarov V., Skliarova I. (2015). Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip. *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 31-34.

Sklyarov V., Skliarova I. (2013a). Digital Hamming Weight and Distance Analysers for Binary Vectors and Matrices, *International Journal of Innovative Computing, Information and Control*, vol. 9, no. 12, pp. 4825-4849.

Sklyarov V., Skliarova I. (2013b). Fast regular circuits for network-based parallel data processing, *Adv. Electr. Comput. Eng.*, vol. 13, no. 4, pp. 47–50.

Sklyarov V., Skliarova I. (2014a). High-performance implementation of regular and easily scalable sorting networks on an FPGA, *Microprocessors and Microsystems*, vol. 38, no. 5, pp. 470-484.

Sklyarov V., Skliarova I., Barkalov A., Titarenko L. (2014b) *Synthesis and Optimization of FPGA-based Systems*. Springer.

Sklyarov V., Skliarova I., Silva J., Rjabov A., Sudnitson A., Cardoso C. (2014c). *Hardware/Software Co-design for Programmable Systems-on-Chip*. TUT Press.

Sun S. (2011a). *Analysis and acceleration of data mining algorithms on high performance reconfigurable computing platforms*. Ph.D. thesis, Iowa State University. http://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=1421&context=etd.

Sun S., Zambreno J. (2011b). Design and Analysis of a Reconfigurable Platform for Frequent Pattern Mining, *IEEE Trans. on Parallel and Distributed systems*, vol. 22, no. 9, pp. 1497-1505.

Tukey J.W. (1977). *Exploratory Data Analysis*. Addison-Wesley.

Wu X., Kumar V., Quinlan J.R. *et al.* (2014). Top 10 algorithms in data mining, *Knowledge and Information Systems*, vol.14, no. 1, pp. 1-37.

Xilinx, Inc. (2014a). *Zynq-7000 All Programmable SoC Technical Reference Manual*. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

Xilinx, Inc. (2014b). *LogiCORE IP AXI DMA v7.1*. http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

Xilinx, Inc. (2014c). *Evaluation Board for the Virtex-7 FPGA User Guide. UG885 (v1.5)*. http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf.

Xilinx, Inc. (2014d). *All Programmable SoC Evaluation Kit (Vivado Design Suite 2014.3). UG961 (v6.0)*. http://www.xilinx.com/support/documentation/boards_and_kits/zc706/2014_3/ug961-zc706-GSG.pdf.

Yuan, Y. (2008). Step-sizes for the gradient method, *International Press American Mathematical Society*, pp. 785-796.

Zhang W, Thurow K., Stoll R. (2014). A Knowledge-based Telemonitoring Platform for Application in Remote Healthcare, *Int. Journal of Computers, Communications and Control*, vol. 9, no. 5, pp. 644-654.

Zmaranda D., Silaghi H., Gabor G., Vancea C. (2013). Issues on Applying Knowledge-Based Techniques in Real-Time Control Systems, *Int. Journal of Computers, Communications and Control*, vol. 8, no. 1, pp. 166-175.

Zuluada M., Milder P., Puschel M. (2012). Computer Generation of Streaming Sorting Networks, *Proc. of the 49th Design Automation Conference*, San Francisco, pp. 1245-1253.