Original scientific paper

Informacije MIDEM
Journal of Microelectronics,
Electronic Components and Materials
Vol. 45, No. 2 (2015), 142 – 152

Zynq-based System for Extracting Sorted Subsets from Large Data Sets

V. Sklyarov[1], I. Skliarova[1], A. Rjabov[2], A. Sudnitson[2]

[1]University of Aveiro / IEETA, Campus Universitário de Santiago, Aveiro, Portugal
[2]Tallinn University of Technology, Tallinn, Estonia

Abstract: The paper describes hardware/software architecture of a system for extracting the maximum and minimum sorted subsets from large data sets, two methods that enable high-level parallelism to be achieved, and implementation of the system in recently appeared on the market Zynq-7000 microchips incorporating a high-performance processing unit and advanced programmable logic from the Xilinx 7th family. The methods are based on highly parallel and easily scalable sorting networks and the proposed technique enabling sorted subsets to be extracted incrementally with very high speed that is close to the speed of data transfer through high-performance interfaces. The results of implementations and experiments clearly demonstrate significant speed-up of the developed software/hardware system comparing to alternative software implementations.

Keywords: processing system; programmable logic; system-on-chip; sorting networks; hardware/software co-design

Sistem na osnovi Zynq za izluščitev razvrščenih podsklopov iz obsežnih podatkovnih sklopov

Izvleček: Članek predstavlja programsko/strojno zasnovo sistema za izluščitev največjih in najmanjših razvrščenih podsklopov v obsežnih podatkovnih sklopih. Predstavljeni sta dve metodi, ki omogočata visoko stopnjo vzporednosti in implementacijo sistema v tržnem ZYNG-7000 mikročipu na osnovi programabilne logike Xilinx sedme generacije. Metode temeljijo na vzporedni in enostavno razširljivih omrežjih ter omogočajo izluščitev podsklopov s hitrostjo blizu hitrosti prenosa podatkov. Rezultati dokazujejo veliko pohitrenje programsko/strojnih rešitev v primerjavi s programskimi rešitvami.

Ključne besede: processing system; programmable logic; system-on-chip; sorting networks; hardware/software co-design

* Corresponding Author's e-mail: skl@ua.pt

1 Introduction

All Programmable Systems-on-Chip (APSoC) from Zynq-7000 family [1,2] combine on the same microchip the dual-core ARM® Cortex™ MPCore™-based high-performance processing system (PS) with advanced programmable logic (PL) from the Xilinx 7th family and may be used effectively for the design of hardware accelerators in such areas as hard real-time systems [3], image [4] and data [5] processing, satellite on-board processing [6], programmable logic controllers [7], driver assistance applications [8], wireless networks [9], and many others [2]. Interactions between the PS and PL are supported by different interfaces and other signals through over 3,000 connections [1]. Available four 32/64-bit high-performance (HP) Advanced eXtensible Interfaces (AXI) and a 64-bit AXI Accelerator Coherency Port (ACP) enable fast data exchange with theoretical bandwidths shown in [1].

Zynq APSoC design flow includes the development of hardware in the PL [10] (supported by available Xilinx IP cores) and software in the PS [11] for different types of applications such as standalone (bare metal) [12], running under an operating system (*e.g.* Linux) [12] and combined [13]. Hardware implemented in the PL can be the same for standalone and Linux applications but software programs use different functions and interaction mechanisms [12]. Since bare metal projects are generally faster, we will consider them as a base which does not exclude using the results for projects running under operating systems. The latter may benefit from available drivers and other support [12]. Since both types of projects can run in parallel in different cores [13] they may be combined if required.

Many electronic, environmental, medical, and biological applications need to process data streams produced by sensors and measure external parameters within given upper and lower bounds (thresholds) [14]. Let us consider some examples. Applying the technique [15] in real-time applications requires knowledge acquisition obtained from controlled systems (e.g. plant). For example, signals from sensors may be filtered and analysed to prevent error conditions (see [15] for additional details). To provide more exact and reliable conclusion a combination of different values need to be extracted, ordered, and analysed. Similar tasks appear in monitoring thermal radiation from volcanic products [16], filtering and integration of information from a variety of different sources in medical applications [17] and so on. Since many systems are hard real-time, performance is important and hardware accelerators may provide significant assistance for software products. Similar problems appear in so-called straight selection sorting (in such applications where we need to find a task with the shortest deadline in scheduling algorithms [18]), in statistical data manipulation and data mining (e.g. [19-22]). To describe one of the problems from data mining informally let us consider an example [19] with analogy to a shopping card. A basket is the 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 [19]. Similar problems appear to determine frequent inquiries at the Internet, customer transactions, credit card purchases, etc. requiring processing very large volumes of data in the span of a day [19]. Fast extracting the most frequent or the less frequent items from large sets permits data mining algorithms to be simplified and accelerated. Sorting of subsets may be involved in many known methods from this area [e.g. 20-22].

Let us consider a system that collects data produced by some measurements or copies such data from a database. A valuable assistance for applications described above may be provided by fast extraction of the maximum and minimum sorted subsets from the set of collected data, where the maximum/minimum sorted subset contains L_{max}/L_{min} data items. This problem can be solved in a software only system. For example, C function qsort permits large data sets to be sorted. After sorting is completed, extracting the maximum and minimum subsets may easily be done collecting them from the top and from the bottom of the sorted set. However, for many practical applications, such as that are referenced in [18,19], performance of the described above operations is important and software functions need to be accelerated. The paper suggests methods and high-performance implementations for solving the indicated above problem in APSoC from the Xilinx Zynq-7000 family.

The remainder of the paper is organized in five sections. Section 2 presents the proposed system architecture and describes overall functionality. Section 3 suggests two novel methods allowing the maximum and minimum sorted subsets to be extracted from large data sets. Section 4 shows how large subsets (for which hardware resources are not sufficient) can be computed and discusses additional capabilities. Implementation in Zynq microchip and the results of thorough evaluation and comparison of software only and software/hardware solutions with explicit indication of the achievable accelerations are discussed in section 5. Section 6 concludes the paper.

2 System Architecture and Functionality

The known results [2,5,12] have shown that software/hardware solutions may be significantly faster than software only solutions. Let us look at Fig. 1. Clearly, software/hardware system is faster if: $T_s > T_{sch} \leq T_{sh} + T_h + T_c$, where T_s, T_{sch}, T_{sh}, T_c, T_h are time intervals required for different modules. In highly parallel implementations software, hardware and interactions between hardware and software can run concurrently. For example, software may run in parallel with hardware; operations in hardware over previously received data may be done at the same time when new data are being transferred. Thus, $T_{sch} \leq T_{sh} + T_h + T_c$. This paper evaluates and compares software/hardware and software only solutions taking into account all the involved communication overheads and paying special attention to high level of parallelism. For instance we would like communication and application-specific operations to be overlapped in hardware as much as possible (see Fig. 1). Note that while hardware only designs may be the fastest, the complexity of such designs is often limited by the available resources in the PL.

