THESIS ON INFORMATICS AND SYSTEM ENGINEERING C126

Network-Based Hardware Accelerators for Parallel Data Processing

ARTJOM RJABOV

TTÜ PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on July 31, 2017.

Supervisors: Dr. Aleksander Sudnitsõn
Department of Computer Systems
Tallinn University of Technology
Tallinn, Estonia
Prof. Valery Sklyarov,
Dr. Iouliia Skliarova
University of Aveiro
Aveiro, Portugal

Opponents: Prof. Dr. Radomir Stankovic
University of Niš, Serbia

Dr. Johnny Öberg
KTH Royal Institute of Technology, Sweden

Defence of the thesis: [August 31, 2017, Tallinn]

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Artjom Rjabov/

ISSN 1406-4731
ISBN 978-9949-83-132-6 (publication)
ISBN 978-9949-83-133-3 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C126

Võrgupõhised riistvarakiirendid paralleelseks andmetöötluseks

ARTJOM RJABOV

TTÜ
KIRJASTUS

TABLE OF CONTENTS

LIST OF PUBLICATIONS

The work of this thesis is based on the following publications:

A Sklyarov, V.; Skliarova, I.; Rjabov, A.; Sudnitson, A. (2017). Fast Iterative Circuits and RAM-based Mergers to Accelerate Data Sort in Software/Hardware Systems, Proceedings of the Estonian Academy of Sciences, 66 (3), 323-335.

B Rjabov, A.; Sklyarov, V.; Skliarova, I.; Sudnitson, A. (2017). RAM-based mergers for data sort and frequent item computation. Conference on Information and Communication Technology, Electronics and Microelectronics (MIPRO2017), Opatija, Croatia, May 22-26, 2017, IEEE.

C Sklyarov, V.; Skliarova, I.; Rjabov, A.; Sudnitson, A. (2016). Computing Sorted Subsets for Data Processing in Communicating Software/Hardware Control Systems. International Journal of Computers Communications & Control, 11 (1), 126−141, 10.15837/ijccc.2016.1.

D Sklyarov, V.; Rjabov, A.; Skliarova, I.; Sudnitson, A. (2016). High-performance Information Processing in Distributed Computing Systems. International Journal of Innovative Computing, Information and Control, 12 (1), 139−160.

E Artjom Rjabov (2016). Hardware-based systems for partial sorting of streaming data. 15th Biennial Baltic Electronics Conference (BEC2016), Tallinn, Estonia, October 3-5, 2016. IEEE, 59−62.

F Sklyarov, V.; Skliarova, I.; Rjabov, A.; Sudnitson, A. (2015). Zynq-based System for Extracting Sorted Subsets from Large Data Sets. Journal of Microelectronics, Electronic Components and Materials, 45, 142−152.

G Rjabov, A.; Sklyarov, V.; Skliarova, I.; Sudnitson, A. (2015). Processing Sorted Subsets in a Multi-level Reconfigurable Computing System. Elektronika ir Elektrotechnika, 21 (2), 30−33, 10.5755/j01.eee.21.2.11509.

H Skliarova, I.; Sklyarov, V.; Rjabov, A.; Sudnitson, A. (2014). Fast Matrix Covering in All Programmable Systems-on-Chip. Elektronika ir Elektrotechnika, 20 (5), 150−153.

I Sklyarov, V.; Skliarova, I.; Rjabov, A.; Sudnitson, A. (2013). Implementation of Parallel Operations over Streams in Extensible Processing Platforms. The 56th IEEE International Midwest Symposium on Circuits and Systems (IEEE MWSCAS 2013), Columbus, Ohio, USA, August 4-7, 2013. IEEE, 852−855.

AUTHOR'S CONTRIBUTION TO THE PUBLICATIONS

Contribution to the papers in this thesis are:

- A The author participated in the decision-making process. The author implemented all hardware and software components of the prototype and conducted experiments. The author prepared the paper for publication.
- B The author developed the concept, implemented all software and hardware components and conducted experiments. The author wrote the paper and presented it at the conference
- C The author developed the concept through numerous discussions with the supervisors. The author executed necessary experiments. The author prepared the paper for publication.
- D The author participated in the decision-making process. The author implemented proposed the system and ran experiments. The author prepared the paper for publication.
- E The author developed the concept, implemented, proposed and investigated the system. The author executed the experiments. The author prepared the paper for publication and presented it at the conference.
- F The author participated in the decision-making process. The author implemented hardware and software parts of the proposed system. The author executed the experiments. The author took part in the preparation of the paper for publication.
- G The author developed the concept. The author implemented hardware and software components of the system. The author executed the experiments. The author prepared the paper for publication.
- H The author participated in the decision-making process. The author implemented all hardware and software components of the prototype and conducted experiments. The author took part in the preparation of the paper for publication.
- I The author participated in the decision-making process. The author implemented the prototype. The author took part in the preparation of the paper for publication.

OTHER RELATED PUBLICATIONS

- Rjabov, A.; Sudnitson, A.; Sklyarov, V.; Skliarova, I. (2016). Interactions of Zynq-7000 devices with general purpose computers through PCI-express: A case study. 18th Mediterranean Electrotechnical Conference (MELECON), Lemesos, Cyprus, 18-20 April 2016. IEEE, 1−4.
- Sklyarov, V.; Skliarova, I.; Silva, J.; Sudnitson, A.; Rjabov, A. (2015). Hardware Accelerators for Information Retrieval and Data Mining. 2015 International Conference on Information and Communication Technology Research (ICTRC), Abu Dhabi, United Arab Emirates, May 17-19, 2015. IEEE, 202−205.
- Sklyarov, V.; Skliarova, I.; Silva, J.; Rjabov, A.; Sudnitson, A.; Cardoso, C. (2014). Hardware/Software Co-design for Programmable Systems-on-Chip. TUT Press.
- Sklyarov, V.; Skliarova, I.; Silva, J.; Rjabov, A.; Sudnitson, A. (2014). Application of Extensible Processing Platforms for Experiments with FPGA-based Circuits. 2014 17th IEEE Mediterranean Electrotechnical Conference (MELECON 2014), Beirut, Lebanon, Apr. 13-16, 2014. IEEE, 467−471.
- Skliarova, I.; Sklyarov, V.; Rjabov, A.; Sudnitson, A. (2014). Hardware/Software Co-design in Extensible Processing Platforms for Combinatorial Search Algorithms. 2014 17th IEEE Mediterranean Electrotechnical Conference (MELECON 2014), Beirut, Lebanon, Apr. 13-16, 2014. IEEE, 462−466.
- Mihhailov, D.; Rjabov, A.; Sklyarov, V.; Skliarova, I.; Sudnitson, A. (2013). Optimization of Address-based Data Sorting Unit with External Memory Support. International Conference on Computer Systems and Technologies (CompSysTech 2013): International Conference on Computer Systems and Technologies (CompSysTech 2013), Ruse Bulgaria, June 28-29, 2013. ACM, 83−90. (ACM International Conference Proceeding Series; 767).

ABBREVIATIONS

AKS – a sorting algorithm. Named after its discoverers Ajtai, Komlós, and Szemerédi

APSoC – All Programmable System on Chip

ASIC – Application-specific integrated circuit

AXI – Advanced eXtensible Interface

BCP – Boolean Constraint Propagation

BM – Bitonic Merge

BRAM – Block RAM

CAE – Compare and Exchange (C/S)

CDMA – Central DMA

CGA – Cellular Genetic Algorithm

C/S – Comparaton/Swapper, Comparator/Switch

CPU – Central Processing Unit

CU – Control Unit

CUDA – Compute Unified Device Architecture

DDR – Double Data Rate,

DMA – Direct Memory Access

DPLL – Algorithm for solving the boolean satisfiability problem, named after its discoverers Davis–Putnam–Logemann–Loveland

EDA – Electronic Design Automation

FPGA – Field-Programmable Gate Array

FR – Feedback Register

FSM – Finite State Machine

GPC – General Purpose PC

GPU – Graphics Processing Unit

NP – Nondeterministic, Polynomial time

OCM – On-Chip Memory

OEM – Odd-Even merge algorithm

OETS – Odd-Even transposition sorter, Odd-Even transition sorter

PCI – Peripheral Component Interconnect

PCIe – PCI Express

PL – Programmable Logic

PS – Processing System

PUF – Physically Unclonable Functions

RAM – Random Access Memory

SAT – Boolean SATisfiability Problem

SIMD – Single Instruction, Multiple Data

SN – Sorting Network

VLSI – Very-large-scale integration

1. INTRODUCTION

The thesis explores methods of creating hardware accelerators for computationally intensive and resource consuming problems. It also addresses hardware/software co-design approaches for solving these tasks and studies different reconfigurable platforms.

The introductory chapter presents the motivation behind the thesis, followed by the problem formulation and the outline of main contributions. The last section of this chapter is an overview of the thesis structure.

1.1. Motivation

Fast information processing is in very high demand in electronic, environmental, medical, and biological applications. They frequently need to process data streams produced by sensors and calculate certain parameters [1]. Signals from sensors may need to be filtered and analyzed to prevent error conditions. To provide a more precise and reliable conclusion, combinations of different values need to be extracted, ordered, and analyzed.

Many methods that are used to solve such problems possess the need for parallel processing of data streams and high repetition of operations. Network-based hardware accelerators for such systems allow to process very high volumes of data simultaneously. The reconfigurable hardware platforms are very appropriate for implementation of such systems because of their low cost, flexibility, availability and many other advantages [2].

The use of reconfigurable technologies may help to overcome challenges that the area of hardware design faces nowadays. By reconfigurable technologies we commonly mean field-programmable gate arrays (FPGA) and programmable systems on chip (PSoC). Those platforms allow the productivity to be increased and time-to-market to be shortened, because of their relatively low cost and fast development methodology. They may be used effectively for both production of final products and design prototyping. New FPGAs constantly appearing on the market permit to design faster and more complex systems. Recently released multiprocessing systems, such as all-programmable ultra-scale PSoC, combine multicore processors, graphical processors (GPU), real-time processors and programmable logic providing us with the possibility to design embedded systems with computational power comparable with that of general purpose computers and lower power consumption [3].

1.2. Problem formulation

The current thesis is focused on network-based hardware accelerators for parallel data processing in the areas or combinatorial search (e.g. Boolean satisfiability and set/matrix covering) and data processing (e.g. sort, search and frequent item computations).

Sorting and searching procedures are needed in numerous computing systems [4]. They can be used efficiently for data extraction and ordering in information processing. Some common problems that they apply to are (see also Figure 1.1):

1. Extracting sorted maximum/minimum subsets from a given set.
2. Filtering data, i.e. extracting subsets with values that fall within given limits.
3. Dividing data items into subsets and finding the minimum/maximum/average values in each subset, or sorting each subset.
4. Finding the value that is repeated most often, or finding the set of n values that are repeated most often.
5. Removing all duplicated items from a given set.
6. Computing medians.
7. Solving the problems indicated in points 1-6 above for matrices (for rows/columns of the matrices).

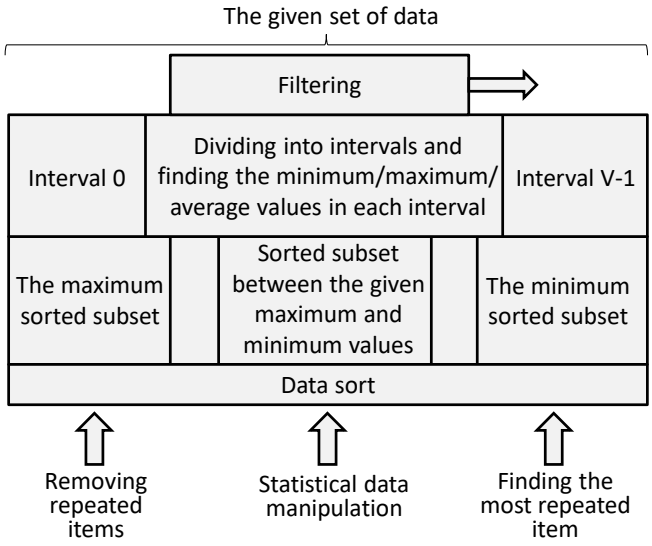

Figure 1.1 Common problems that are frequently solved in information processing systems [5]

We target our results towards reconfigurable platforms because these devices are regarded more and more as a universal platform that enables computational algorithms to be significantly accelerated. The following known architectures

are analyzed, compared, and explored in this work: 1) advanced FPGAs incorporating embedded blocks (DSP slices, embedded cores, etc.) and supported by existing soft cores; 2) programmable systems on chip (PSoC) that enable on-chip interactions between an embedded processing multi-core system and a reconfigurable logic with embedded blocks. The main idea is to select problems from the areas listed above and evaluate effectiveness of different architectures assuming also their potential combination in a new (proposed architecture) that might be the most efficient.

1.3. Contributions

In this thesis, we present novel methods and hardware/software architectures for acceleration of data sorting and merging, filtering and subset extraction, parallel covering of matrices/sets, Hamming weight computation. The results are presented in numerous recent publications. The proposed solutions outperformed many known alternatives (and many of them by a significant margin). Comparisons have been done with software only systems and other known FPGA-based systems known from publications.

The main contributions of this thesis are summarized as follows:

- Hardware/software architectures for fast extraction of minimum and maximum sorted subsets from large data sets and three methods of such extractions based on highly parallel and easily scalable sorting networks.
    - Three methods of subsets extraction.
    - Filtering and very large subsets extraction
- Hardware/software architectures for data sorting that involve sorting and merging operations.
    - The solution based on hardware sorting with subsequent software merging of sorted subsets using embedded processor of PSoC and general purpose PC.
    - The solution based on hardware sorting with subsequent hardware merge of small sorted subsets with software merge of larger subsets.
- Hamming weight/distance counters/comparators based on FPGA lookup tables (LUTs).
- A novel technique for implementation of matrix/set covering algorithms in hardware and software of recent all programmable systems-on-chip.

1.4. Thesis Organization

The remaining part of the thesis contains 5 chapters. Chapter 2 provides background information on hardware parallel processing and network-based design and makes a review of state-of-the-art in this area. It also contains a survey of related works of the topics.

Chapter 3 contains descriptions of all proposed methods of data sorting and merging, minimal and maximal subset extraction, Hamming weight calculation and matrix covering.

Chapter 4 presents architectures of hardware/software co-design based on FPGA, PSoC and general purpose PC. It also describes implementations of the proposed methods using these approaches.

Chapter 5 provides experimental results of the proposed methods using proposed hardware/software co-design approaches and contains comparison with known alternatives.

Chapter 6 concludes the thesis and discusses the directions for the further research.

2. RELATED WORKS

Highly parallel networks for sorting and searching enable numerous operations to be executed simultaneously. They have been extensively studied in VLSI area [6] [7]. These methods are very appropriate for devices which provide massive parallelism like GPU [8] or FPGAs [9] and PSoCs. All these platforms have their advantages and disadvantages. FPGA work on a relatively low speed, but provide flexibility which makes it possible to develop optimized application specific solutions. GPU usually work on much faster clock rates, but have fixed architecture. Additional advantage of FPGA is much better energy efficiency. GPU offer shorter development time, but recent high-level synthesis tools and emerging of hybrid PSoC platforms reduced development time for FGPA as well. Choosing the right platform is always a tradeoff between all these factors [10] [11] [12] [13].

2.1. Sorting

Parallel algorithms for data sorting have been studied in computer science for decades. There are many different parallel sorting algorithms [6]. Most notable of them are Parallel QuickSort [14], Parallel Radix Sort [15], Sample Sort [16] [17], Histogram Sort [18] and a family of algorithmic methods known as sorting networks [19]. The latter present a great interest for hardware acceleration. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagate through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line.

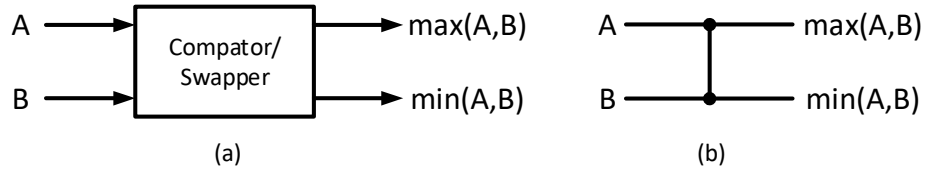

Figure 2.1 (a) A comparator/swapper block. (b) A comparator/swapper block in Knuth notation.

Sorting networks are composed solely from circuits known as comparators/swappers (C/S) or "compare and exchange blocks" (CAE). Single C/S unit is depicted in Figure 2.1(a). For inputs A and B the top output of C/S gives us the result of the function max(A,B), and the bottom output gives min(A,B). Figure 2.1(b) shows the most common way to represent sorting networks – Knuth notation or Knuth diagram. This notation is used in the rest of this work.

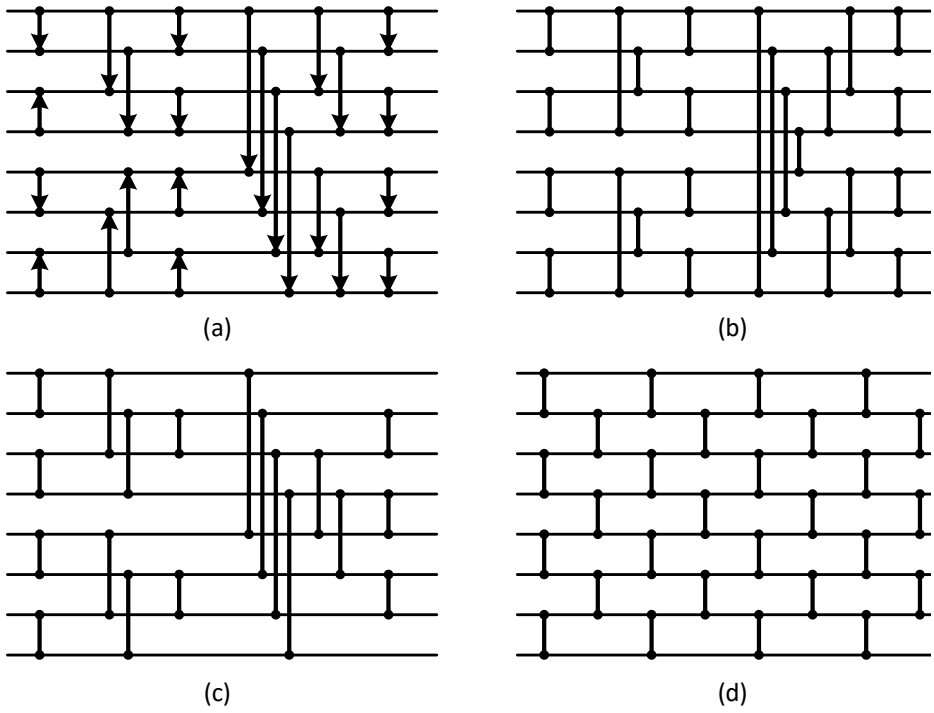

Figure 2.2 Different sorting networks with 8 inputs: (a) "Butterfly network" version of Bitonic sorting network. (b) Bitonic sorting network without reversal. (c) Odd-even merge sorting network. (d) Odd-even transposition sorting network.