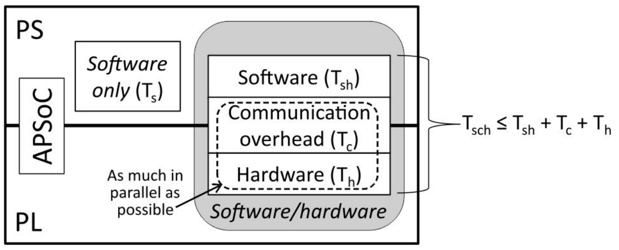

PS
Software only (T_s)
APSoC
Software (T_{sh})
Communication overhead (T_c)
Hardware (T_h)
As much in parallel as possible
PL
Software/hardware
$T_{sch} \leq T_{sh} + T_c + T_h$

Figure 1: Software only and software/hardware systems

Fig. 2 presents the proposed software/hardware architecture. Extracting subsets is done in an application-specific processing block (ASP) which is entirely implemented in the PL. We will discuss the ASP in the next section with all necessary details. There is another block in the PL called communication-specific processing (CSP) which interacts with the PS, i.e. it receives a large set of data items step by step in blocks and transfers the extracted sorted subsets. Besides, CSP is responsible for exchange of control signals between the PS and PL.

The PS is responsible for solving the following tasks:

1. Acquiring data and saving them in either on-chip memory (OCM) or external memory that is DDR.
2. Forming requests to extract subsets in the PL which is done through a set of control signals.
3. Collecting extracted subsets and storing them in OCM or external memory.
4. Verifying the results.
5. Solving exactly the same problem in software. This point is required just for experiments and comparison.
6. Computing the consumed time.

The PL is responsible for solving the following tasks:

1. Processing control signals received from the PS which are: a request (*start*) to begin data processing; source address in memory of input data (i.e. *the address of the set that has to be handled*); destination address in memory of output data (i.e. *the address to copy the extracted subsets*); *the number of blocks* Q of input data transferred from the PS to PL; and *the number of items* in the last block K_{last}. The PL also forms two signals that are sent to the PS which are: an interrupt generated as soon as the job is completed (i.e. the subsets have been extracted and copied to memory) and the number of clock cycles consumed in the PL which is needed for experiments and comparisons.
2. Extracting subsets on requests from the PS in highly-parallel ASP.
3. Counting clock cycles consumed in the PL from receiving the request up to generating the interrupt.

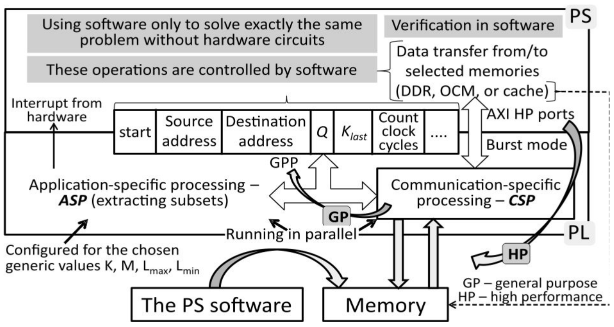
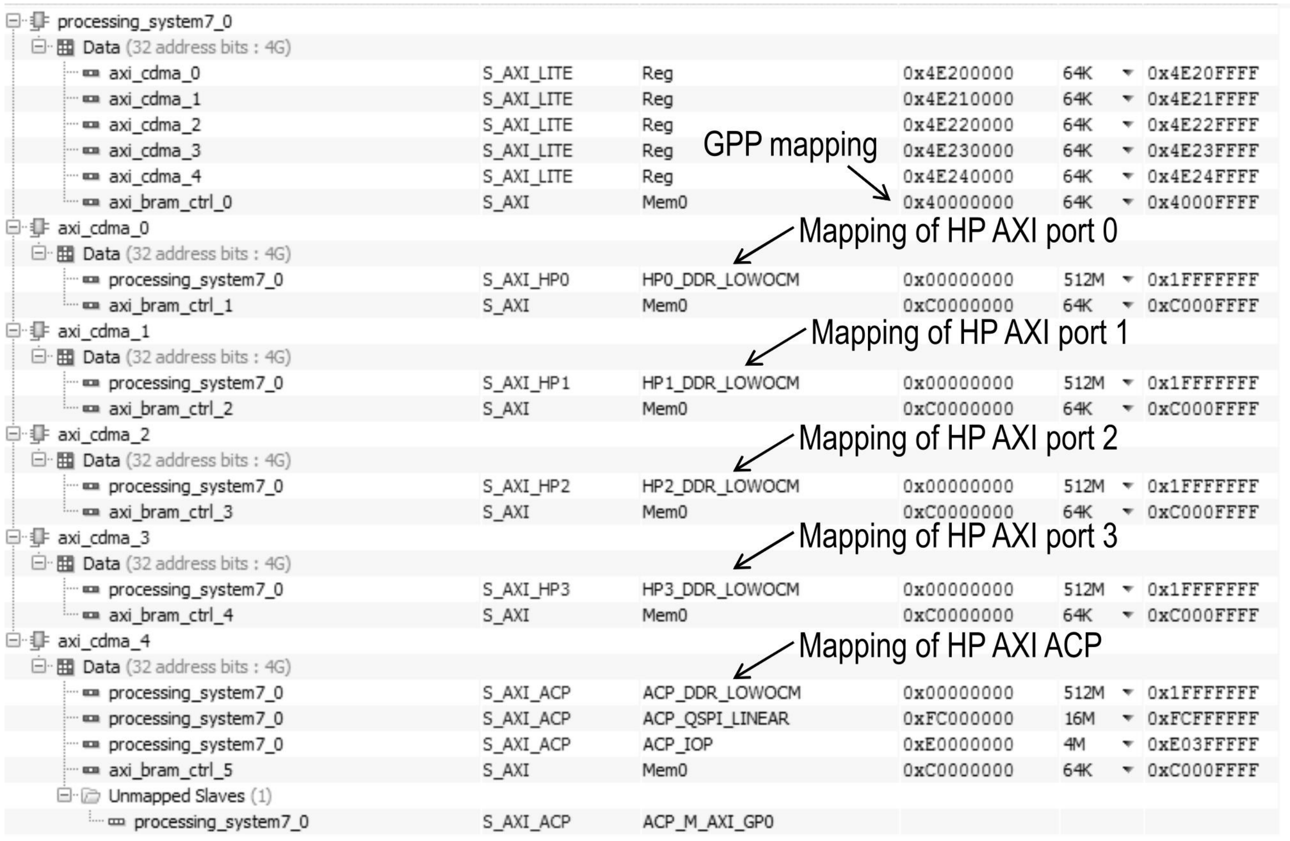

PS
Using software only to solve exactly the same problem without hardware circuits
Verification in software
These operations are controlled by software
Data transfer from/to selected memories (DDR, OCM, or cache)
Interrupt from hardware
start
Source address
Destination address
Q
K_{last}
Count clock cycles
....
AXI HP ports
Burst mode
GPP
Application-specific processing – ASP (extracting subsets)
Communication-specific processing – CSP
GP
Running in parallel
PL
HP
Configured for the chosen generic values K, M, L_{max}, L_{min}
GP – general purpose
HP – high performance
The PS software
Memory

Figure 2: The proposed software/hardware architecture



processing_system7_0					
Data (32 address bits : 4G)					
axi_cdma_0	S_AXI_LITE	Reg	0x4E200000	64K	0x4E20FFFF
axi_cdma_1	S_AXI_LITE	Reg	0x4E210000	64K	0x4E21FFFF
axi_cdma_2	S_AXI_LITE	Reg	0x4E220000	64K	0x4E22FFFF
axi_cdma_3	S_AXI_LITE	Reg	0x4E230000	64K	0x4E23FFFF
axi_cdma_4	S_AXI_LITE	Reg	0x4E240000	64K	0x4E24FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0x40000000	64K	0x4000FFFF
axi_cdma_0					
Data (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x00000000	512M	0x1FFFFFFF
axi_bram_ctrl_1	S_AXI	Mem0	0xC0000000	64K	0xC000FFFF
axi_cdma_1					
Data (32 address bits : 4G)					
processing_system7_0	S_AXI_HP1	HP1_DDR_LOWOCM	0x00000000	512M	0x1FFFFFFF
axi_bram_ctrl_2	S_AXI	Mem0	0xC0000000	64K	0xC000FFFF
axi_cdma_2					
Data (32 address bits : 4G)					
processing_system7_0	S_AXI_HP2	HP2_DDR_LOWOCM	0x00000000	512M	0x1FFFFFFF
axi_bram_ctrl_3	S_AXI	Mem0	0xC0000000	64K	0xC000FFFF
axi_cdma_3					
Data (32 address bits : 4G)					
processing_system7_0	S_AXI_HP3	HP3_DDR_LOWOCM	0x00000000	512M	0x1FFFFFFF
axi_bram_ctrl_4	S_AXI	Mem0	0xC0000000	64K	0xC000FFFF
axi_cdma_4					
Data (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x00000000	512M	0x1FFFFFFF
processing_system7_0	S_AXI_ACP	ACP_QSPI_LINEAR	0xFC000000	16M	0xFCFFFFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE0000000	4M	0xE03FFFFF
axi_bram_ctrl_5	S_AXI	Mem0	0xC0000000	64K	0xC000FFFF
Unmapped Slaves (1)					
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0			

Figure 3: Address mapping from Vivado 2014.2 block design editor

Note that for experiments and comparisons some additional signals for interactions between the PS and PL may be needed.

There are some generic parameters for which hardware in the PL is statically configured (see Fig. 2). They are:

K – the number of items that are handled in hardware in each block ($K_{last} \le K$);
M – the size of each data item;
L_{max} – the number of items in the maximum subset;
L_{min} – the number of items in the minimum subset.

Selection of proper AXI ports is very important. Experiments in [23] have shown that for transferring a small number of data items (from 16 to 64 bytes) general-purpose input/output ports (GPP) are always the best. In Zynq APSoC there are four available 32-bit GPP, two of which are masters and the other two are slaves from the side of the PS. They are optimized for access from the PL to the PS peripherals and from the PS to the PL registers/memories [24]. Since the latter feature is what we need, a master GPP was chosen for transferring control signals shown in Fig. 2. AXI ACP allows cache memory of application processing unit (APU) in the PS to be involved for data transfers and there exists an opportunity to provide either cacheable or non-cacheable data from/to the indicated above memories (i.e. OCM or DDR) [23]. Mapping of memories may be done in computer-aided design software (in our case in Xilinx Vivado block design editor according to addresses given in [1] and shown in Fig. 3, and in Xilinx Software Development Kit - SDK). Experiments in [12,23] have shown that for transferring large volumes of data items AXI ACP is very appropriate. Thus, this port was chosen to receive the source set from memory (OCM or DDR) in the PL and to copy extracted subsets from the PL to memory.

Fig. 4 gives more details about the chosen software/hardware interactions where: solid arrows (→) indicate who is the master (the beginning) and who is the slave (the end); triple compound lines show control flow; and dashed lines indicate directions of data flow (i.e. one direction - → or both directions - ↔). Control (and possibly a small number of additional auxiliary) signals are transferred through GPP. An initial (source) set and extracted subsets are copied through AXI ACP. The used memory (OCM or DDR) is indicated by the respective mapping both in hardware (see Fig. 3) and in software, which in our case was described in C language, and the mapping is done like the following:

Note that additional details about mapping with many examples can be found in [12].

The snoop controller [1] in Fig. 4 provides cacheable and non-cacheable access to memories (OCM or DDR) [1]. Cache area can be either disabled or enabled in software with the aid of function Xil_SetTlbAttributes [25]. In particular data *received from/copied to* memories may be pre-cached, i.e. they can be first saved into faster cache and then transferred with the main goal to increase performance of communications. Note that for standalone programs cache memory is entirely available. For programs running under an operating system (such as Linux) some area in cache memory may be used by programs of the operating system and the size of available cache memory is reduced. Many additional details can be found in [12].

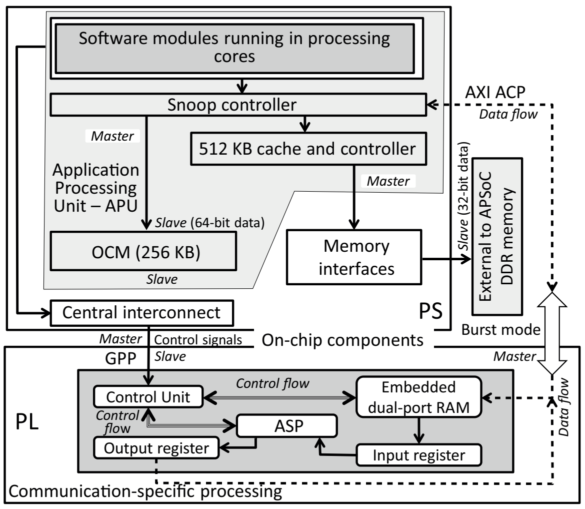


Figure 4: Hardware/software interactions

Initial (source) data set and extracted subsets are accommodated in memory as it is shown in Fig. 5. All necessary details about particular locations and sizes are supplied from the PS to PL through GPP (see Fig. 2).