The problem of finding the optimal sorting network is a very well-known problem in computer science and remains a subject of extensive research [20] [21]. One of the most famous results on the sorting network depth was obtained by Ajtai et al. in their AKS network [22]. However, further research showed that more common merge sorting networks require less comparator layers than this proposed network. AKS network is faster only for very large number of inputs and it is impossible to implement a network of that size with modern technology [6] [23].

The majority of modern hardware sorting network implementations use more practical even-odd and bitonic mergers invented by Kenneth E. Batcher [24] [19]. Bitonic sorting network is based on sorting of bitonic sequence. It is a sequence which monotonically increases and then monotonically decreases or can be modified by circular shifting to become monotonically increasing and decreasing. Original Batcher's design of Bitonic network is shown in Figure 2.2(a). It utilizes "butterfly network" concept where C/S blocks swap data in different direction. More common and intuitive representation of Bitonic network is shown in Figure 2.2(b). In this version of sorter all comparators point in the same direction. The rewiring was done based on the rule that every sequence of two sorted sets can become Bitonic by reversing one of them. Another Batcher's sorting algorithm is Even-Odd Merge sort. It is based on parallel merging of odd and even elements of two monotonic sequences with subsequent applying the column of parallel comparators. Sorting network based on Batcher's Odd-Even Merge algorithm is shown in Figure 2.2(c).

Other types are rarer (see for example the comb sort [25] in [26], the bubble and insertion sort in [27] [9]). Research efforts are concentrated mainly on networks with a minimal depth or number of comparators and on co-design, rationally splitting the problem between software and hardware. The regularity of the circuits and interconnections are studied in [28] [29] [30] where networks with iteratively reusable components were proposed.

A notable concept of sorting network design is a periodic network. The term has been proposed by Schröder in [31]. This type of network consists of identical sequences of comparators. The simplest and one of the most well-known examples is Odd-Even Transition (also known as Odd-Even Transposition or OETS) network depicted in Figure 2.2(d). It was proposed by Grasseli [32] and Kautz [33] and proved by Knuth in [4]. Traditional implementation of OETS is less efficient than Batcher's networks, but it is more reliable and simpler. Salloum and Wang proved that OETS has good fault-tolerant properties [34].

Hematian et al. proposed an optimized OETS network in [35]. They modified the network by connecting the first and the last items together and thus making the network in a ring shape. This approach reduces the total number of comparisons. It is shown in [36] that very regular odd-even transition networks with two sequentially reusable vertical lines of comparators are more practical because they operate at a higher clock frequency and provide sufficient throughput. These proposed improvements were developed with focus on FPGA implementation.

Two of the most frequently investigated parallel sorters on FPGAs are based on sorting [27] and linear [37] networks. Three types of sorting networks have been studied: pure combinational (e.g. [27] [9] [29]), pipelined (e.g. [27] [9] [29]), and combined (partially combinational and partially sequential) (e.g. [28] [30]). The linear networks, which are often referred to as linear sorters [37], take a sorted list and insert new incoming items in the proper positions. The method is the same as the insertion sort [4] that compares a new item with all

the items in parallel, then inserts the new item at the appropriate position and shifts the existing elements in the entire multi-item vector. Additional capabilities of parallelization are demonstrated in the interleaved linear sorter system proposed in [37]. The main problem with this is that it is applicable only for small data sets (see, for example, the designs discussed in [37], which accommodate only tens of items).

Sorting is a very computationally expensive and time consuming operation which requires a lot of hardware resources. There are different approaches proposed to overcome these limitations. Utilizing iterative networks with reusable comparators permits to process significantly larger data sets. Another two possibilities to get rid of these problems are utilization of a relatively small parallel sorter along with a merging circuit or implementation a partial sorting.

Different approaches of hardware sorting units were studied by Marcelino et al. in [38]. They implemented a hardware/software hybrid sorter with a sorting unit based on insertion sorting algorithm and unbalanced merging unit. They also utilized Batcher's Even-Odd sorting network for software implementation and experimented with different combinations of software (QuickSort, Even-Odd network) and hardware (Insertion sorting, unbalanced merge). They also discussed possibilities of using pipelined sorting networks and balanced merging units.

Another hardware merger based on a partial Bitonic mergers form [39] was proposed by Song et al. in [40]. They implemented a parallel pipelined merge tree based on this concept which can merge simultaneously up to 32 sorted data sets. Partial Bitonic sorters were used in their architecture instead of C/S blocks. This approach significantly speeds up the merge operation, but also requires more FPGA LUTs for the comparisons. Another advantage of their design in comparison to other merge-tree implementations is that it eliminates the intensive memory usage.

Chen and Prasanna in [41] proposed a hardware/software hybrid solution for accelerating database operations using FPGA and CPU. Their sorting algorithm is based on merge-sort algorithm where first few sorting stages are implemented in FPGA as folded bitonic sorting networks. The rest of the algorithm is implemented in CPU. The complete system was implemented in Xilinx Zynq ZC7020 PSoC device. Their hardware/software algorithm achieved 3.1x faster performance than software only (on the same CPU) performance.

GPU are also used for implementation of specific parallel algorithms such as sorting networks [42]. Buck and Purcell showed how to implement bitonic merge sort on GPU efficiently [43]. Kipfer and Westermann in [44] demonstrated implementation of Even-Odd Merge sort and improved efficiency of sorting by using full resources of GPU.

Greb and Zachmann in [45] presented a parallel sorting algorithm based on bitonic sort for GPU implementation. They reported slightly better results than previously published ones. Segupta et al. implemented radix-sort and quicksort

[46] [8]. Sintorn et al. also proposed implementation of QuickSort for GPU [47]. Radix-sort by Segupta showed 50% faster performance than plain bitonic sort. Quicksort implementation performed worse than sorting networks. Sintorn and Assarsson developed a sorter for GPU based on merge sort with introduction of partial quicksort and bucketsort sorting on the later stages of merge network to over merge sort disadvantages. They report that their system is 10% faster than GPU-based radix sort [48]. Satish et al. continued work on radix-sort and reported that their results were the best for GPU at that time [49]. Their algorithm is included in NVIDIA CUDA SDK since version 2.2. Leischner et al. proposed a comparison based GPU Sample sort which showed better results in some cases [50]. Another combination of bitonic sort and merging was proposed by Ye et al. in [51]. Their sorter performed faster than previous comparison-based techniques, but slower than radix-sort. All these solutions were developed for single GPU systems. Tanasic et al. proposed merge-sort based sorter for multi-GPU systems [52].

2.2. Partial sorting

Parallel sorters are very efficient, but their implementation is always limited by available resources. One of the possible solutions is to implement reduced sorters for partial sorting, because in many practical cases only partial sorting is needed. One of these cases is a maximum and minimum subsets finding.

The problem of finding subsets of minimum and maximum values is known, but very low number of solutions exist. The majority of works in this area are focused on finding 1 or 2 maximal or minimal values in data sets [53] [54] [55], but only few works are focused on subsets.

Frarmahini-Farahani et al. investigated the problem of partial sorting and max-set-selection in [39]. They proposed a modular design of a partial sorting system based on Batcher's Odd-Even and Bitonic sorting networks. Their system is built on sorting blocks constructed from Batcher's Odd-Even Merge (OEM) and Bitonic sorting networks (BM), where bitonic sorters are reduced in order to get sorted maximal (or minimal) subset. They also proposed an approach to select unsorted maximal subset by replacing bitonic sorters with maximum selection units. Their proposed system takes $N=2^n$ data items and extracts minimal or maximal sorted subset of $M=2^m$ items, where *n* and *m* are whole numbers and $1 \leq M < N$. In theory this technique is extendable to 2^n-to-2^m size. Also they proposed an architecture for iterative max selection units that can potentially work with data streams. Another solution of this problem was developed by Biroli and Wang in [56]. Their approach is not based on sorting networks, but still uses parallel comparators. They applied fast circuit topologies for single max/min value search by Goren et al. [54] to find a subset of the largest or smallest values. In contradiction to Frarmahini-Farahani they didn't use Batcher's networks. Both works focused on finding relatively small subsets.

The work by Frarmahini-Farahani is more suitable to deal with large subsets, but its expansion will lead to large area consumption.

Another example of partial sorting application is a classification problem [57] [58] (sometimes it is called simply „partial sorting“ [59] [60]). Bertrossi et al. proposed a classifying network in [61]. They developed two algorithms based on Leighton's Columnsort algorithm [23] which is based on sorting networks. The comparators in their designs were replaced by the classifying circuits [62]. They proved the efficiency of Columnsort algorithm for the classification problem solving. The classifying network based solutions showed better results than the traditional columnsort with comparator-based sorting networks.

2.3. Frequent items encountering

The majority of frequent item encountering techniques are software-based. Different algorithms and techniques were studied and compared in [63] by Cormode et al.

Teubner et al. suggested to use FPGAs in [64] and [65]. They proposed three different hardware designs with various trade-offs for the frequent item search. The first proposed solution is an almost straightforward hardware implementation of software Space-Saving algorithm with min-heap data structure in RAM blocks for data storage. The second solution is also based on the same algorithm, but instead of BRAMs with min-heap structure they used two search trees implemented in lookup tables in order to get rid of min-heap sorting. This approach showed significantly better results for relatively small amounts of data, but execution performance dropped with growing sizes of the circuit. In order to overcome the drawbacks of the second solution they decided to reduce the number of connections by using an array for the data storage, where each data is only connected to its two neighbors. The pipelined circuit of their third solution choses the best results in terms of performance and scalability. They achieved throughput four times higher than the best published result.

Shi et al. in [66] implemented a hardware accelerator for frequent item intersect algorithm Eclat. They designed a comparator network for two data vectors comparison. This circuit acts as a fragment of the algorithm merge part. According to their experimental result this approach showed from 6x to 26.7x speedup of the algorithm to the best software implementation existed.

2.4. Search problems

Examples of combinatorial search are matrix/set covering, the Boolean satisfiability (SAT), graph coloring and others. Many tasks are NP-complete and, thus, they are time consuming [67].

Given the broad applicability of SAT solvers, there has been much effort devoted to exploring efficient search strategies [68] [69] [70]. The majority of SAT solvers are based on the classic sequential Davis-Putham-Logemann-Loveland (DPLL) algorithm and its derivations. In recent years many parallel SAT solvers emerged [71]. Moreover, given the ease of parallelization of some parts of proposed algorithms, there has been much interest in the hardware implementation of SAT solvers. Large study of available hardware solvers was done by Skliarova and Ferrari in [72]. They also proposed their own novel hardware solver that utilizes matrix representation of Boolean functions.

Since then a few new approaches were suggested. Kanazawa and Maruyama developed a parallel hardware solver based on WSAT search algorithm. The circuit can be described as a network of buffers and clause elevators. The algorithm runs as many independent tries as possible and evaluates only clauses that are possibly unsatisfied by a flipping of a variable [73] [74].

Gulati et al. proposed a hardware SAT solver with the problem partitioning for ASIC in [75] and FPGA in [76]. Their FSM-based circuit performs the traversal of the implication graph and the conflict clause generation in parallel.

Haller and Sigh proposed another FPGA-based SAT solver which uses off-chip DRAM memory in order to overcome the on-chip memory limitation [77]. Davis et al. developed a hardware/software SAT solver, where only Boolean constraint propagation (BCP) is accelerated by hardware [68]. Suzuki and Maruyama implemented in [78] a partial hardware acceleration of SatElite algorithm in order to minimize DRAM delay.

Matrix representation of SAT problem also fits SIMD GPU approach. Luo and Liu implemented solvers based on greedy local search GSAT algorithm and genetic CGA algorithms in GPU. Their CGA implementation performed faster than the CPU [79]. Another GSAT-based GPU SAT solver was proposed by Deleau et al. in [80]. It showed poor results compared to CPU WalkSAT implementation. Meyer et al. proposed a CUDA SAT solver framework based of massive process parallelism [81].

Beckers et al. adapted a hybrid approach to GPU [82]. In their system the CPU executes MiniSAT algorithm while GPU runs parallel local search (Tabu Walk) and provides the Tabu list. Fujii and Fujimoto explored GPU-based acceleration of Boolean constraint propagation for SAT problem [83].

2.5. Hamming weight

The Hamming weight for a general vector (not obligatory binary) is defined as the number of its non-zero elements. Although many modern general purpose processors form Intel [84] and ARM [85] can calculate Hamming weight natively, it still presents an interest for hardware implementation because of its wide applicability.

King et al. in [86] proposed a fully combinational architecture for hardware digital Hamming weight comparator for artificial weightless neural network.

A Hamming weight comparator based on a bit sorter was proposed by Pedroni in [87]. It is a triangular matrix or a network of simple logic blocks made from trivial gates AND+OR. This circuit sorts ’0’ and ’1’ in a word and returns the sorted sequence. The HW comparison circuit is a reduced bit sorter. The unnecessary layers are removed.

Piestrak in [88] proposed another Hamming weight comparator circuit based on Knuth’s optimal sorting network. He developed two different comparison circuits. One that compares Hamming weight of a vector with some pre-defined threshold and another which compares Hamming weights of two vectors. This method showed significantly better results than Pedroni curcuit or other methods.

Parhami in [89] designed another Hamming height comparator based on Hamming weight counters that consists of a tree of ripple-carry adders. His circuit is capable of counting Hamming weight, comparing it with a fixed threshold as well as comparing two vectors. The reported experimental results showed an improvement over Piestrak’s results.

2.6. Hamming distance

Hamming weight is closely related to Hamming distance calculation. Many practical applications use Hamming weight calculators as a part of Hamming distance comparing units. In hardware implementations the Hamming distance is usually calculated by applying XOR operation to two vectors and subsequent Hamming weight calculation.

Although efficient Hamming weight calculators mentioned above can be used for computing Hamming distance, in many practical applications other, not very efficient solutions, were used. Appiah et al. [90] used a multiplexer network to calculate Hamming distance. Jin et al. in [91] developed a Hamming distance module for high-speed optical flow estimation. Their solution compares two 120-bit vectors which are divided into a pair of 15 contiguous 8-bit substrings. Basically their Hamming distance comparator is composed from Hamming weight calculators based on adder trees with additional buffers. Kovačević et al. in [92] used a network of adders for Hamming distance calculation in their Hamming neural network implementation.

2.7. Practical applications

Sorting and searching procedures are needed in numerous computing systems. They can be used efficiently for data extraction and ordering in information processing. Some common problems that they apply to are: extracting sorted maximum/minimum subsets from a given set; filtering data, i.e. extracting subsets with values that fall within given limits; dividing data items into subsets and finding the minimum/maximum/average values in each subset, or sorting each subset; finding the value that is repeated most often, or finding the set of n values that are repeated most often; removing all duplicated items from a given set; computing medians; solving the problems indicated above for matrices (for rows/columns of matrices).

Parallel sorters are in high demand in high-performance computing, including cosmological simulations [93]. Parallel sorting is also used in benchmarks for testing supercomputers [94]. Sorters based on sorting networks are suitable for hardware-based median filters which are commonly used in image processing [95]. The hardware median filter is a circuit that receives an array of data and returns the median value [96] [97] [98].

Many applications do not require all inputs to be sorted. Some of them involve selecting only maximal and minimal values. 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) [1]. Let us consider some examples. Applying the technique [99] in real-time applications requires knowledge acquisition obtained from controlled systems (e.g. plant). For example, signals from sensors may be filtered and analyzed to prevent error conditions (see [99] for additional details). To provide more exact and reliable conclusion a combination of different values need to be extracted, ordered, and analyzed. Similar tasks appear in monitoring thermal radiation from volcanic products [100], filtering and integration of information from a variety of different sources in medical applications [101] 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 [102]) and high-energy physics (where only the most energetic particles need to be analyzed [103]).

Maximum and minimum subsets extraction is required in searching, statistical data manipulation and data mining (e.g. [104] [105] [106] [107]). To describe one of the problems from data mining informally let us consider an example [104] 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 [104].

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 [104]. 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.

Full and reduced sorting networks are being studied in the area of correction problem, where sorting of almost sorted data set is required. The most recent works on this topic were done by Kik et al. [108] [109], Piotrów [110] and Stachowiak [111] [112].

In the scope of parallel vector processing we discuss practical application of Hamming weight and Hamming distance. Many analysis and filtering problems can be solved through Hamming weight counting for the vectors and comparison of the results.

Hamming weight and Hamming distance calculators and comparators are widely in use in variety of different applications. Hamming weight is a key part of many combinatorial search related tasks like Boolean satisfability and matrix covering. Hamming distance calculation is an essential operation in image recognition [90] [113] [114], and is used in many other areas like optical flow estimation [91] and more recently for physically unclonable functions (PUFs) [115] [116].

Hamming distance calculators are the essential part of Hamming neural networks [117]. It is a network which implements the optimum minimum error classifier, a unit that calculates Hamming distances of input data, compares them with pattern in memory and selects the data with the minimum distance, which becomes the first layer pattern that represents the most similar object. Lippman in [117] showed that this type of network has many advantages over the earlier Hoping network.