To extract the maximum and/or minimum sorted subsets the following sequence of operations is executed:

1. The PS prepares source data in memory, calculates the number of blocks $Q = \lceil N/K \rceil$ (the value K is predefined), the number of items in the last block (which can be less than K), and indicates source and destination addresses. Here, N is the total number of data items that have to be processed.

```
#define OCM_ADDRESS                 0x00000000        // OCM address (see [1] for details)
#define DDR_ADDRESS                 0x16D84000        // DDR address (see [1] for details)
#define GPIO_BASE_IO_Control        0x40000000        // GPP address (see [1] for details)
#define HP_ADDRES                   OCM_ADDRESS       // for this example OCM address is chosen
```

2. The PS sets the start signal that is permanently tested in the PL.
3. As soon as the signal start is set, the PL transfers blocks of data in burst mode and saves them in a dedicated dual-port embedded block RAM (one port is assigned for transferring data from the PS to PL and another port for copying data from the block RAM to PL registers considered in the next section).

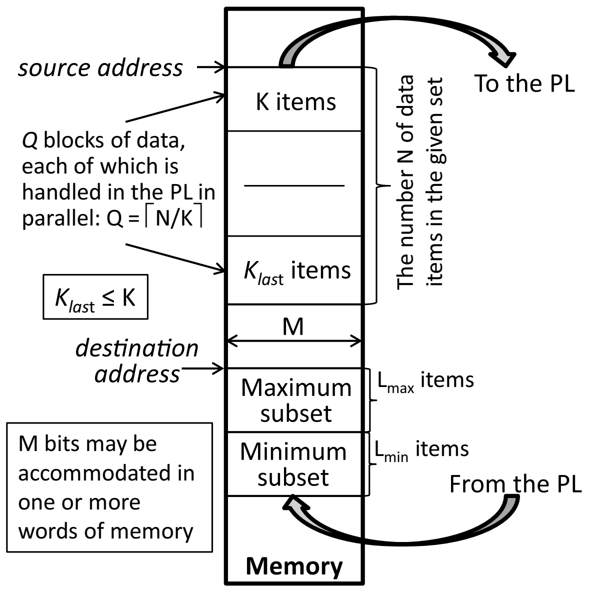

source address
K items
Q blocks of data, each of which is handled in the PL in parallel: Q = ⌈N/K⌉
K_{last} items
$K_{last} \leq K$
The number N of data items in the given set
To the PL
M
destination address
Maximum subset
L_{max} items
Minimum subset
L_{min} items
M bits may be accommodated in one or more words of memory
From the PL
Memory

Figure 5: Accommodation of the initial data set and the extracted subsets in memory

4. As soon as the first block is completely transferred to the block RAM through the first port, it is copied through the second port to PL registers that are used as inputs of sorting networks for extracting subsets in ASP.
5. The maximum and minimum subsets are incrementally constructed using methods from the next section and subsequent blocks of source data are transferred from memory to the block RAM in parallel.
6. The block RAM is organized as a circular buffer as it is shown in Fig. 6. If it becomes full data transfer is suspended until space for subsequent block is freed.
7. As soon as all Q blocks are processed the maximum and minimum subsets are ready (the details will be given in the next section).
8. The maximum and minimum subsets are copied from the PL to memory (see Fig. 5).
9. As soon as the previous point is completed, the PL generates a hardware interrupt to the PS indicating that the job has been finished (the details about such interrupts with examples can be found in [12]).
10. Optionally, the PL may count the number of clock cycles for solving the problem in hardware that it supplied to the PS through GPP.
11. PS may solve other problems in parallel with the PL. However, as soon as the interrupt is generated it is handled by the PS. Hence, the extracted subsets may immediately be used, for example, as data needed for projects of higher hierarchical levels.

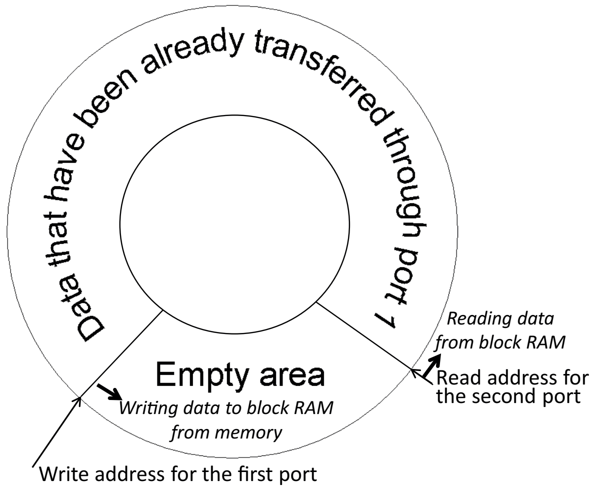

Data that have been already transferred through port 1
Reading data from block RAM
Empty area
Read address for the second port
Writing data to block RAM from memory
Write address for the first port

Figure 6: Block RAM organized as a circular buffer

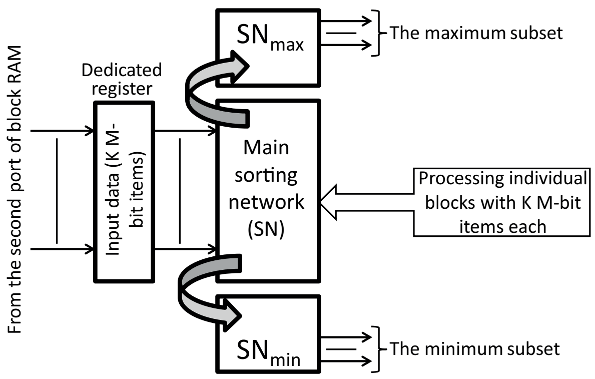
The circular buffer in Fig. 6 is managed by the PL control unit (see Fig. 4) that is a finite state machine. The buffer is built in the PL block RAM which is written through the first port (used for transfer data from the PS) and read through the second port (used to copy data from the block RAM to PL registers). As soon as the buffer is full, data transfer from the PS to PL is suspended. As soon as some area of the buffer is released (because data have already been read) data transfer is renewed.

3 Methods for Extracting Sorted Subsets

Let set S containing N M-bit data items be given. The maximum subset contains L_{max} largest items in S and the minimum subset contains L_{min} smallest items in S ($L_{max} \leq N$ and $L_{min} \leq N$). We mainly consider such tasks for which $L_{max} << N$ and $L_{min} << N$ which are more common for practical applications. Large and very large subsets may also be extracted and section 4 explains how to compute them. Experiments with such subsets are also reported in section 5. Sorting will be done in highly parallel networks, such as [26] or [27]. Since N may have very large value (millions of items) it cannot completely be processed in hardware due to unavailability of sufficient resources.