Combinatorial search algorithms are frequently involved to solve optimization problems. Matrix/set covering is one of the problems in optimization. It belongs to partitioning problems arising in such practical applications as scheduling aircrafts, location emergency stations in urban areas, fault testing of electronic circuits, resource distribution in multi-core systems, and many others [67]. Boolean satisfability problem solvers have many applications in EDA (Electronic Design Automation) fields, such as logic minimization, test pattern generation, routing in field-programmable devices [69].

2.8. Summary

This chapter contains a brief survey of related works. The purpose of this chapter is to inform a reader about the current state of the art in the researched area and to provide necessary information needed to understand subjects studied in this work.

This describes different hardware-based methods of parallel sorting, subset extraction and combinatorial search. Additionally it introduces the basic principles of network-based design. This survey showed us that although this area is very well researched, some topics covered in our works, like, for example subset extraction, are underdeveloped and very few publications about them exist. Also this chapter describes related works in a scope of Hamming weight and Hamming distance calculating. The last section of the chapter presents a survey of practical applications of our research subjects, which shows us that solving problems discussed in this thesis is in very high demand.

The most common drawbacks of the techniques and their implementations mentioned in this chapter are intensive resource usage and small volumes of data items that can be processed with them. Many sorting methods are not suitable for processing of high-speed data streams. Very few solutions exist in the area of partial data sorting and subsets extraction, as well as combined solutions of these operations, which are in a high demand in many practical applications. The techniques proposed in this thesis, which are based on highly parallel networks were developed in order to overcome these disadvantages.

3. NETWORK-BASED SOLUTIONS FOR PARALLEL DATA AND VECTOR PROCESSING

In this chapter the proposed methods of parallel data and vector processing are presented. We explore various highly parallel network-based algorithms for acceleration of solving different computationally intensive and resource consuming problems. Section 3.1 describes the proposed method for data sorting based on sorting network and merging [118] [5] [119]. Section 3.2 presents methods of solving the minimum and maximum subset extraction problem [120] [121] [122]. Section 3.3 discusses a solution for Hamming weight calculation [123]. Section 3.4 describes a practical application of the methods proposed in this chapter for fast matrix covering [124].

3.1. Data sorting

Sorting networks are widely used in data [9] and vector [88] processing and they enable comparison and swapping operations over multiple data items to be executed in parallel. A review of recent results in this area can be found in [36] where it is shown that many researchers and engineers consider such technique as very beneficial for data and vector processing in FPGAs and PSoCs. Although the methods [24] [19] enable the fastest theoretical throughput, the actual performance is limited by interfacing circuits supplying initial data and transmitting the results and the communication overheads do not allow theoretical results to be achieved in practical designs.

In our approach we use a periodical pipelined Odd-Even Transposition sorting network, which requires a significantly smaller number of comparators/swappers (C/S) than the most widely used Batcher's networks from [24] [19]. In this approach many C/S are active in parallel and reused in different iterations. The proposed circuit (see Figure 3.1) contains N M-bit registers $Rg_0,\ldots,Rg_{N-1}$. Unsorted input data are loaded to the circuit through N M-bit lines $d_0,d_{1,\ldots},d_{N-1}$. For the fragment on the left-hand side of Figure 3.1, the number N of data items is even, but it may also be odd. Each C/S is shown in Knuth notation (**:**) [4] and it compares items in the upper and lower registers and transfers the item with the larger value to the upper register and the item with the smaller value to the lower register (see the upper right-hand corner of Figure 3.1). Such operations are applied simultaneously to all the registers linked to even C/S in one clock cycle (indicated by the letter α) and to all the registers linked to odd C/S in a subsequent clock cycle (indicated by the letter β). This implementation may be unrolled to an even-odd transposition network [44], but vertical lines of C/S in Figure 3.1 are activated sequentially and the number of C/S is reduced compared to [44] by a factor of N=2. For example, if the number N is even then the circuit from [44] requires $N \times (N - 1)/2$ C/S and the circuit in Figure 3.1 – only $N - 1$ C/S. The circuit in [44] is combinational and the circuit in Figure 3.1 may require up to N iterations. The number N of

iterations can be reduced very similarly to [29]. Indeed, if beginning from the second iteration, there is no data exchange in either even or odd C/S, then all data items are sorted. If there is no data swapping for even C/S in the first iteration, data swaps for odd C/S may still take place. Note that the network [44] possesses a long combinational delay from inputs to outputs. The circuit in Figure 3.1 can operate at a high clock frequency because it involves a delay of just one C/S per iteration (i.e., in each rising/falling edge of the clock).

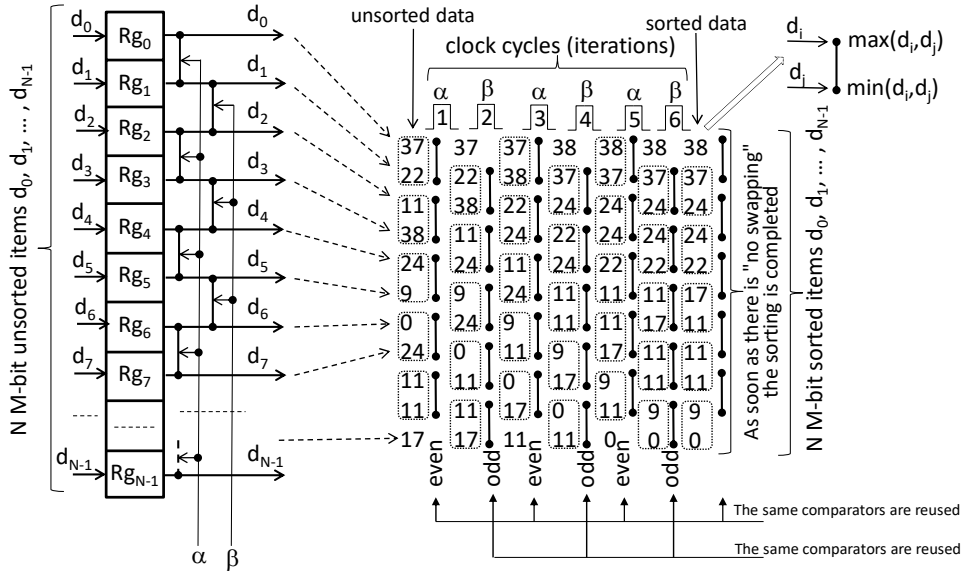

Figure 3.1 Pipelined Odd-Even Transposition network [118]

Let us look at the example shown in Figure 3.1 (N = 11, M = 6). Initially, unsorted data d_0, d_1,…,d_{10} are copied to Rg_0;…;Rg_{10}. Each iteration (6 iterations in total) is forced by an edge (either rising or falling) of a clock. The signal α activates the C/S between the registers (Rg_0,Rg_1),(Rg_2,Rg_3),…,(Rg_8,Rg_9). The signal β activates the C/S between the registers (Rg_1,Rg_2),(Rg_3,Rg_4),…,(Rg_9,Rg_{10}). There are 10 C/S in total. Rounded rectangles in Figure 3.1 indicate elements that are compared at iterations 1-6. Data are sorted in 6 clock cycles and 6 < N = 11. Unrolled circuits from [44] would require 50 C/S with the total delay equal to the delay of N sequentially connected C/S.

Although the proposed approach requires less C/S blocks than the most practically used Batcher's networks and allows to sort significantly larger amounts of data, resources still restrict from sorting very large amounts of data in parallel. In order to overcome this obstacle we propose different approaches of sorting network combinations with software and hardware merging of sorted data subsets.

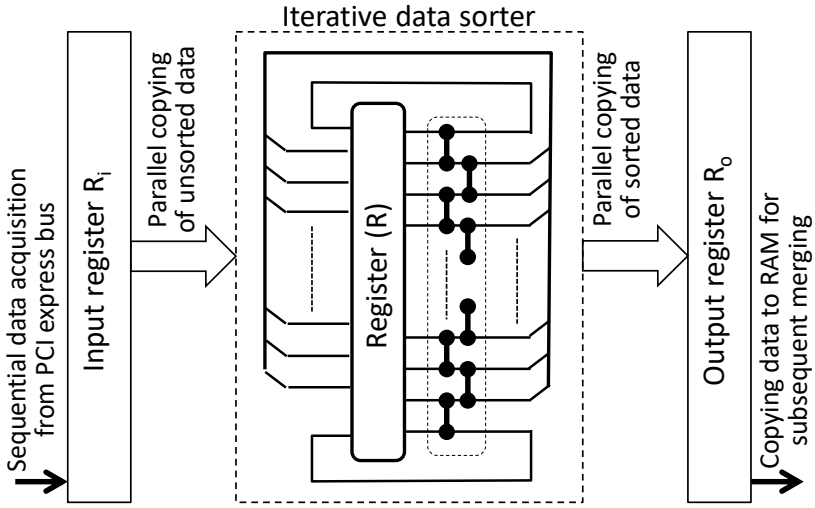

Figure 3.2 The circuit for sorting blocks [118]

Figure 3.2 depicts the used iterative network, the core of the hardware architecture. There are also two additional registers R_i and R_o. The register R_i sequentially receives N data items from the inputs. It was explained above that such N items compose one block that can be entirely sorted in the network. In practice, four items are packed and thus, parallel writing to the register R_i of four 32-bit items is actually done. As soon as the first block is received, all data items from this block are sorted in the iterative network, and the maximum number of clock cycles is N/2. At the same time, data items from the next block are received. As soon as data items from the first block are sorted, they are copied in parallel to the output register R_o. After that the second block is copied to the register R and sorted (see Figure 3.2) and the third block is being received from the inputs. At the same time, the first sorted block is copied to the embedded block-RAM for subsequent data merging. Hence, the first sorted block will be copied to RAM after acquisition of two blocks from the inputs. Then data acquisition from the inputs, data sorting, and copying data to the merger will be done in parallel. We can see from Figure 3.2 that there are just two sequential levels of C/S in the iterative data sorter. Thus, the delay is very small and we can apply high synchronization frequency. Additional improvements are done to adjust the speed of data acquisition and sorting. Indeed, one block of N data items is received in N/4 clock cycles and the sorting time is up to N/2 clock cycles, i.e. it is almost two times longer.

Figure 3.3 demonstrates how to adjust the speed. There are now two iterative data sorters running in parallel. The first sorter processes data from the first half of the register R_i and the second sorter processes data from the second half of the register R_i. At the beginning, two blocks with 2×N items are copied to the R_i and it involves $2\times N/4 = N/2$ clock cycles. Then two blocks are sorted in parallel, which also involves up to N/2 clock cycles. Finally, two sorted blocks

are copied to two dual-port embedded block-RAMs. The respective write port is configured for data width 64. Thus, pairs of data items are copied in each clock cycle and it involves totally also N/2 clock cycles for both blocks. Therefore, everything is completely adjusted.

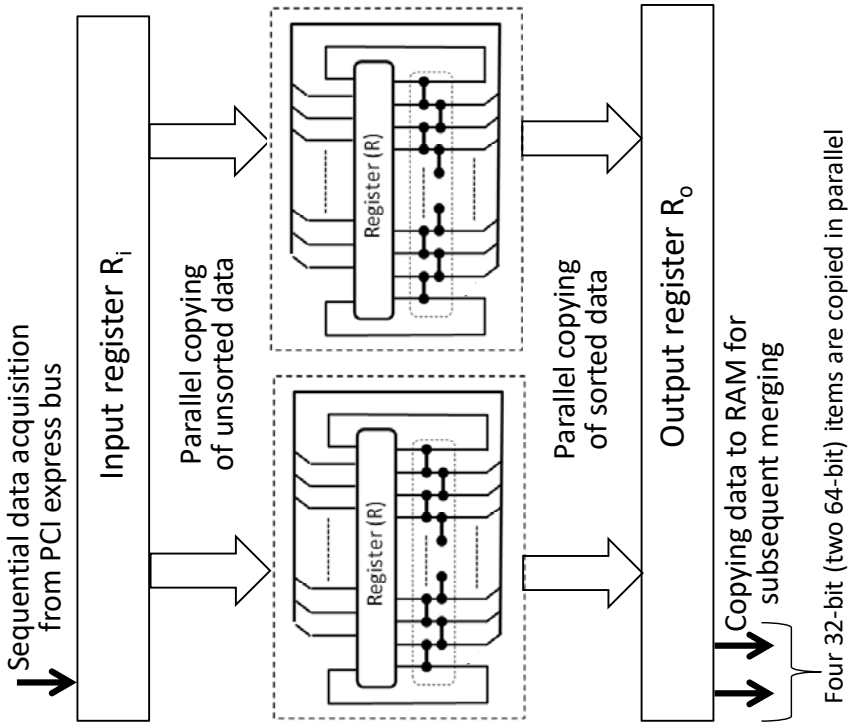

Figure 3.3 Adjusting the number of clock cycles required in different blocks [118]

3.1.1. Data sorting with subsequent software merge

Although using iterative periodic sorting networks permits sorting significantly larger data sets, than with other sorting networks, resource availability still puts us into certain boundaries. The limitation of the input data size might be unacceptable for many practical applications and the sorter must be designed with the capability of sorting the unlimited data sets.

Data sorting can be combined with the data merge in order to overcome this problem. Using hardware network-based sorter together with a general purpose processor allows us to implement the merge operation completely or partially software.

The first approach is a hardware/software system which sorts blocks in hardware with subsequent merging in software. This method requires use of a general purpose CPU which works in cooperation with hardware modules. We

have developed two variations of this method. The first uses a system with programmable logic and embedded CPU on the same chip such as PSoC. The second requires external CPU which communicates with the hardware through some interface like very high speed PCI express.

The proposed system sorts relatively small subsets of larger sets in hardware and then merges the subsets in software of a higher level system (see Figure 3.4). The initial set of data that is to be sorted is divided into L subsets of N items. Each subset is sorted in hardware using the referenced networks. Merging is executed as shown in Figure 3.4, in a host system/processor that interacts with the hardware. Each horizontal level of merging permits the size of blocks to be doubled. Thus, if $N = 2^{10} = 1{,}024$ and $K = 2^{20} = 1{,}048{,}576$ items are to be sorted, then 10 levels of mergers are required (see Figure 3.4). Clearly, the larger are the blocks sorted in FPGAs the fewer merging are needed. Thus, we have to sort in hardware as many data items as possible with such throughput that is similar to throughput of sorting networks.

This algorithm is identical for all the proposed implementations and can be described as follows. The sorter receives blocks composed of N M-bit data items and stored in memories (such as external DDR and OCM). The sorter executes iterative operations over multiple parallel data and is controlled by a dedicated finite state machine (FSM) called Sorter Control Unit. The ports are also controlled by a dedicated FSM. The results of sorting are copied back to memory and then the software merges incoming blocks of sorted data.

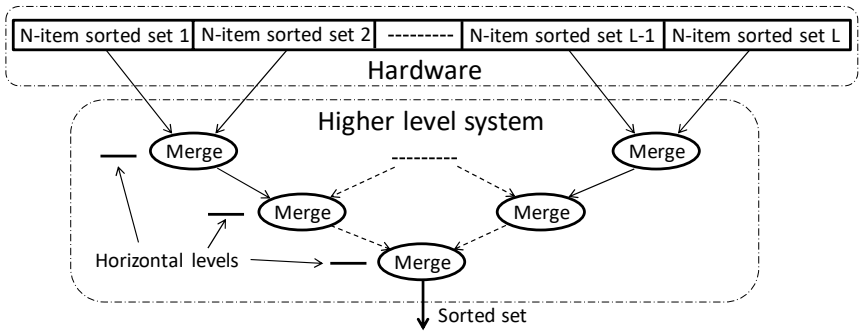

Figure 3.4 Hardware/software system for data sorting and merging [118]

The hardware part informs the general purpose processor through interrupt signals when the sorting operation over data blocks is completed. After receiving the interrupts signal the software merge operation triggers. The software and hardware parts access the data in memory independently. The software is in idle state while waiting for the data to become available. The sorting in hardware and merging in software can be done in parallel if necessary. The size of the data blocks depends only on resource availability of the chosen device.

3.1.2. Data sorting with subsequent hardware merge

The second method is similar to the previous one, but it is capable of supplying larger blocks of sorted data for subsequent data merge in software. The main idea behind this method is to implement the merge system in hardware alongside the sorting network. This method significantly increases amount of data that can be sorted in hardware. Although the sorting network works much faster than the merge algorithm, the latter requires much less resources. The combination of the sorting network with the merging tree significantly enhances the sorting for very large data sets. Figure 3.5 demonstrates this architecture.

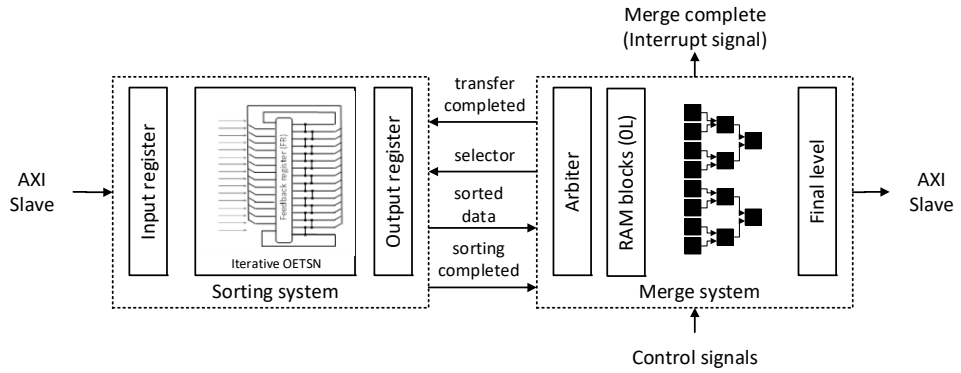

Figure 3.5 Hardware architecture of sorter/merger

The hardware implementation of this method consists of two major components – sorting system and merge system. The sorting system is identical to the one described in the previous method, but all its outputs are connected now to the merging system. The merge component performs the merge algorithm using a tree-like structure, which is done on the basis of embedded block-RAM. Figure 3.6 shows one level of merging. Input data come from two embedded block-RAMs, merged, and copied to a new embedded block RAM. There are two address counters for each input RAM. At the beginning they are set to 0. Two data items are read and compared. If the item is selected from the first RAM then the address counter of the first RAM is incremented, otherwise the address counter of the second RAM is incremented. Two N-item blocks are merged in 2×N clock cycles. Different types of parallel merging have been verified and compared. We found that the best result (i.e. the fastest and the less resource consuming) is produced in a simple RAM-based circuit depicted in Figure 3.6.

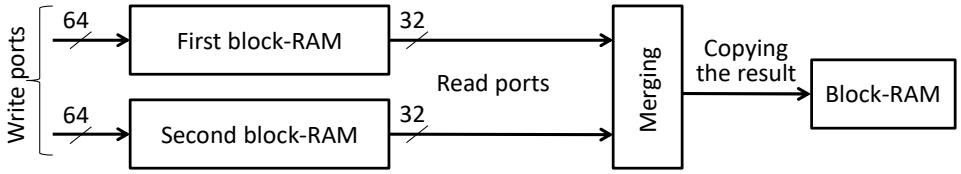

Figure 3.6 Simple merging two sorted blocks [118]

There are G levels to merge L sorted blocks and $^{2G-1} < L \leq 2^{G}$. The first level is composed of L embedded block-RAMs. The second level is composed of L/2 embedded block-RAMs, and the last level is composed of one embedded block-RAM. The size of each RAM for the first level is N 32-bit words for reading and N/2 64-bit words for writing. The size of each subsequent level is doubled. Initially, L embedded block-RAMs of the first level are filled in with sorted blocks. Then these blocks are merged at the second level. Afterwards the blocks of the second level are being merged at the third level and at the same time the block-RAMs of the first level are being filled in with a new subset of L sorted blocks. Thus, many subsets of L blocks will be processed in parallel and this is a special type of pipeline organized based on embedded block-RAMs (see Figure 3.7).

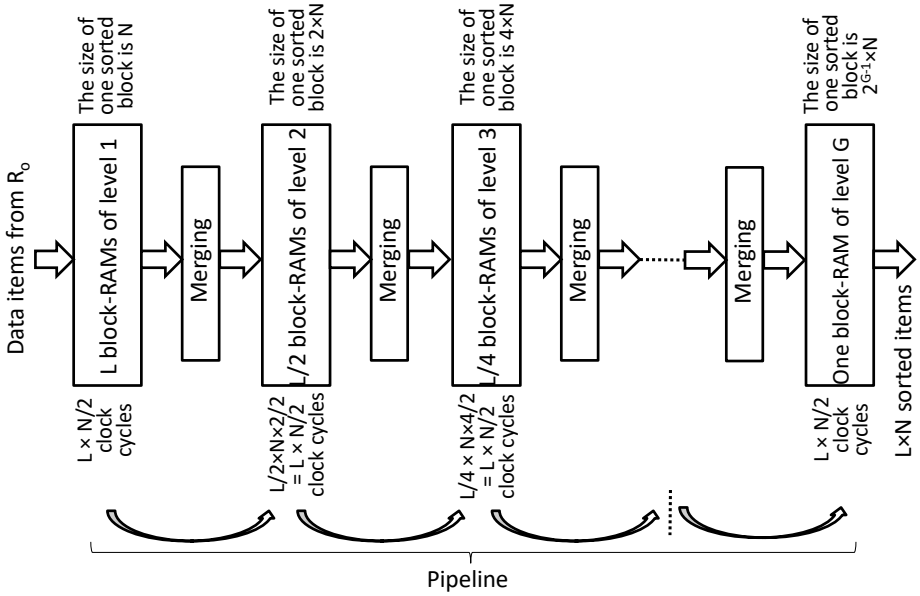

Figure 3.7 Pipelined merging with embedded block-RAM [118]

Architecture in Figure 3.7 permits many sets with L blocks (each block contains N M-bit data items) to be sorted in pipeline in a way that is shown in Figure 3.8. Equal numbers enclosed in circles indicate steps executed in parallel. It was shown above that the first time the level 1 block-RAM will be filled in with

sorted data from the first block is after 3×N/2 clock cycles. After that it is updated with the new block in N/2 clock cycles. So, an additional delay appears just from the beginning and it is avoided in the subsequent steps. As soon as data are copied to the first level RAM, merging is started and the sorted data are copied from the first level to the second level RAM. This process involves L×N/2 clock cycles. During this period of time the first-level RAM is used for merging and new data items cannot be copied to this RAM. In fact it is possible to merge and to sort data at the same time. However, we found that such merger requires a complex arbitration which significantly increases hardware resources leading to reducing the size N of blocks. Finally, such more complicated circuits do not give any advantage. This means that the resulting throughput cannot be increased. As soon as merging is completed, all data are copied to the second-level RAM and the first-level RAM may be refilled with new L sorted blocks.

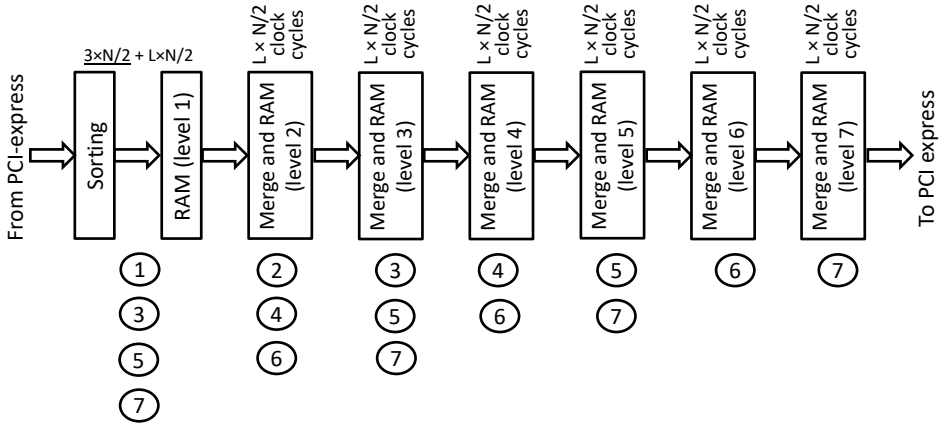

Figure 3.8 Parallel operations in the proposed architecture [118]

Figure 3.8 explicitly indicates parallel operations. For example, number 7 enclosed in circle indicates operations executed in parallel, which are merging at levels 3, 5, 7 and data sorting. This method can be applied to data sorting of very large sets (tens and hundreds of millions of data items). In this case, the GPC divides a very large set into subsets composed of L×N data items. The subsets are sorted in the pipelined structure shown in Figure 3.8 and then merged in software of GPC. Section 5.1.2. demonstrates that the implemented in Virtex-7 FPGA data sorter allows to sort data in hardware for L=128 and N=512. Thus, 512×128 = 65,536 32-bit data items (or 256 KB) are sorted and then 256 KB blocks can be merged in software. It will be shown in the section 5.1.2. that sorting in hardware (including data exchange with GPC) is faster that similar sorting in software. Merging larger blocks permits the time of sorting in software to be considerably reduced.

3.1.3. Sorting and merging with parallel data item counting

We propose a method of data sorting algorithm based on parallel sorting network with subsequent merge and data counting for the sorting acceleration and frequent item computation. The functionality of the merge units from the system described in the previous section is expanded by adding the operation of compressing the data by counting of the repeated data.

The circuit compares all the items in two sorted subsets of N data items and merges them into one sorted subsets. The maximal size of the final data set is 2×N items as in the system proposed in section 3.2.1. This worst case scenario can occur if no repeated items were found in both input subsets.

Although the maximal number of clock cycles for merging N-item blocks is 2×N, our system with compression and item counting requires less clock cycles for the data sets with repeated items. Every subsequent level of merging requires less clock cycles than the previous one, because the compression and counting was partially done in the previous level.

Modyfied fragment from Figure 3.6. is depicted on Figure 3.9 (a). The compression and the counting of the items is done in "compare and add" block shown in Figure 3.9(b). The system stores the data item which was written after the previous comparison and compares it with both inputs. If the item part of the item/count pair previously written to the RAM block is not equal to both of them, then the merger writes the item/count pair with larger item value to the output RAM block and increments both write address counter and read address counter for the input with the largest value. Otherwise, the merger does not increment the write address of the block and writes the new count number to the count part of the item/count pair. The new count number is the sum of the count parts of the previously written data item and the count of one of the inputs, which has an item part equal to the previously written one. During the first level of merging every pair has '1' as its count value. All zeros in the count part mean that the total number of repetitions exceeded the capabilities of the RAM block.

The RAM blocks of every item of the merging system are capable of storing all data from the inputs, but if the sorted set supplied to the merger contains repeated items, the system does not fill the RAM blocks completely. The merger reads the value from the write address register of the mergers from the previous level. It informs the merger about how many item pairs were actually written during the previous merge operation.

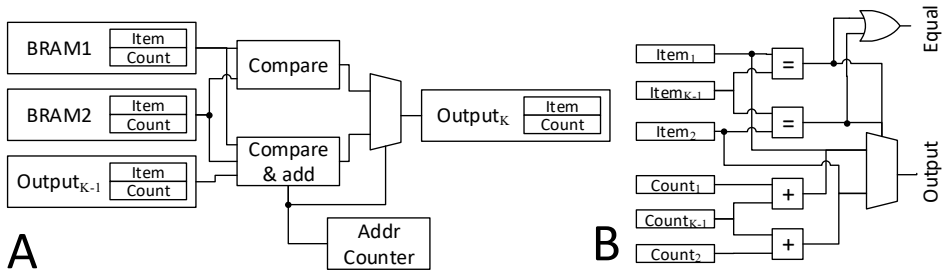

Figure 3.9 "Merge and count" architecture: A) General architecture of the merger B) Compare and add operation [119]

3.2. Partial sorting and minimum/maximum subset extraction

The network-based sorting circuit described above can be used efficiently for solving numerous supplementary tasks. One of these tasks is the extraction of the maximum and/or minimum subsets from the sorted sets. Also solving these tasks requires much less resources and therefore can use hardware more efficiently.

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. Since N may be very large (millions of items), the set cannot be completely processed in hardware because the resources required are not available.

We propose three different methods for finding minimum/maximum subsets. All these methods are based on sorting networks described in the previous chapter and perform partial sorting. At first we describe how to use these methods for simultaneous calculation of maximum and minimum subsets. After that other tasks and additional functionality will be discussed. Figure 3.10 depicts generalized architecture for all methods.

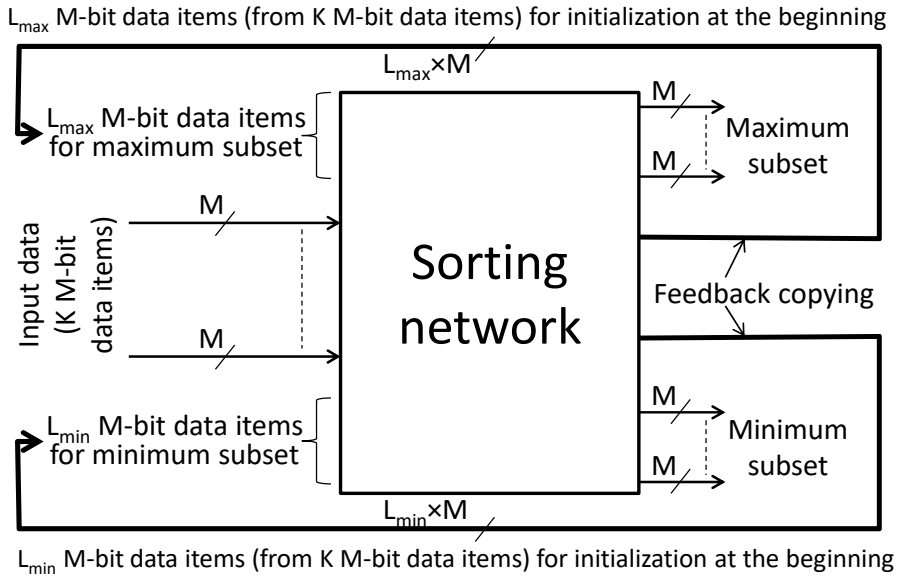

Figure 3.10 Computing the maximum and the minimum sorted subsets [121]

All methods are based on pipelined OETS network described above and designed for streaming data. The sorting unit receives the incoming data and outputs current minimum and maximum subsets every iteration. Data are incrementally received in blocks containing exactly K items and then processed by parallel networks described below. 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 for subsequent processing. The following steps describe how the system works with streaming data identical to all proposed methods:

1. The first block containing K M-bit data items is copied to input registers and becomes available at the inputs of sorting unit.

2. The block is sorted in parallel in the sorting unit with one of proposed methods.

3. L_{max} sorted items with maximum values become available on the outputs of the upper half of the sorting unit. L_{min} sorted items with minimum values become available on the outputs of the bottom half of the sorting unit.

4. A new block is copied to the input register 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 system block RAM and Q-1 blocks have been handled in the sorting unit, 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, the 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.

3.2.1. Method based on three sorting networks

The first method involves three sorting networks: one main sorting network (SN) and two additional sorting networss (SN_{min} and SN_{max}).

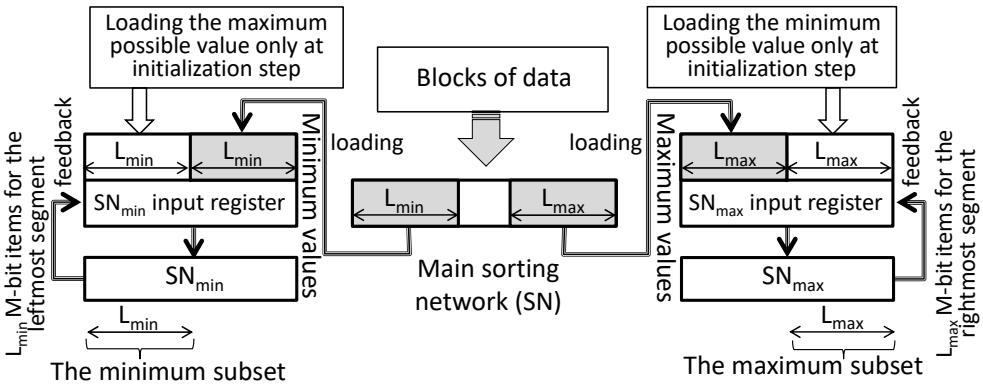

Figure 3.11 The first method of extracting the maximum and minimum sorted subsets [120]

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 Figure 3.11. At initialization step, these parts are pre-loaded with possible maximum and minimum values which data from the source set may have. 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 [19] (such as even-odd merger) or in sequential iterative networks from [36] (such as iterative OETS). 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 Figure 3.11. L_{min} sorted items with minimum

values are loaded in a half of the SN_{min} input register as it is shown in Figure 3.11. 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 blocks are handled.

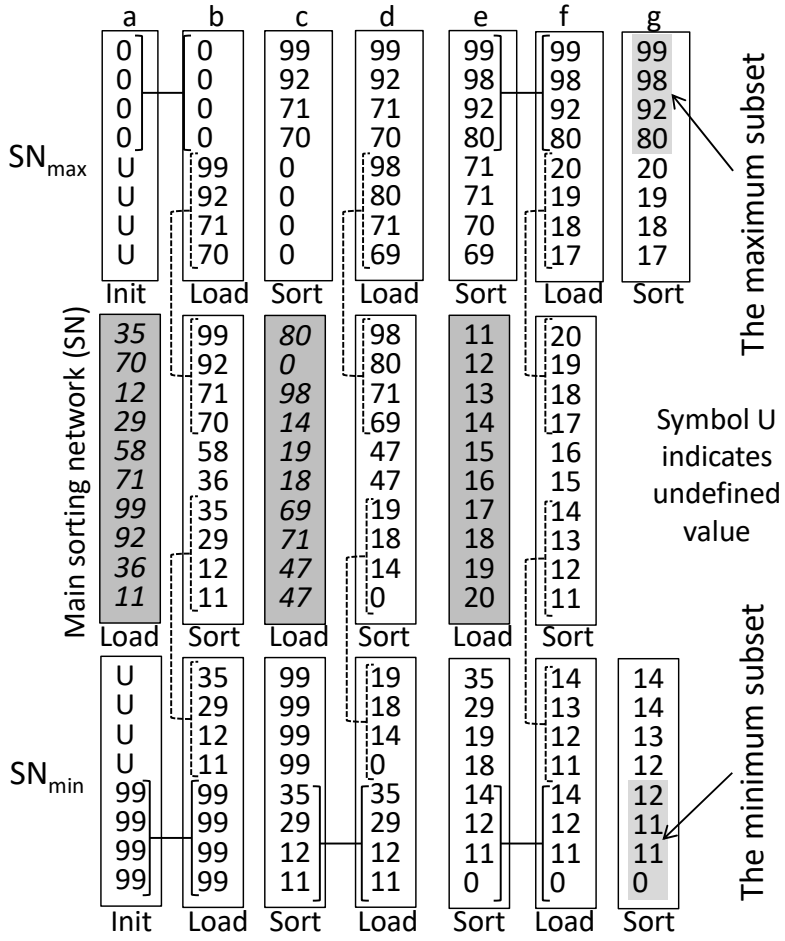

Figure 3.12 Example of extracting sorted subsets using the first method [120]

Figure 3.12 shows an example, assuming 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 Figure 3.12, 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 Figure 3.12 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 Figure 3.12) 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.

3.2.2. Method based on swapping networks

In the second method we use the circuits introduced in [57]. They are also composed of comparators/swappers explained in [4]. 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 [57] 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 the two sorted subsets and the lower half outputs of the network contain the smallest values from the two sorted subsets. Additionally, the outputs of this circuits form two Bitonic sequences. The swapping network depicted in Figure 3.13 transforms sorted sequences A and B to Bitonic sequences A and B, where all elements of Bitonic sequence A are larger than all elements in Bitonic sequence B.