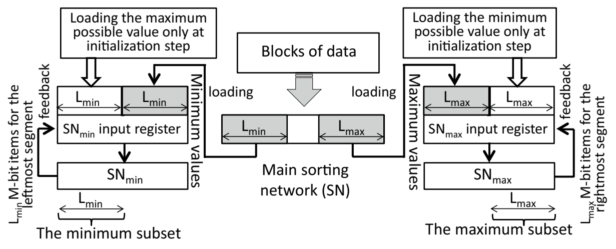
We suggest solving the problem iteratively using hardware architecture of ASP shown in Fig. 7. Data are incrementally received in blocks containing exactly K items and then processed by parallel networks described below. We mentioned above that the last block may contain less than K items. If so, it will be extended up to K items (we will talk about such extension a bit later). Part of sorted items with maximum values will be used to form the maximum subset and part of sorted items with minimum values will be used to form the minimum subset. As soon as all Q blocks have been handled the maximum and/or minimum subsets will be ready to be transferred to the PS.

We suggest two methods enabling the maximum and minimum sorted subsets to be incrementally constructed. The first method is illustrated in Fig. 8.



From the second port of block RAM
Dedicated register
Input data (K M-bit items)
Main sorting network (SN)
SN$_{max}$
SN$_{min}$
The maximum subset
The minimum subset
Processing individual blocks with K M-bit items each

Figure 7: Basic hardware architecture for ASP



Loading the maximum possible value only at initialization step
Loading the minimum possible value only at initialization step
Blocks of data
loading
loading
feedback
feedback
L$_{min}$
L$_{max}$
SN$_{min}$ input register
SN$_{max}$ input register
SN$_{min}$
SN$_{max}$
Minimum values
Maximum values
Main sorting network (SN)
L$_{min}$ M-bit items for the leftmost segment
L$_{max}$ M-bit items for the rightmost segment
The minimum subset
The maximum subset

Figure 8: The first method of extracting the maximum and minimum sorted subsets

Sorting networks SN$_{min}$ and SN$_{max}$ have input registers. The minimum and maximum sorted subsets will be built incrementally in halves of registers indicated at the bottom part of Fig. 8. At initialization step, these parts are pre-loaded with possible maximum and minimum values which data from the source set may have. Such values can be indicated by the PS in additional fields through GPP or calculated in the PL. Then the following steps are executed:

1. The first block containing K M-bit data items is copied from block RAM and becomes available at the inputs of the main SN.
2. The block is sorted in parallel in the main SN which can be done in combinational networks from [26] (such as even-odd merger) or in sequential iterative networks from [27] (such as iterative even-odd transition network). In the last case additional control is provided.
3. L$_{max}$ sorted items with maximum values are loaded in a half of the SN$_{max}$ input register as it is shown in Fig. 8. L$_{min}$ sorted items with minimum values are loaded in a half of the SN$_{min}$ input register as it is shown in Fig. 8. All the items are resorted by the relevant sorting networks SN$_{max}$ and SN$_{min}$.
4. A new block is copied from block RAM and becomes available at the inputs of the main SN. Such operations are repeated until all Q-1 blocks are handled.
5. The last block may contain less than K items and it is processed slightly differently. As soon as all Q blocks have been transferred from the PS to the PL block RAM and Q-1 blocks have been handled in ASP, the last block (if it is incomplete) is extended to K items by copying the largest item from the created minimum sorted subset. Thus, the last block becomes complete. Clearly, largest item from the created minimum sorted subset cannot be moved again to the minimum subset and the last block is handled similarly to the previous blocks.

Let as look at an example in Fig. 9.

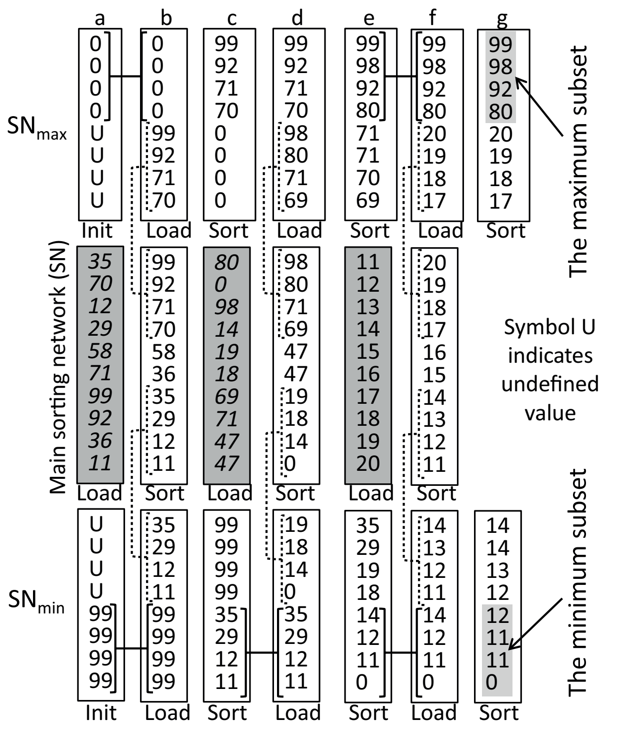


Figure 9: Example of extracting sorted subsets using the first method

It is assumed that the minimum possible value of data items is 0 and the maximum possible value is 99 (clearly, other values may also be chosen). At the first step (a), shown in left-hand part of Fig. 9, input registers for SN_{max} and SN_{min} are initialized, and the first block of data becomes available for the main SN. U indicates undefined values. At the next step (b) input registers are updated as it is shown by dashed fragments in Fig. 9. At step (c) a new block of data becomes available. Note that loading the register for the main SN can be done in parallel with copying L_{max}/L_{min} to SN_{max}/SN_{min}. Items in SN_{max} and SN_{min} are sorted as soon as the relevant input registers are updated. After executing steps (a) - (g) the maximum and minimum sorted subsets are ready (see the right-hand part of Fig. 9) for the items shown in grey in the main SN. Clearly, this method enables the maximum and minimum sorted subsets to be incrementally constructed for very large sets.

The idea of the second method is illustrated in Fig. 10 on the same example from Fig. 9.