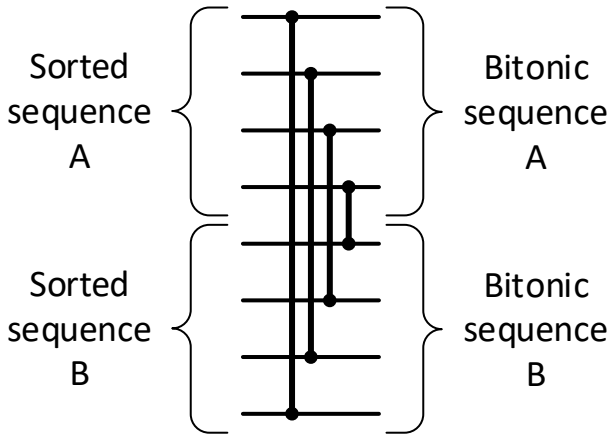

Figure 3.13 Swapping network

The idea of the second method is illustrated in Figure 3.14 on the same example from Figure 3.12.

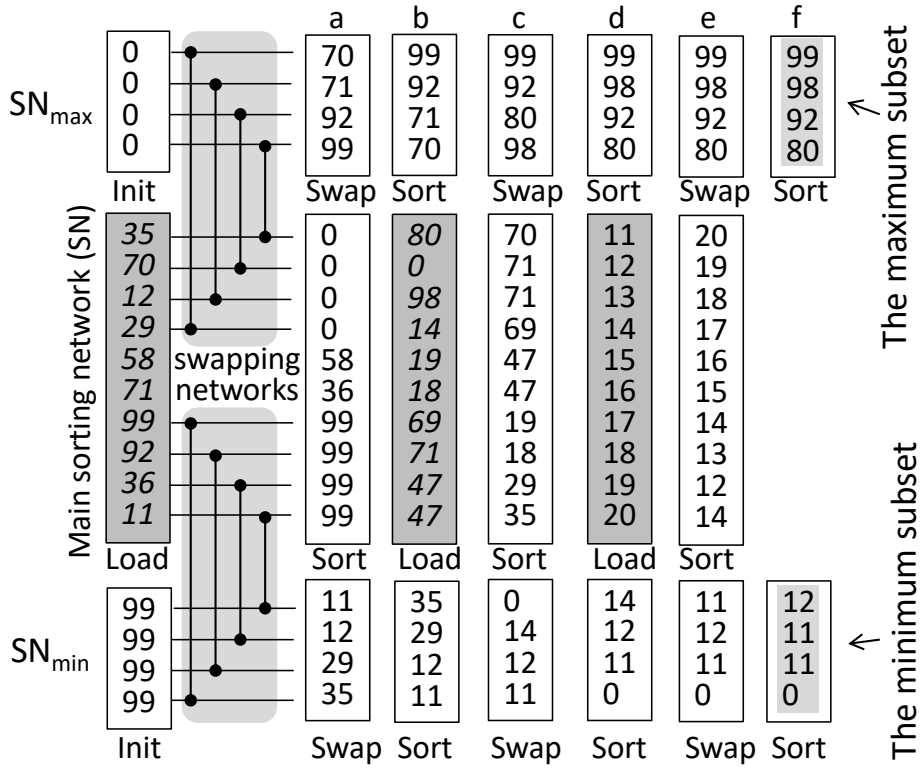

Figure 3.14 Example of extracting sorted subsets using the second method [120]

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 Figure 3.12). Much like Figure 3.12 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 Figure 3.14 which contain swapping networks described above. 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 Figure 3.14 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 Figure 3.12 The lower swapping network in Figure 3.14 functions similarly.

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

Also in some practical applications receiving sorted maximal and minimal subsets is not required and only unsorted ones are needed. In that case we can turn the second sorting network off during the last iteration of the algorithm.

3.2.3. Method based on single sorting network

The third method is similar to the first one, but instead of three independent sorting networks, it has only one network and is based on breaking links between comparators.

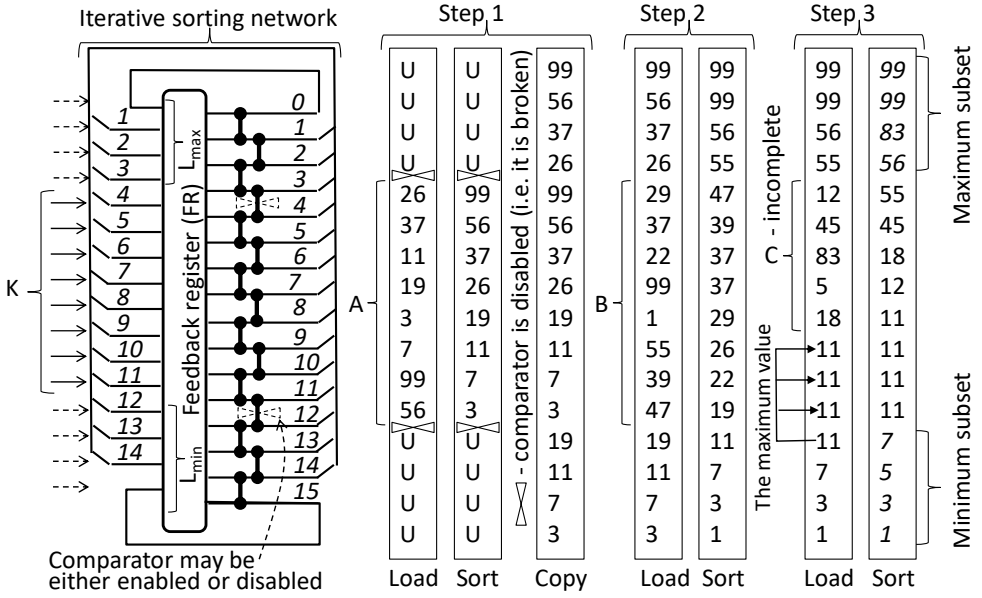

Figure 3.15 An example of sorting using the method based on single sorting network. [121]

At the first step, the first K M-bit data items are sorted in the network [36] which processes $L_{max}+K+L_{min}$ data items but comparators linking the upper part (handling L_{max} M-bit data items) and the lower part (handling L_{min} M-bit data items) are deactivated (i.e. the links with the upper and bottom parts are broken). So, sorting is done only in the middle part handling K M-bit items. As soon as the sorting is completed, the maximum subset is copied to the upper part of the network and the minimum subset is copied to the lower part of the network.

From the second step, all the comparators are properly linked, i.e. the network from [29] handles $L_{max}+K+L_{min}$ items, but the feedback copying (see the first

step and Figure 3.15) is disabled. Now for each new K M-bit items the maximum and the minimum sorted subsets are appropriately corrected, i.e. new items may be appended.

Let us look at the example shown in Figure 3.15 for which: N = 21, K = 8, L_{max} = L_{min} =4, and S = 26,37,11,19,3,7,99,56,29,37,22,99,1,55,39,47,12,45,83,5,18. The set S is divided into the following three subsets: A = 26,37,11,19,3,7,99,56, B = 29,37,22,99,1,55,39,47, and C = 12,45,83,5,18.

Note that the last subset C contains only 5 elements and is incomplete. Symbol U in Figure 3.15 indicates undefined value. The iterative sorting network is exactly the same as in [36]. There are 3 steps in Figure 3.15. At the first step, K (K=8) items are sorted and copied to the maximum and minimum subsets.

Two comparators are disabled in accordance with the explanations given above (breaking links of the middle section in the sorted network with the upper and the lower sections). At the second step, all the network comparators are enabled and L_{max}+K+L_{min} items are sorted by the iterative network with feedback register (FR). All necessary details can be found in [36]. It is easy to show that the maximum number of iterations is $\lceil(\max(L_{max},L_{min})+K)/2\rceil$ and much like the previous case this number is almost always smaller [36]. At the last (third) step, the incomplete subset C is extended to K items by copying the maximum value (11) from the minimum subset 11,7,3,1 to the positions of missing data (see Figure 3.15). After sorting L_{max}+K+L_{min} items at the step 3 the final result is produced.

3.2.4. Separate maximum and minimum extraction

Some practical applications don't require maximal and minimal subsets simultaneously. For this purpose a reduced partial sorter that contains one main and one additional sorting network was proposed.

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) of methods described above. Figure 3.16 depicts first two methods reduced only for maximum extraction.

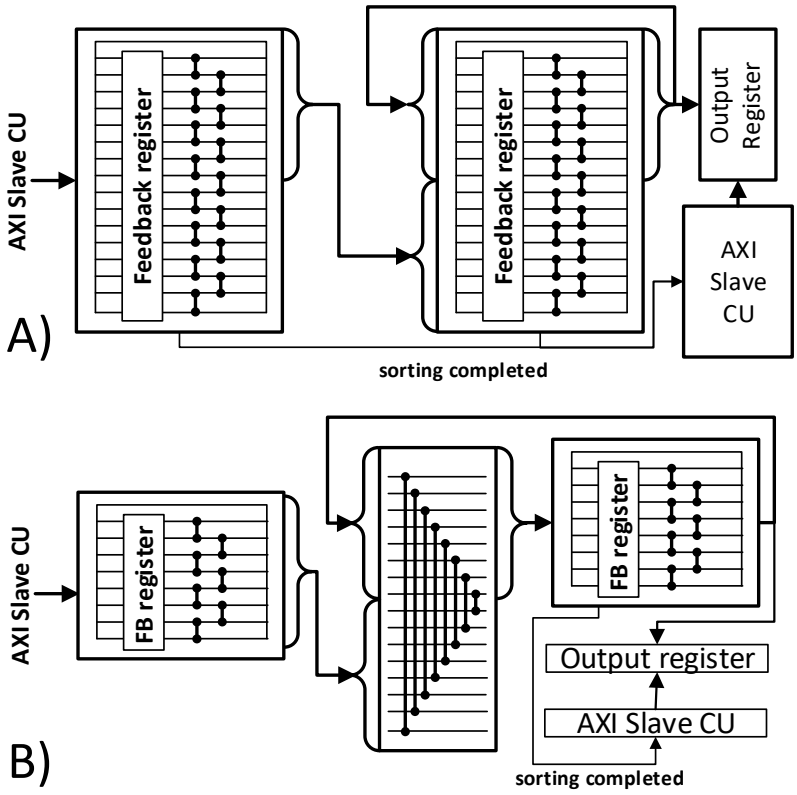

Figure 3.16 Sorters for extraction of maximal subsets [122]

Method A in Figure 3.16 is a reduced version of the method based of three sorting networks. This version of the partial sorter utilizes two sorting networks of the same size. The first sorting network receives blocks of data and sorts them. After the sorting is completed, the maximal (or minimal) half loads into the second sorting network along with maximal (or minimal) half of outputs of the second network. For maximal set selection, in the initial step the second network is loaded with zeros. For minimal set selection, it is loaded with maximal possible value. After all the data is transmitted, the system waits for the completion of sorting in both sorting networks. The maximal (or minimal) half of the outputs of the second network is loaded in the output register and waits for read request.

Method B in Figure 3.16 is a reduced version of the method based on swapping networks. This method doesn't require sorting minimal or maximal subset of the current iteration with results of the previous iteration. That is why sorting networks can be reduced twice. Both networks are connected here with a swapping network. All outputs of the first sorting network are connected to the swapping network along with all outputs of the second sorting network. On the outputs of the second network we receive unsorted maximal and minimum

subsets of the input data, where all items of the upper half of the network are larger than all items of the lower half.

The second method obviously requires less hardware results than the first method and can be combined with partial Bitonic sorter because of utilizing the swapping network. Although both methods are more or less equivalent for extracting both minimal and maximal subsets at the same time, the second method should be more suitable for separate extracting.

3.2.5. Very large scale subsets extraction

For some practical applications the maximum and minimum subsets may be large and the available hardware resources become insufficient to implement sorting networks. 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 of the sorting network, i.e. circuits 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 CPU. The software part 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 software maximum subset is assigned to B_u. The maximum value from the corrected in software minimum subset is assigned to B_l. The values B_u and B_l are supplied to hardware.

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 software) 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 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 [125]. In Chapter 5 we will present the results of experiments for such sorters.

3.2.6. Filtering

Input data may optionally be filtered allowing only items that fall within pre-given constraints to be processed. Let B_u and B_l be predefined upper (B_u) and lower (B_l) bounds for the given set S. We would like to use one of the circuits described above only for such data items D that fall within the bounds B_u and B_l, i.e. $B_l \le D \le B_u$ (or, possibly, $B_l < D < B_u$). Figure 3.17 depicts the proposed architecture that enables data items to be filtered at run-time (i.e. during the data exchange between hardware and software). There is an additional block on the upper input of the MUX, which takes a data item I_k and executes the operation indicated on the right-hand part of Figure 3.17. If the counter is incremented, then a new register is chosen to store data item I_k. Otherwise, the signal WE (write enable) is passive and a new item with a value that is out of the bounds B_u and B_l is not recorded in the registers.

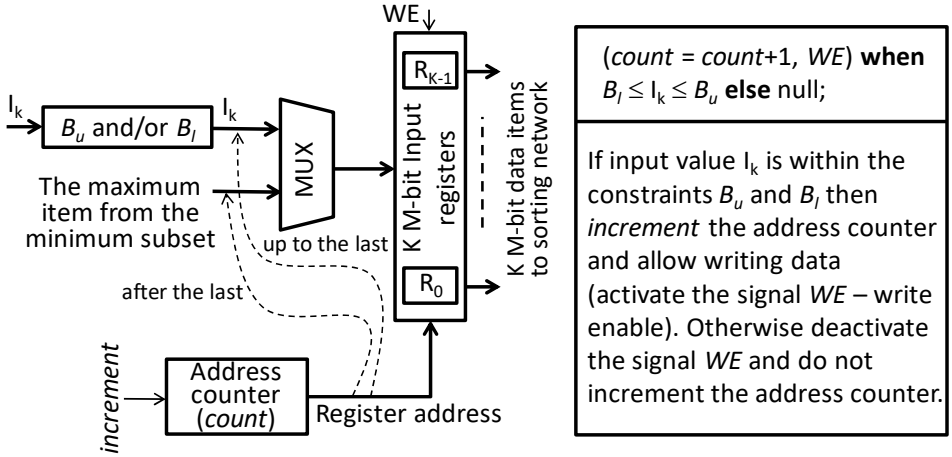

Figure 3.17 Digital filter [121]

Let us look at the same example in Figure 3.15 for which we choose $B_u = 90$ and $B_l = 10$. At the first step incoming data items have preliminary been filtered, the values 99, 7, and 3 have been removed (because they are either greater than $B_u = 90$ or less than $B_l = 10$), and the subset A with 8 items is built from 11 first elements of the set S. At the second (last) step, the values 99, 1, and 5 have been removed, and the subset B = 55,39,47,12,45,83,18 is built from the remaining allowed elements of the set S. Since there are 7 items in B and K = 8, this subset is incomplete.

3.3. Hamming Weight

We propose Hamming Weight counting circuit based on network of FPGA lookup tables (LUTs). An FPGA LUT(n,m) can be used to directly implement arbitrary Boolean functions $f_0,...,f_{m-1}$ of n variables $x_0,...,x_{n-1}$. Clearly, h LUTs(n,m) can be configured to calculate the Hamming weight w(A) of a vector $A=\{a_0,...,a_{n-1}\}$, where $h=\lceil(\log_2(n+1))/m\rceil$. The idea is to build a network from LUTs(n,m) that can find the Hamming weight w(A) for an arbitrary vector A of size N and then to compare this weight with either a fixed threshold κ or with the weight of another binary vector B assuming that the Hamming weight of B has been found similarly. Since Hamming distance d(A,B) = w(A XOR B) we can find d(A,B) as Hamming weights of "XORed" arguments A and B.

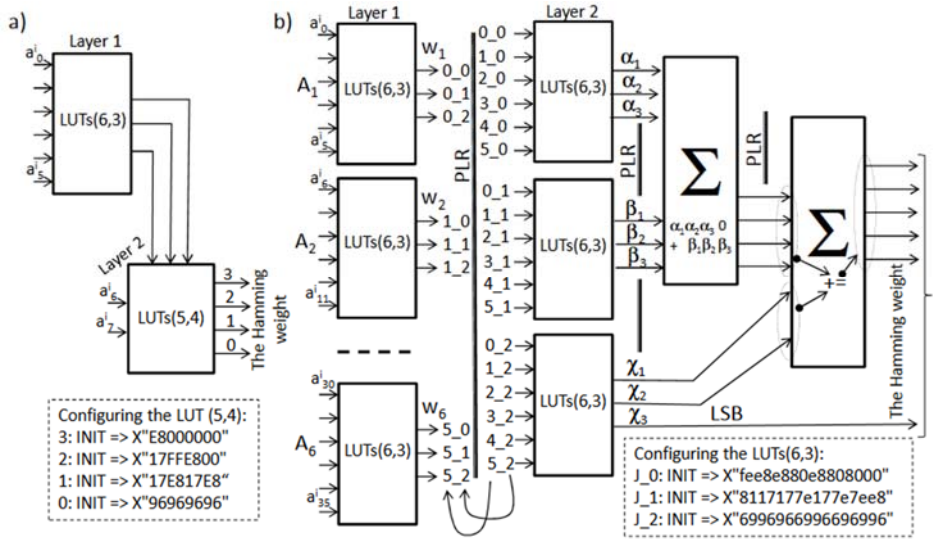

Figure 3.18 Hamming weight counters for N=8 (a) and N=36 (b) [123]

An analysis of practical applications shows that the majority of them require the Hamming weight/distance count/comparison for such values of N that are divisible by 8, 32, or 36. We suggest two optimized LUT-based designs permitting the Hamming weight to be found for N=8 (Figure 3.18(a)) and N=36 (Figure 3.18(b)). For N=32 either four bits in Figure 3.18(b) are assigned to 0 or the results of Figure 3.18(a) are incrementally added in a tree-based structure much similar to [89] composed of the design in Figure 3.18(a) and adders. The circuit in Figure 3.18(b) without two right adders Σ has $\lceil(\log_2(n+1))/m\rceil\times(\lceil N/n\rceil+\lceil (N/2)/n\rceil)$ LUTs(n,m). Even for m=1 (the worst case) we need only 27 LUTs for Zynq xc7z020 containing totally 53200 LUTs.