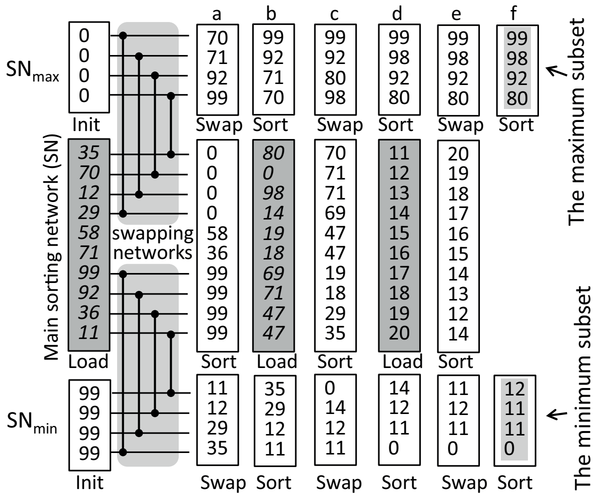


Figure 10: Example of extracting sorted subsets using the second method

Now the size of the networks SN_{max} and SN_{min} was reduced twice (there are now just 4 M-bit inputs instead of 8 in Fig. 9). Much like Fig. 8 both these networks have input registers (4 M-bit registers for our example). At initialization step SN_{max} and SN_{min} are filled in with the minimum and maximum values which are assumed as before to be 0 and 99. There are two additional fragments in Fig. 10 which contain circuits from [28]. They are composed of comparators shown in Knuth notation [29]. Any comparator converts a two-item input to the two-item output in such a way that the upper value is greater than or equal to the lower value. Let us call circuits from [28] a *swapping network*. If they are applied to two sorted subsets with equal sizes then it is guaranteed that the upper half outputs of the network contain the largest values from two sorted subsets and the lower half outputs of the network contain the smallest values from two sorted subsets. If we resort separately the upper and the lower parts then two sorted subsets will form a single sorted set. Let us analyse the upper swapping network in Fig. 10. At step (a) inputs of the network are sorted subsets {0,0,0,0} and {99,92,71,70}. Thus, two new subsets {70,71,92,99} and {0,0,0,0} are created. Sorting them enables the maximum sorted subset {99,92,71,70} with four items to be found on outputs of SN_{max}. At step (c) inputs of the swapping network are sorted subsets {99,92,71,70} and {98,80,71,69} and two new subsets {99,92,80,98} and {70,71,71,69} are created. Sorting them enables the maximum sorted subset {99,98,92,80} to be built. At step (e) inputs of the swapping network are sorted subsets {99,98,92,80} and {20,19,18,17} and no swapping is done. Hence, the maximum sorted subset is {99,98,92,80} and it is the same as in Fig. 9. The lower swapping network in Fig. 10 functions similarly.

The second method involves an additional delay on the comparators of swapping networks but eliminates copying (through feedbacks in Fig. 8) from the main SN to SN_{max} and SN_{min}. Besides, the sizes of SN_{max} and SN_{min} are reduced twice.

Let us discuss now an attainable complexity of sorting networks in the PL. It is shown in [5,27] that even in relatively complex field-programmable gate arrays (FPGAs) the size K is limited. For example, for even-odd merge and bitonic merge networks [26] K cannot exceed a few hundreds of 32-bit items even for very advanced FPGAs (such as the largest devices from the Xilinx Virtex-7 family [30]). In Zynq devices and circuits from [31] the maximum value of K cannot exceed 100 of 32-bit items. Iterative even-odd transition networks from [27] permit significantly larger number of items (exceeding thousands of 32-bit items) to be processed and they may efficiently be used for computing sorted subsets in hardware. Fig. 11 gives an example of the network from [27] which permits up to K = 16 data items to be sorted.

K M-bit data items that have to be sorted are loaded (from block RAM) to the feedback register (FR). Sorting is executed in a segment of even-odd transition network composed of two linked lines with even and odd comparators. Sorting is completed in K/2 iterations (clock cycles) at most. Note, that almost always the number of iterations is less than K/2 because of the technique [27] according to which if there is no swaps of data on the right-most line of the comparators then sorting is completed. Note that the network [27] possesses significantly smaller combinational delays than networks from [26]. Besides, in the proposed architec-

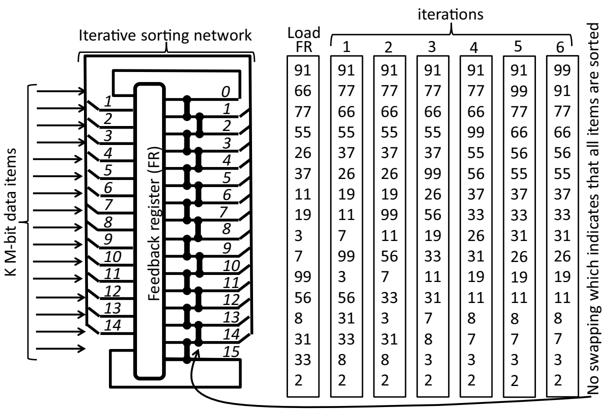


Figure 11: An example of iterative sorting network from [27] for K=16 data items

ture (see Fig. 4) iterations are done at the same time as subsequent data are being received from the PS. Such parallelism enables delays to be optimally adjusted allowing the total performance to be improved.

4 Computing Large Subsets and Additional Capabilities

For some practical applications the maximum and minimum subsets may be large and the available hardware resources become insufficient to implement sorting networks. Indeed, in accordance with [12] the largest sorting network that can be implemented in Zynq microchip xc7z020-1clg484c (that will further be used for experiments) is 512 32-bit items. The arising problem can be solved using the following technique. Let l_{max} and l_{min} be constraints for the upper (SN_{max}) and bottom (SN_{min}) parts in Fig. 7, i.e. the circuits SN_{max} and SN_{min} with larger values (than l_{max} and l_{min}) cannot be implemented due to the lack of hardware resources or because of some other reasons. Let the parameters for the maximum and minimum subsets be greater than l_{max} and l_{min}, i.e. $L_{max} > l_{max}$ and $L_{min} > l_{min}$. In such case the maximum and minimum subsets can be computed iteratively as follows:

1. At the first iteration, the maximum subset containing l_{max} items and the minimum subset containing l_{min} items are computed. The subsets are transferred to the PS (to memories). The PS removes the minimum value from the maximum subset and the maximum value from the minimum subset. Such correction avoids loss of repeated items at subsequent steps. Indeed, the minimum value from the maximum subset (the maximum value from the minimum subset) can appear for subsets to be subsequently constructed in point 3 below and they will be lost because of filtering (see point 3).
2. The minimum value from the corrected in the PS maximum subset is assigned to B_u. The maximum value from the corrected in the PS minimum subset is assigned to B_l. The values B_u and B_l are supplied to the PL through GPP.
3. The same data items (from memory), as in point 1 above, are preliminary filtered in the PL in such a way that only items that are less or equal than B_u and greater or equal than B_l are allowed to be transferred to block RAM, i.e. computing sorted subsets is done only for the filtered data items. Thus, the second part of the maximum and the minimum subsets will be computed and appended (in the PS) to the previously computed subsets (such as subsets from point 1).
4. The points 2 and 3 above are repeated until the maximum subset with L_{max} items and the minimum subset with L_{min} items are computed.

Note, that if the number of repeated items is greater than or equal to l_{max}/l_{min}, then the method above may generate infinite loops. This situation can easily be recognized. Indeed, if any new subset (that is sent from the PL to the PS) contains the same value repeated K times then an infinite loop will be created. In such case we can use another method based on software/hardware sorters from [12]. In the next section we will present the results of experiments for such sorters.