The Hamming weight for N>36 can be found in a similar tree-based structure. There are two layers in Figure 3.18(a) with LUTs(6,3) and LUTs(5,4). The

first layer counts $w(a^i_0,...,a^i_5)$ and the second layer takes the results of the first layer and finally determines the 4-bit weight $w(a^i_0,...,a^i_7)$. The delay from the inputs to the outputs is equal to just 2 LUT delays. There are also two layers in Figure 3.18(b) with LUTs(6,3) and two combinational adders. The first layer is composed of 6 LUTs(6,3) and it outputs six Hamming weights $w_1,...,w_6$ for six sub-vectors A1,...,A6 of the input vector. The second layer contains 3 LUTs(6,3) and it outputs Hamming weights $\alpha_1\alpha_2\alpha_3$, $\beta_1\beta_2\beta_3$, $\chi_1\chi_2\chi_3$ of the most significant bits (MSB) in w1,...,w6 ($\alpha_1\alpha_2\alpha_3$), the middle bits in w1,...,w6 ($\beta_1\beta_2\beta_3$) and the less significant bits (LSB) in w1,...,w6 ($\chi_1\chi_2\chi_3$). The final result is computed by two combinational adders as it is shown in Figure 3.18(b). We found that any layer with index greater than $\lceil \log_n N \rceil$ is not cost-effective because either the size of output weights will be increased compared to the previous layers or LUTs will be used not-efficiently. All LUTs in Figure 3.18(b) are configured identically.

3.4. Matrix covering

We have studied combinatorial search problems that utilize Hamming weight calculating and sorting and one of them is matrix covering problem.

The covering problem can identically be formulated on either sets [67], [126] or matrices [67]. Let $A = (a_{ij})$ be a 0-1 incidence matrix. The sub-set $Ai = \{j \mid a_{ij} = 1\}$ contains all columns covered by row i (i.e. the row i has value 1 in all columns of the sub-set Ai). The minimal row cover is composed of the minimal number of the sub-sets Ai that cover all the matrix columns. Clearly, for such sub-sets there is at least one value 1 in each column of the matrix. Let us consider an example from [2] of a set S and sub-sets $S_1,\ldots,S_6$ (Figure 3.19), which can be represented in the form of the following matrix A:

	1	2	3	4	5	6	7	8	9	10	11	12
S_1:	1	1	0	0	1	1	0	0	1	1	0	0
S_2:	0	0	0	0	0	1	1	0	0	1	1	0
S_3:	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
S_4:	*0*	*0*	*1*	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*
S_5:	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*
S_6:	0	0	0	1	0	0	0	1	0	0	0	0

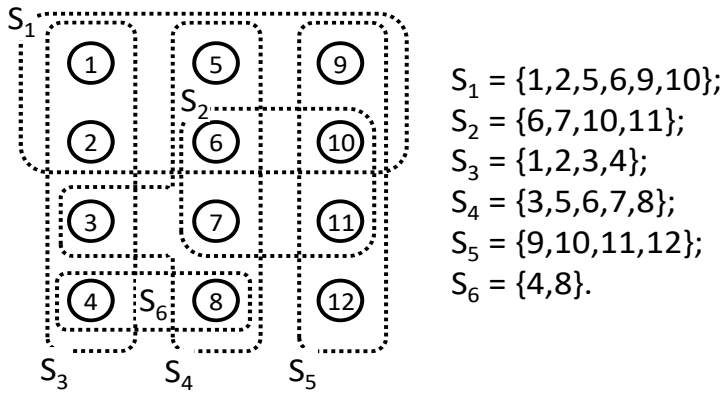

Figure 3.19 An example of a set S with sub-sets $S_1,...,S_6$ from [124]

We consider below a slightly modified method from [127] that is applied to binary matrices exemplified above and the matrix from Figure 3.19 [126] will be used to illustrate the steps of the chosen method that are the following:

1. Finding the column C_{min} with the minimum Hamming weight (HW) that is the number of ones. If there are many columns with the same (minimum) HW, selecting such one for which the maximum row is larger, where the maximum row contains 1 in the considered column and the maximum number of ones;
2. If HW = 0 then the desired covering does not exist, otherwise from the set of rows containing ones in the column C_{min} finding and including in the covering the row R_{max} with the maximum HW;
3. Removing the row R_{max} and all the columns from the matrix that contain ones in the row R_{max}. If there are no columns then the covering is found otherwise go to the step 1.

Let us apply the step 1–3 to the matrix A above:

1. The column 12 is chosen;

2. The row S5 is included in the covering;

3. The row S5 and the columns 9, 10, 11, 12 are removed from the matrix.

1. The remaining columns contain the following number of ones: 2, 2, 2, 2, 2, 3, 2, 2. The column 3 is chosen because for this column the row S4 has the maximum HW equal to 5;

2. The row S4 is chosen and included in the covering;

3. The row S4 and the columns 3, 5, 6, 7, 8 are removed from the matrix.

1. The remaining matrix contains rows S1, S2, S3, S6 and columns 1, 2, 4 with the following HWs: 2, 2, 2. The column 1 is chosen;

2. The row S3 is chosen and included in the covering;

3. After removing the row S3 the covering is found and it includes the rows S3, S4, S5 shown in italic font in the matrix above. The minimum covering is the same as in [126] that was found with a different algorithm.

We suggest the given matrix to be unrolled in such a way that all its rows and columns are saved in hardware (in programmable logic of FPGA of PSoC) registers. Note that more than a hundred of thousands of such registers are available in the recent low-cost FPGAs. This technique permits all rows and columns to be accessed and processed in parallel.

Figure 3.20 demonstrates the unrolled matrix **A** shown above (and repeated in Figure 3.20 for convenience). HW counters compute HW for all the rows/columns in parallel using combinational circuits, such as that are proposed in [36].

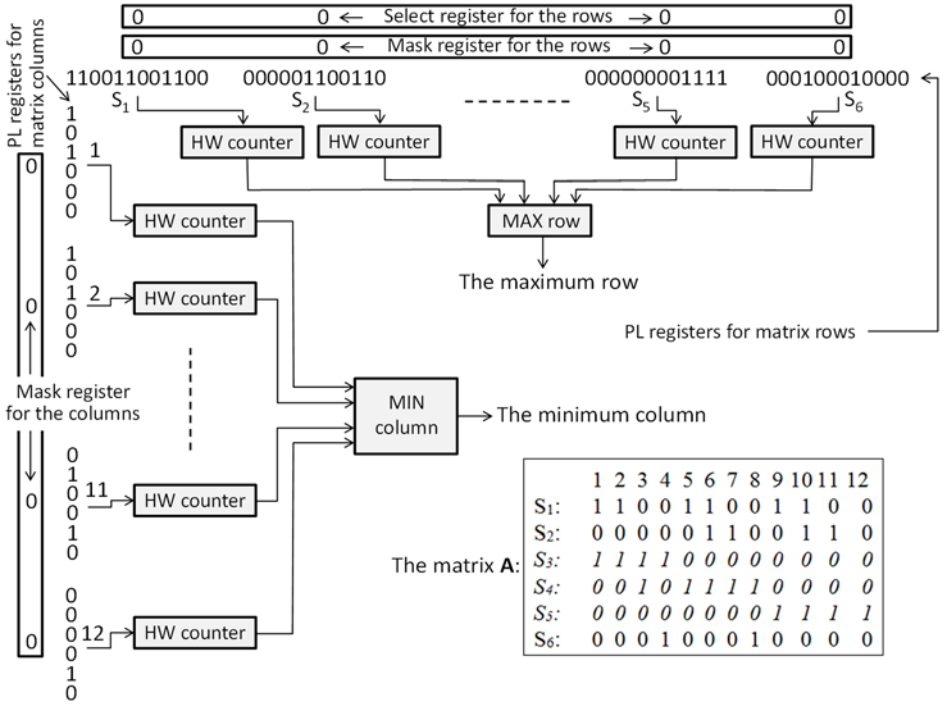

Figure 3.20 Architecture of the proposed hardware accelerator on an example of unrolled matrix [124]

The MIN column and MAX row circuits permit to find out the minimal column C_{min} and the maximum row R_{max}. It is shown in [128] that these circuits can be built as MAX-MIN fully combinational networks producing the results faster than in 20 ns. Since all the circuits (computing HW and the maximum/minimum values) are functioning in parallel, the steps 1 and 2 may be completed faster than in 20 + 20 = 40 ns even in low-cost FPGAs. So, a very significant acceleration can be expected.

Figure 3.21(a) presents such a circuit for a matrix 32 × 32 for which the number of bits in any HW is 6 (because the maximum number of ones in a 32-bit vector is 32 that can be represented by a 6-bit code). A particular (simplified) example for only 6 input items 3, 14, 21, 11, 14, 27 is given in Figure 3.21(b). The maximum value (27) is found in a combinational circuit with only 3 gate level delays. Clearly, there is 5 gate level delay for matrices 32 × 32 and 6 gate level delay for matrices 64 × 64.

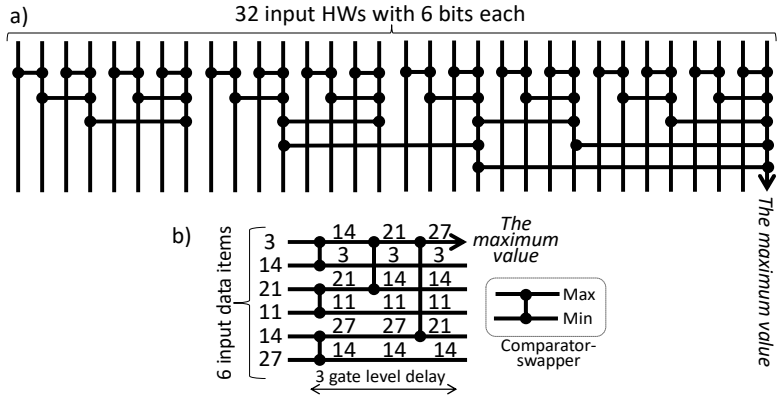

Figure 3.21 MAX circuit from [8] for 32 × 32 matrix (a); an example (b). [124]

Since all the circuits (computing HW and the maximum/minimum values) are functioning in parallel, the steps 1 and 2 may be completed faster than in 20 + 20 = 40 ns even in low-cost FPGAs. So, a very significant acceleration can be expected.

In accordance with the proposals, the matrix is unrolled only once and any reduced matrix is formed by masking previously selected rows and columns. One select register and two mask registers (one for rows and another one for columns) shown in Figure 3.20 are additionally allocated in the PL. The select register is zero-filled at the beginning of the step 1 and after the step 1 it indicates by values 1 those rows that have to be chosen by the selected column (i.e. such rows have values 1 in the selected column). The mask registers are filled in with zeroes at the beginning of the algorithm and they mask (by the values 1) those rows and columns that have been removed from the matrix in each iteration. For example, the select register contains the value 000010 after the first step in the example matrix A. The mask registers after the first iteration in the example are set to 000000001111 for the columns and 000010 for the rows. After the second iteration they are updated as 001011111111 for the columns and 000110 for the rows.

3.5. Summary

This chapter describes methods proposed in this research. It presents the pipelined periodic sorting network – which serves as a basis of our data sorting solutions. The advantage over other sorting networks were discussed and possible drawbacks were stated.

We proposed two approaches of full data sorters with different combination of sorting networks and merging of sorted subsets. The first proposed method of data sorting involves parallel sorting of data fragments in hardware with subsequent merging of those fragments in software. The second method also suggest a combination of hardware and software components, but hardware in this case performs both network-based sorting and merge operation based on a tree-like structure of block RAM-based mergers. The second method relies on sorting smaller data fragments than the first method, but hardware merging allows larger sorted data fragments supplying for subsequent data merging.

Partial sorting methods for maximal and minimal subset extraction were also described in this chapter. We proposed three different methods for this problem solving and discussed advantages of each of them. The first method is based on main and additional sorting networks with copying data between them. The second method is based on swapping network, which permits avoiding additional data copying and reducing the size of additional sorting networks. The third method is based on switchable C/S block which permits using a single sorting network and less C/S blocks. The possibility of filtering and extracting very large scale subsets, which allow to go beyond hardware limitation, was also discussed.

The architecture of FPGA LUT-based circuit for Hamming weight calculation was presented. The hardware system for matrix covering also described in this chapter utilizes this circuit along with a combinational network composed of C/S blocks for maximal and minimal item extraction.

Also we proposed hardware-based system for matrix covering, which utilizes LUT-based Hamming weight calculators and comparator networks.

4. HARDWARE/SOFTWARE CO-DESIGN

The known results [129] [130] [131] have shown that software/hardware solutions may be significantly faster than software only solutions. Hardware only solutions are the fastest, but they are not suitable for the majority of practical application, because of the resource limitation. We explored different platforms for hardware/software co-design.

In this chapter three different approaches are explained: processing system and programmable logic combination on PSoC [120] [121], FPGA/PC combination and a three-level system which combines programmable logic and processing system of PSoC with a host PC [5] [118] [119]. Also we describe hardware architectures based on these approaches for methods proposed in the previous chapter.

4.1. PS/PL system

This chapter describes our approach of hardware software co-design for PSoC architecture from the Zynq-7000 family. It is an architecture that combines the dual-core ARM® Cortex™ MPCore™-based processing system and Xilinx programmable logic on the same microchip. There are similar solution from other FPGA manufacturers [132] [133] [134], but we focus on Xilinx platforms.

Figure 4.1 illustrates interactions between the basic functional components of the Zynq-7000 PSoC [135] that contains two major top-level blocks: the processing system (PS) and the programmable logic (PL). Communications with external devices are provided through multiplexed input/outputs (MIO) with potential extension from the PL through extended MIO (EMIO). Zynq PSoCs offer numerous communication mechanisms from simpler general-purpose input to more advanced data exchange through AXI interfaces allowing access to external DDR memory, to on-chip memory (OCM) and to level 2 cache of PS.

Software and hardware can be designed autonomously and linked in a hardware/software system. To increase performance, the most time-consuming parts of software might be redesigned and implemented in hardware.

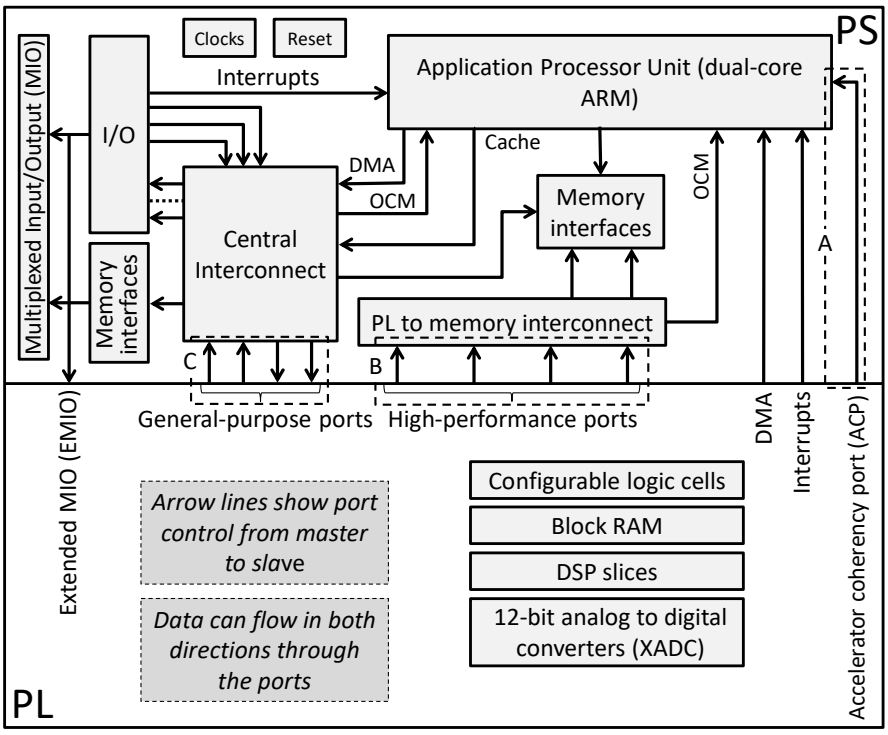

Figure 4.1 Interactions between the basic functional components of the Zynq-7000 PSoC [125]

Let us look at Figure 4.2. 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, $T_{sch} \leq T_{sh} + T_h + T_c$. For instance we would like communication and application-specific operations to be overlapped in hardware as much as possible (see Figure 4.2). 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.

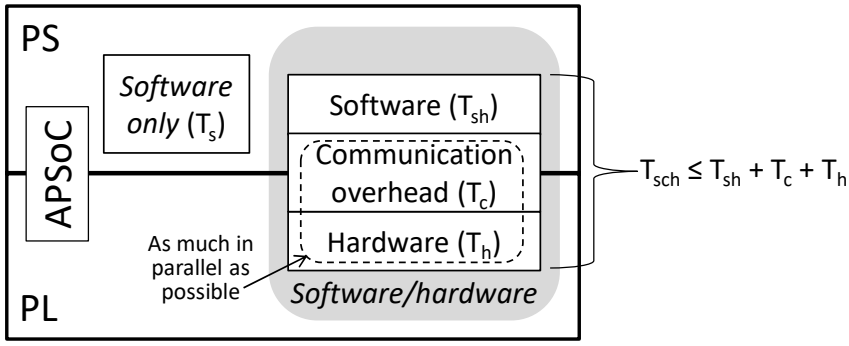

Figure 4.2 Software only and software/hardware systems [125]

Fig. 4.3 presents the proposed software/hardware architecture for the problems discussed in Chapter 3. All hardware acceleration is done in an application-specific processing block (ASP) which is entirely implemented in the PL. 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 the PL which is done through a set of control signals.
3. Collecting data and performing tasks in software.
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. 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. Performing computations 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.