For some practical applications only the maximum or the minimum subsets need to be extracted. This task can be solved by removing the networks SN_{min} (for finding only the maximum subset) or SN_{max} (for finding only the minimum subset).

5 Implementations, Experiments and Comparisons

Fig. 12 shows the organization of experiments. We have used a multi-level computing system [12]. Initial (source) data are either generated randomly in software of the PS with the aid of C language rand function (see number 1 in Fig. 12) or prepared in the host PC (see number 2 in Fig. 12). In the last case data may be generated by some functions or copied from available benchmarks. Computing subsets in software/hardware systems is done completely in Zynq APSoC xc7z020-1clg484c housed on ZedBoard [32] with the aid of the described above software/hardware architecture (see Fig. 4). Computing subsets in software only sorters is completely done in the PS calling C language qsort function which sorts data and after that the maximum and minimum subsets are extracted from the sorted data. The results are verified in software running either in the PS (see number 3 in Fig. 12) or in the host PC (see number 4 in Fig. 12). Functions for verification of the results are given in [12]. Verification time is not taken into account in the measurements below. Methods that are used for copying files between the PC and APSoCs are explained in [12] with examples.

Synthesis and implementation of hardware modules were done in Xilinx Vivado 2014.2 design environment from specifications in VHDL. Standalone software applications have been created in C language and uploaded to the PS memory from Xilinx SDK (version 2014.2) using methods described in [12]. Interactions with APSoC are done through the SDK console window.

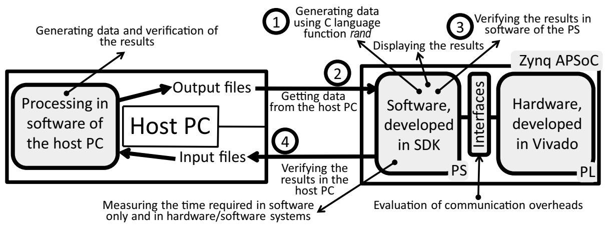

Generating data and verification of the results
1 Generating data using C language function *rand*
3 Verifying the results in software of the PS
Displaying the results
Zynq APSoC
2
Processing in software of the host PC
Host PC
Output files
Input files
Getting data from the host PC
4
Verifying the results in the host PC
Software, developed in SDK
PS
Interfaces
Hardware, developed in Vivado
PL
Measuring the time required in software only and in hardware/software systems
Evaluation of communication overheads

Figure 12: Experimental setup

For all the experiments 64-bit AXI ACP port was used for transferring blocks between the PL and memories. More details about this port can be found in [12,23,33]. The size of each block for burst mode is chosen to be 128 of 64-bit items (two 32-bit items are sent/received in one 64-bit word). Two memories were tested: the OCM and external (on-board) DDR. The OCM is faster because it provides 64-bit data transfers [1], but the size of this memory is limited to 256 KB. The available on ZedBoard 4 Gb DDR provides 32-bit data transfers.

The measurements were based on time units (returned by the function XTime_GetTime [34]) for $L_{max} = L_{min} = 64$, M=32, and K = 200. Each unit returned by this function corresponds to 2 clock cycles of the PS [35]. The PS clock frequency is 666 MHz. Thus, any unit corresponds to approximately 3 ns. The PL clock frequency was set to 100 MHz. Fig. 13 shows the time consumed for computing the maximum and minimum subsets for data sets with different sizes in KB (from 2 to 128). Since M=32 the number of processed words (N) is equal to the indicated size divided by 4. Fig. 14 shows the acceleration of software/hardware systems comparing to software only systems. Note that Figs. 13, 14 present diagrams for OCM. If DDR memory is used then communication overheads are slightly increased but acceleration in the software/hardware systems comparing to software only system is again significant. For M=64 speed-up is increased in almost 2 times.

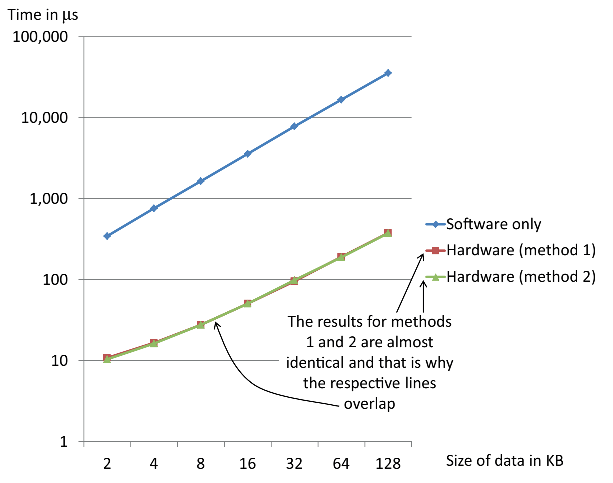


Figure 13: Computing time in software only and software/hardware systems

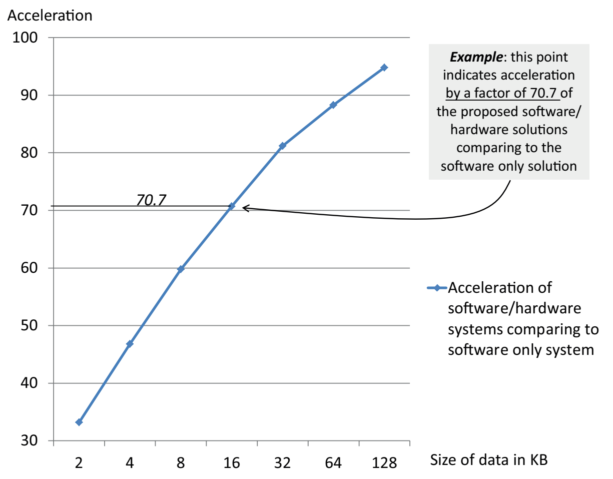


Figure 14: Acceleration of software/hardware systems comparing to software only system

If only the maximum or only the minimum subsets have to be computed the acceleration is almost the same, but the occupied hardware resources are reduced.

If the size of the requested subsets is increased in such a way that all data need to be read from memory several times (see section 4) then acceleration is decreased. Table 1 presents the results for extracting larger subsets (containing from 127 to 505 32-bit data items) from 128 KB set.

Table 1: The results for extracting larger subsets from 128 KB set

N	127	190	253	316	379	442	505
Time in µs	926.4	1,393.7	1,856.7	2,320.5	2,780.4	3,245.5	3,708.9

For very large subsets acceleration may even be less than 1, i.e. software only system becomes faster. In such cases software/hardware sorters from [12] can be used directly and they provide acceleration for all potential cases even for $L_{max} = N$ or $L_{min} = N$. Such acceleration is not as high as in Fig. 14 and it is equal to 6 for N = 512, K = 256 (now K is the size of blocks sorted in hardware and further merged in software) and 1.4 for N = 33,554,432, K = 256. These results were taken from experiments with data sorters from [12] (in all experiments M=32). We found that for small and moderate subsets the proposed here methods provide significantly better acceleration.