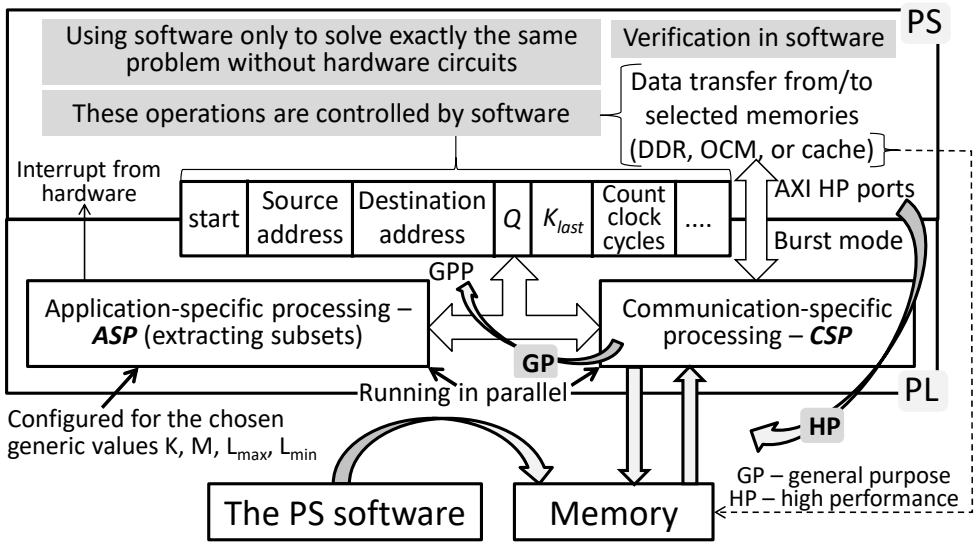

Figure 4.3 The proposed software/hardware architecture for data sorting [125]

Selection of proper AXI ports is very important. Experiments in [136] 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 PSoC 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 [137]. Since the latter feature is what we need, a master GPP was chosen for transferring control signals shown in Figure 4.3. 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) [136]. Mapping of memories may be done in computer-aided design software. Experiments in [136] [131] 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.

Figure 4.4 gives more details about the chosen software/hardware interactions where: solid arrows ($\rightarrow$) indicate who is the master (the beginning) and who is the slave (the end); double compound lines show control flow; and dashed lines indicate directions of data flow (i.e. one direction - $\rightarrow$ or both directions - $\leftrightarrow$). 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 and in software.

The snoop controller [135] in Figure 4.4 provides cacheable and non-cacheable access to memories (OCM or DDR). Cache area can be either disabled or enabled in software. 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 [131].

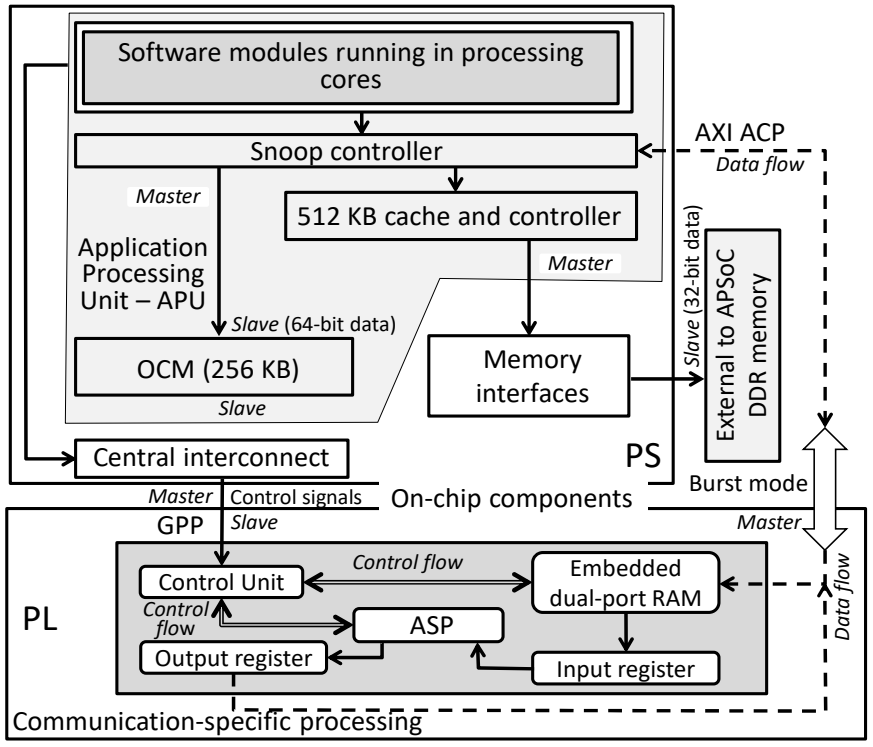

Figure 4.4 Hardware/software interactions [120]

4.1.1. Hardware/software co-design for subset extraction

This section presents the hardware architecture for methods of subsets extractors from section 3.2 based on proposed hardware/software approach.

Initial (source) data set and extracted subsets are accommodated in memory as it is shown in Figure 4.5. All necessary details about particular locations and sizes are supplied from the PS to PL through GPP (see Figure 4.4).

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 the source and destination addresses. Here, N is the total number of data items that have to be processed.
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).
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 previous chapter 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 Figure 4.6. 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 are given in section 3.2).
8. The maximum and minimum subsets are copied from the PL to memory (see Figure 4.5).
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.
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 an 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.

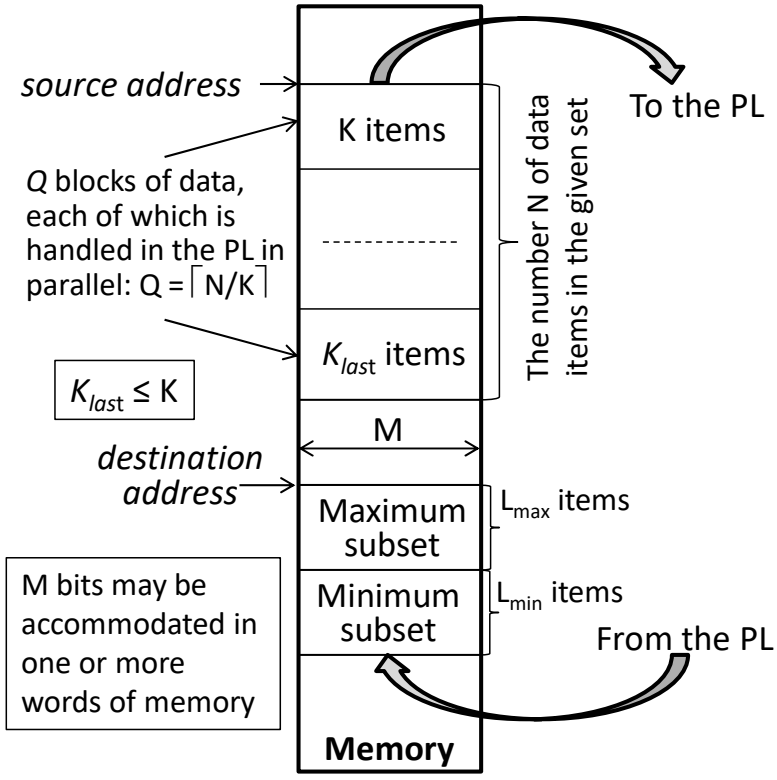

Figure 4.5. Accommodation of the initial data set and the extracted subsets in memory [120]

The circular buffer in Figure 4.6 is managed by the PL control unit 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.

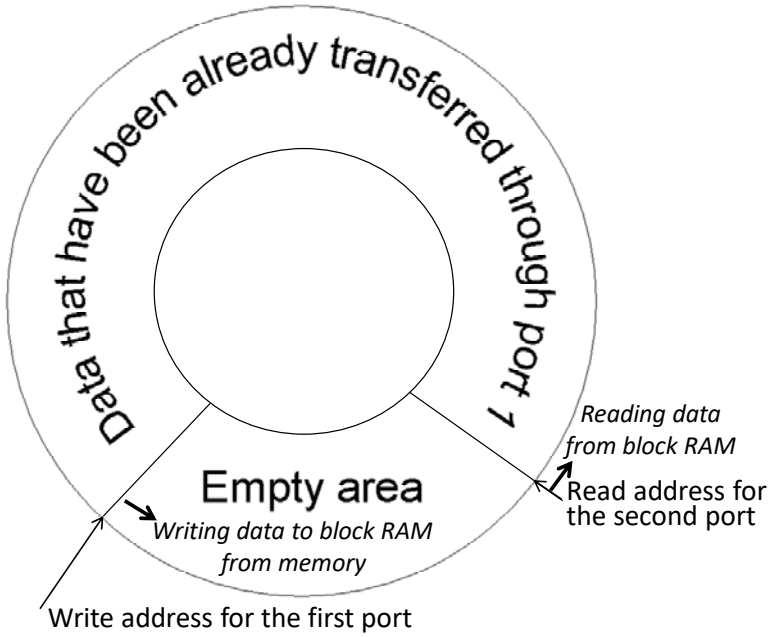

Figure 4.6 Block RAM organized as a circular buffer [120]

4.1.2. Hardware/software system for search problems

Figure 4.7 presents the proposed partitioning in software and hardware modules (assuming implementation in Zynq PSoC) of the considered algorithm that enables the minimal covering to be found.

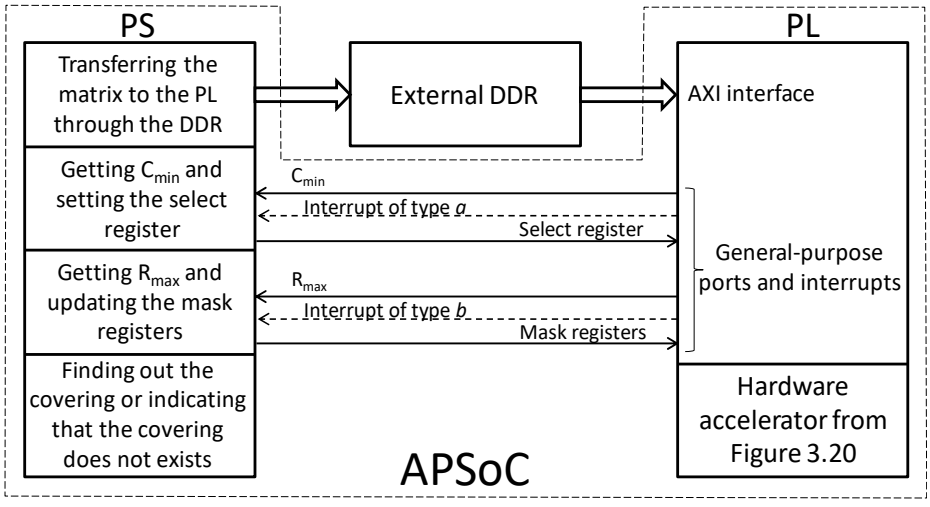

Figure 4.7. Partitioning of the algorithm in software and hardware modules [124]

Software in the PS is responsible for the following steps:

1. Getting from a host computer or generating the matrix, unrolling it, and saving in external DDR memory as a set of rows and a set of columns;
2. As soon as C_{min} is found, the PL generates an interrupt of type a. The PS receives the C_{min} and sets the select register in the PL through general-purpose ports;
3. As soon as R_{max} is found, the PL generates an interrupt of type b. The PS receives the R_{max} and sets the mask registers in the PL through general-purpose ports [135];
4. At any iteration it is checked if the solution is found or if it does not exist. If the solution is found it is indicated by the PS or transmitted to the host computer and the algorithm is completed.

Hardware in the PL implements the architecture in Figure 3.20 and is responsible for the following steps:

1. Getting the unrolled matrix from external DDR through high-performance AXI Interface and saving the rows and columns in slice registers as it is shown in Fig. 3.20.
2. Getting from the PS select/mask vectors and setting/updating the select and the mask registers.
3. Finding out the value C_{min} at each iteration and as soon as the value of C_{min} is ready, generating an interrupt of type a.
4. Finding out the value R_{max} at any iteration and as soon as the value of R_{max} is ready, generating an interrupt of type b.

4.2. FPGA-based system with host PC

Another platform that have been studied is a two-level system that combines general purpose PC for software part and FPGA for hardware acceleration. The data transfer between hardware and software levels is organized through PCI-express. The architecture of hardware accelerator part of this two-level architecture is shown in Figure 4.8. On-board DDR memory is used for preliminarily data collection by FPGA and storage. The DDR is controlled by a memory interface generator.

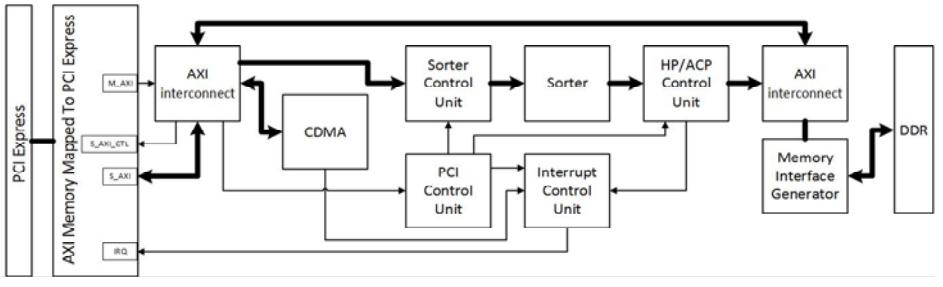

Figure 4.8 Architecture of a two-level FPGA-based system for data sorting [5]

Data transfer in the host PC is organized through direct memory access (DMA) module that was developed for PC and FPGA integration of the two-level system. The hardware uses the Intellectual Property (IP) core of the central direct memory access (CDMA) module [138] to copy data through AXI PCI express (AXI-PCIE) [139]. The project is similar to [140] and links CDMA and AXI-PCIE modules based on a simple data mover (i.e. the mode "scatter gather" is not used). A master port (M-AXI) of the AXI-PCIE operates similarly to GP ports in [136] and supplies control instructions from the PC to customize data transfers. The instructions indicate the physical address of data for PC memory, the size of transferred data, etc.

Software in the host PC runs the 32-bit Linux operating system (kernel 3.16) and executes programs (written in the C language) that take results from PCI-express (from the accelerator) for further processing. To support the data exchange between two parts of the system, a driver (kernel module) for general purpose PC was developed. The driver creates in the directory /dev a character device file that can be accessed through read and write functions, for example write(file, data_array, data_size). Up to 5 base address registers (BAR) can be allocated but we used just one.

The PC BIOS assigns a number (an address) to the selected BAR and a corresponding interrupt number that will be later used to indicate the completion of a data transfer. As soon as the driver is loaded, a special operation (probe) is activated and the availability of the device with the given identification number (ID) is verified (the ID is chosen during the customization of the AXI-PCIE). Then a sequence of additional steps is performed (see [141] for necessary details). A number of file operations are executed in addition to the probe function. In our particular case, access to the file is done through read/write operations. Figure 4.9 demonstrates the interaction of a user application with the driver (kernel module) and some additional operations that may be executed.

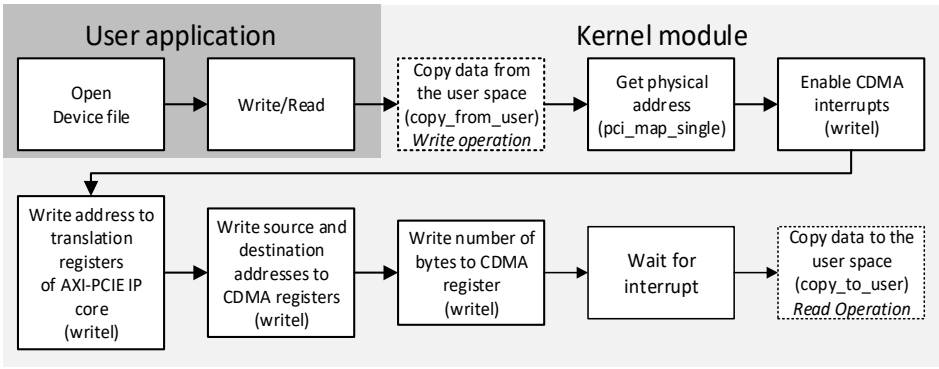

Figure 4.9 Kernel module [5]

As soon as a user program calls the read function, the read(file, data_array, data_size) function gets the address in the user memory space and the number of

bytes that need to be transferred. Initially, the data are copied to a buffer and then the physical address of the buffer is obtained. Now the data are ready to be transferred from PSoC/FPGA. Then the data are copied and the driver is waiting for an interrupt indicating that the data transmission is complete. The necessary operations for generating the interrupt are given in [138]. Additional details can be found in [141].

For the methods in sections 3.1-3.2 the proposed networks can be used as follows. The sorter receives blocks composed of N M-bit data items that are collected from inputs initially and stored in DDR memory. Interactions with memory are done through the memory interface block (see Figure 4.8). The sorter executes iterative operations over multiple parallel data and is controlled by a dedicated finite state machine (FSM) called Sorter Control Unit (see Figure 4.8). The ports are also controlled by a dedicated FSM (see HP/ACP Control Unit in Figure 4.8). The results of sorting are copied back to memory and then transmitted to the host PC through the PCI-express bus. Specially developed dedicated circuits are responsible for data collection and organization that is done in accordance with the established requirements. Finally, the dedicated circuits prepare data in memory so that these data can be processed in the FPGA and the results of the processing (stored in memory) are ready to be transmitted to the host PC. The blocks CDMA with control units (PCI Control Unit and Interrupt Control Unit in Figure 4.8) are responsible for transmitting data.

4.3. Three-level system

The third approach can be described as a combination of both system discussed previously: PS/PL system and a system with a host PC.