6 Conclusion

The paper suggests hardware/software architecture for fast extraction of minimum and maximum sorted subsets from large data sets and two methods of such extractions based on highly parallel and easily scalable sorting networks. The basic idea of the methods is incremental construction of the subsets that is done concurrently with transfer of initial data (source sets) through advanced high-performance interfaces in burst mode. Thorough experiments were done with entirely implemented on-chip designs in Zynq xc7z020-1clg484c device housed on ZedBoard. The size of initial sets varies from 512 to more than 33 million of 32-bit words. The results demonstrate significant speed-up comparing to pure software implementations in the same Zynq device, namely performance was increased by 1-2 orders of magnitude for small subsets and by a factor ranging from 1.4 to 6 for very large subsets.

7 Acknowledgments

This research was supported by EU through European Regional Development Funds, the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, ESF grant 9251, and Portuguese National Funds through FCT - Foundation for Science and Technology, in the context of the project PEst-OE/EEI/UI0127/2014.

8 References

1. Xilinx, Inc. (2014). Zynq-7000 All Programmable SoC Technical Reference Manual. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
2. Crockett L.H., Elliot R.A., Enderwitz M.A., and Stewart R.W. (2014). The Zynq Book. University of Strathclyde.
3. Hao L. and Stitt G. (2012). Bandwidth-Sensitivity-Aware Arbitration for FPGAs. *IEEE Embedded Systems Letters*, 4(3), 73-76.
4. Bailey D.G. (2011) Design for Embedded Image Processing on FPGAs. John Wiley and Sons.
5. Sklyarov V., Skliarova I., Barkalov A., and Titarenko L. (2014) Synthesis and Optimization of FPGA-based Systems. Springer.
6. Cristo, A., Fisher, K., Gualtieri, A.J., Pérez, R.M., and Martínez, P. (2013). Optimization of Processor-to-Hardware Module Communications on Spaceborne Hybrid FPGA-based Architectures. *IEEE Embedded Systems Letters*, 5(4), 77-80.
7. Canedo, A., Ludwig, H., and Al Faruque, M.A. (2014). High Communication Throughput and Low Scan Cycle Time with Multi/Many-Core Programmable Logic Controllers. *IEEE Embedded Systems Letters*, 6(2), 21-24.
8. Santarini, M. (2013). All Eyes on Zynq SoC for Smart Vision. *XCell Journal*, 83(2), 8-15.
9. Dick, C. (2013). Xilinx All Programmable Devices Enable Smarter Wireless Networks. *XCell Journal*, 83(2), 16-23.
10. Xilinx, Inc. (2014) Vivado Design Suite Guides. http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/design_tools.html.
11. Xilinx, Inc. (2014). Zynq-7000 All Programmable SoC Software Developers Guide. UG821 (v9.0). http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf.
12. Sklyarov, V., Skliarova, I., Silva, J., Rjabov, A., Sudnitson, A., and Cardoso, C. (2014) Hardware/Software Co-design for Programmable Systems-on-Chip. TUT Press.
13. Xilinx, Inc. (2013). Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors. http://www.xilinx.com/support/documentation/application_notes/xapp1078-amp-linux-bare-metal.pdf.
14. Sklyarov, V. and Skliarova, I. (2013). Digital Hamming Weight and Distance Analyzers for Binary Vectors and Matrices. *International Journal of Innovative Computing, Information and Control*, 9(12), 4825-4849.
15. Zmaranda, D., Silaghi, H., Gabor, G., and Vancea, C. (2013). Issues on Applying Knowledge-Based Techniques in Real-Time Control Systems, *International Journal of Computers, Communications and Control*, 8(1), 166-175.
16. Field, L., Barnie, T., Blundy, J., Brooker, R.A., Keir, D., Lewi, E., and Saunders, K. (2012) Integrated field, satellite and petrological observations of the No-

vember 2010 eruption of Erta Ale. *Bulletin of Volcanology*, 74(10), 2251–2271.
17. Zhang, W., Thurow, K., and Stoll, R. (2014). A Knowledge-based Telemonitoring Platform for Application in Remote Healthcare. *International Journal of Computers, Communications and Control*, 9(5), 644-654.
18. Verber, D. (2011), Hardware implementation of an earliest deadline first task scheduling algorithm. *Informacije MIDEM*, 41(4), 257-263.
19. Baker, Z.K. and Prasanna, V.K. (2006). An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems. Proc. 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, USA, 67-75.
20. Sun, S. (2011). 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.
21. Wu, X., Kumar, V., Quinlan, J.R., et al. (2014). Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1), 1-37.
22. Firdhous, M.F.M (2010). Automating Legal Research through Data Mining. *International Journal of Advanced Computer Science and Applications*, 1(6), 9-16.
23. Silva, J., Sklyarov, V., and Skliarova I. (2015) Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip. *IEEE Embedded Systems Letters*, 7(1), 31-34.
24. Neuendorffer, S., and Martinez-Vallina, F. (2013). Building Zynq Accelerators with Vivado High Level Synthesis. Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays, Monterey, CA, USA, 1-2.
25. Xilinx, Inc. (2014). OS and Libraries Document Collection UG647. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/oslib_rm.pdf.
26. Baddar, S.W.A.-H., and Batcher, K.E. (2011). Designing Sorting Networks. A New Paradigm. Springer.
27. Sklyarov, V., and Skliarova, I. (2014). High-performance implementation of regular and easily scalable sorting networks on an FPGA. *Microprocessors and Microsystems*, 38(5), 470-484.
28. Alekseev, V.E. (1969). Sorting Algorithms with Minimum Memory. *Kibernetica*, 5, 99-103.
29. Knuth, D.E. (2011). The Art of Computer Programming. Sorting and Searching, vol. III. Addison-Wesley.
30. Xilinx, Inc. (2014). 7 Series FPGAs Overview. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.
31. Mueller, R., Teubner, J., and Alonso, G. (2012) Sorting networks on FPGAs. *Int. J. Very Large Data Bases*, 21 (1), 1–23.
32. 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.
33. Sadri, M., Weis, C., When, N., and Benini, L. (2013). Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ. Proceedings of the 10th FPGAWorld Conference, Copenhagen/Stockholm.
34. Xilinx, Inc. (2013). LogiCORE IP AXI Master Burst v2.0. Product Guide for Vivado Design Suite. http://japan.xilinx.com/support/documentation/ip_documentation/axi_master_burst/v2_0/pg162-axi-master-burst.pdf.
35. Xilinx, Inc. (2014). Standalone (v.4.1). UG647. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/oslib_rm.pdf.

Arrived: 09. 11. 2014
Accepted: 14. 04. 2015