Certain Zynq devices (for example, Xilinx zc706) with support of PCI-Express interface allow us to build a three-level architecture which combines PS and PL of PSoC with general purpose PC. Figure 4.10 shows the basic architecture for data transfer between a host PC and an PSoC through PCI-express.

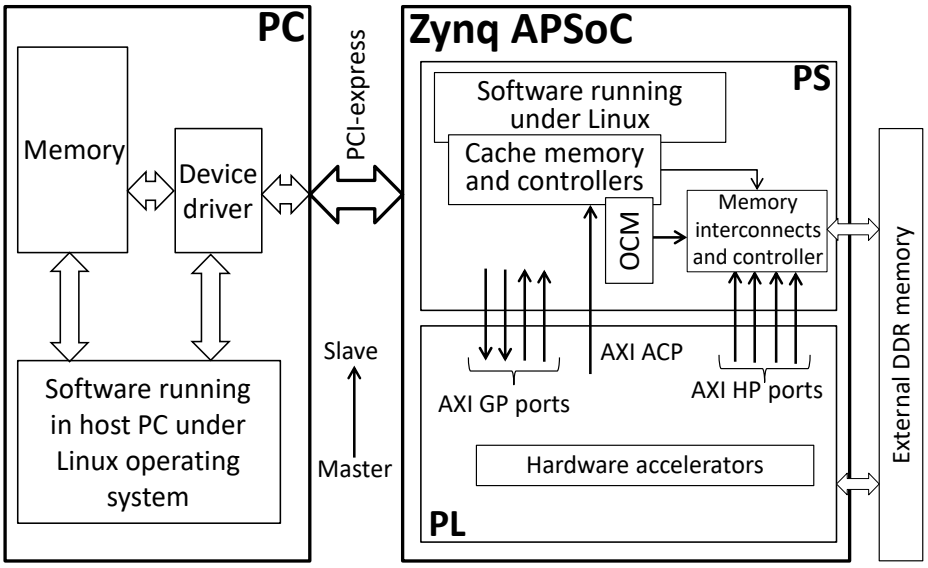

Figure 4.10 Basic architecture for data transfer between a host PC and a PSoC through PCI-express [5]

Figure 4.11 presents the architecture of hardware accelerator part of the three-level system for the example of distributed data sort. The architecture is based on the hardware accelerator from the previous section, but includes PS part of the PSoC. We assume that the data collected in the PSoC are preprocessed in the PSoC by applying various highly parallel circuits, and the results are transferred to the host PC through the PCI-express bus. The device driver for general purpose PC is similar to the one described in section 4.2. The CDMA module can be connected to either AXI HP or AXI ACP interfaces in PSoC and transmits data from either on-chip memory (OCM) or external DDR. After supplying the addresses, the number of data bytes (that need to be transferred) is indicated and the data transmission is started. As soon as data transmission is completed, the CDMA module triggers an interrupt that has to be properly handled (the interrupt number is determined by the BIOS of the host PC). The following customization is done for 1) AXI-PCIE: legacy interrupts, 128 bits data width, and 2) CDMA: 256 bytes burst size, 128 bits data width. Note that the architecture in Figure 4.11 allows data transfers in both directions, i.e. data from the PC may also be received.

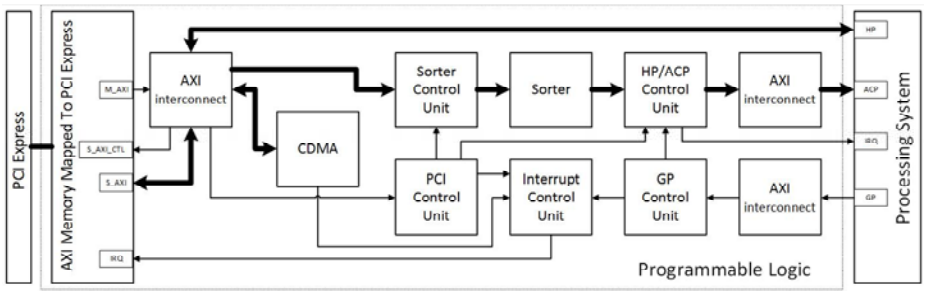

Figure 4.11 Architecture of a three-level PSoC-based system for data sorting [5]

The proposed architecture is similar to the architecture described in section 4.2. The main difference that in addition to two levels – PL (FPGA) and host PC system, there is another level – PS of the PSoC system. Interaction between the PL and PS are implemented as in PS/PL system described in section 4.1. but with inclusion of Zynq PS as in PS/PL system, The data processing logic (for example, sorter) receives blocks composed of N M-bit data items that are collected from inputs initially and stored in memories (such as external DDR and OCM). In case of a three-level system, transactions with memory are done through AXI HP/ACP ports of PS (see Figure 4.11) and not through the memory interface block (see Figure 4.8). Other steps of the method are also similar to the two-level system with host PC, but PSoC PS, instead of specially developed dedicated circuits, is responsible for data collection and organization that is done in accordance with the established requirements. The PS prepares the data so it can be processed in the PL and transmitted to the host PC.

4.4. Summary

This chapter proposes and describes different approaches of hardware/software co-design for reconfigurable FPGA and PSoC devices. The first approach involves usage of PS and PL of PSoC device with communication between them through AXI interface ports.

Also this chapter presents two-level and three-level approaches using a general purpose computer (host PC) and communication through high-speed PCI express interface. The first level of the two-level system is programmable logic (FPGA) and the second is host PC running Linux operating system. The three-level system was designed for PSoC devices and has an additional level which is PS of the PSoC device. Linux Kernel module was written for integration of both architectures.

This chapter covers all aspects of hardware/software implementations of methods proposed in the previous chapter. The proposed techniques and methods are suitable for processing large volumes of items and designed to work with streaming data. The architectures presented in this research can be easily utilized for different hardware accelerators.

5. EXPERIMENTS

This chapter describes experimental results of systems discussed in Chapter 3. All experimental setups are based on approaches presented in Chapter 4. The platforms and techniques were chosen according to requirements and features of the systems.

Experiments were done with different prototyping boards. The platforms for hardware/software systems that involve PL and PS of PSoC device were Xilinx ZC702 [142] and ZedBoard [143]. For the systems that involved data transfer between PC and accelerators, we used two boards. The first is the Xilinx ZC706 [144] evaluation board containing the Zynq-7000 XC7Z045 PSoC device with PCI express endpoint connectivity "Gen1 4-lane (x4)". The PS is the dual-core ARM Cortex-A9 and the PL is a Kintex-7 FPGA from the Xilinx 7th series. The second board is VC707 and it contains the Virtex-7 XC7VX485T FPGA from the Xilinx 7th series with PCI express endpoint connectivity "Gen2 8-lane (x8)" [145]. All designs were done for: 1) hardware in the PL of PSoC/FPGA synthesized from specifications in VHDL that describe circuits interacting with Xilinx IP cores (Xilinx Vivado Design Suite 2016.2); 2) software in the PS of PSoC developed in C language (Xilinx Software Development Kit – SDK 2015.1); 3) user programs developed in C running under the Linux operating system in the host PC. The PL clock frequency is 125 MHz. The PS frequency is 666 MHz. Data were transferred from the ZC706/VC707 to the host PC through PCI-express. The PCI-express bus frequency is 100 MHz. The host PC contains Intel core i7 3820 3.60 GHz.

5.1. Data sorting

We have implemented different hardware/software systems based on methods described in Chapter 3. All the proposed approaches were implemented and tested on FPGA and PSoC platforms. Experiments were conducted for hardware sorting with subsequent software merge and for hardware sorting and merging with subsequent software merging for very large subsets.

5.1.1. Hardware sorting of data subsets and software merging

Hardware/software system based on hardware sorting network-based sorters with subsequent software merge was implemented using two different approaches: GPP+FPGA and three-level system described in sections 4.2 and 4.3. Figure 5.1 demonstrates organization of experiments with data sorters for the three-level system.

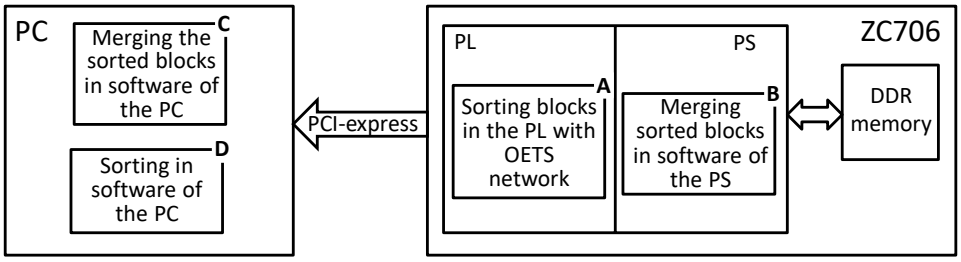

Figure 5.1 Organization of experiments with data sorters (the size of one block is 1024 32-bit data items) [5]

We assume that data are collected by the ZC706/VC707 board and stored in DDR memory (in the experiments, data are produced as described in point 1 below). Subsequently, different components (A, B, C, D) may be involved in data processing:

1. Data are randomly generated and sorted using only networks in hardware (component A), indicated below as *Sorting blocks*;
2. Data are generated and sorted in the PC, indicated below as *PC sort*.
3. Data are transferred from the ZC706/VC707 to the PC through PCI-express and sorted by software in the PC (component D), indicated below as *PC sort+data transfer*;
4. Data are completely sorted in the PSoC (the set of data items is decomposed into blocks, blocks are sorted in the PL by the networks described in section 3.1 , the sorted blocks are merged in the PS to produce the final result) and the sorted data are transferred to the PC through PCI-express (components A and B), indicated below as *Sorting+PS merge*;
5. Data are completely sorted in PSoC/FPGA and in the PC in such a way that: a) blocks of data are sorted in the PL of PSoC or in FPGA; b) the sorted blocks are transferred to the PC through PCI-express; and c) the blocks are merged by software in the PC (components A and C). This case is indicated below as *Sorting+PC merge*.

Sorting in hardware only (see point 1 above) permits the circuits that process the maximum possible number of data items and can be entirely implemented in the programmable logic without any support from software to be evaluated. We also present the results of evaluation of the circuits including threshold values that are potential limitations of the methods proposed.

Evaluation of the proposed circuits has been done through a set of experiments with the network described in section 3.1 (depicted in Figure 3.1), selecting four data sets sizes of 512, 1024, 2048, and 4096 items (32 bits). The results are shown in Figure 5.2.

We counted only the percentage of look-up tables (LUTs), which are the primary PL/FPGA resources that are used for the network. The percentage of other resources is lower, for example, the percentage of flip-flops for the FPGA does not exceed 23% and for the PL – 31% for all data set sizes (from 512 to 4096). From Figure 5.2 we can see that the available resources permit only iterative networks of up to 2048 32-bit data items to be implemented. Thus 2048 is the threshold for hardware only implementations based on the microchips indicated above. A preliminary evaluation shows that 8192 items is the maximum threshold value for hardware-only implementations of the circuit from section 3.1 in the most advanced FPGAs/PSoCs currently available on the market.

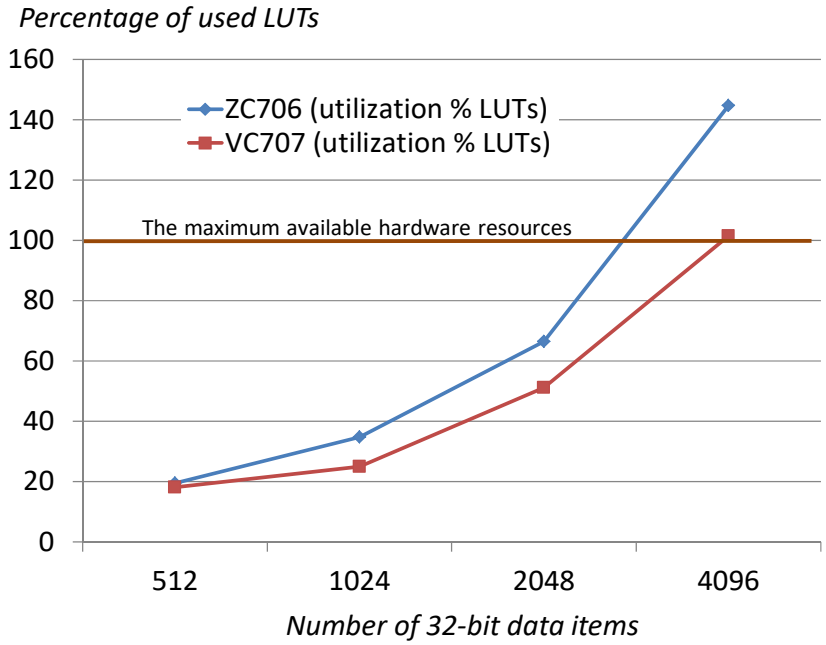

Figure 5.2 The results of sorting in hardware only using iterative networks described in section 3.1 [5]

The results obtained for the five measurements indicated above are reported in Figure 5.3 (the two curves *PC sort* and *PC sort + data transfer* show the same results without and with data transfers). The result for each type of experiment is an average of 64 runs.

The following conclusions can be drawn from Figure 5.3:

- The fastest results were obtained for the components A and C, i.e. pre-sort in the PL with a subsequent merge in the PC (see point 4 above). Note that the fastest (the lowest) curve in Figure 5.3– is built for sorting individual subsets only. Thus, the complete data set has not been sorted and the relevant results cannot be used for comparisons.

- The slowest result is shared between the remaining two cases (see points 2, 3 above).
- Note that for almost all data sizes, sorting and merging in PSoC is faster than sorting in PC software. Thus, cheaper (than PC) PSoCs are more advantageous and may be used efficiently for embedded applications.
- Sorting blocks in the PL network (see Figure 3.1) is significantly faster than subsequent merging. All communication and protocol overheads were taken into account.

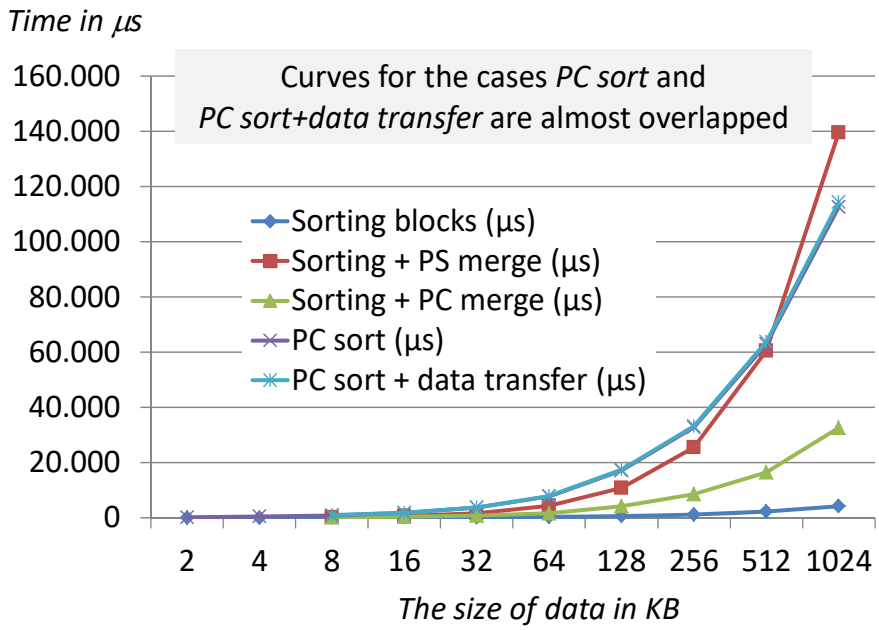

Figure 5.3 The results of experiments with the three-level system sorting data (the size of one block is 1024 32-bit data items) [5]

Similar experiments were done with the VC707 prototyping board, but with blocks of data containing 2048 32-bit data items (i.e. the blocks sorted in the hardware network are two times larger). The results are shown in Figure 5.4.

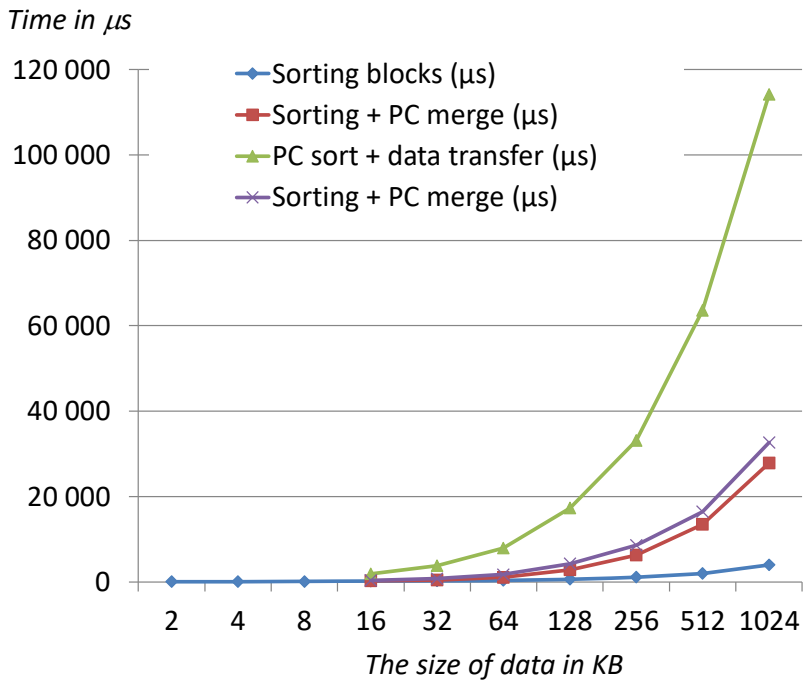

Figure 5.4 The results of experiments with the two-level system sorting data (the size of one block is 2048 32-bit data items) [5]

From analyzing these results we can conclude that:

- Using an FPGA from the Virtex-7 family, sorting in hardware networks is slightly faster, but the difference is negligible.
- Using larger blocks (2048 vs. 1024) allows sorting in point 4 (see the beginning of this section) to be faster by a factor ranging from 1.2 to 1.8. This is because the depth of software merges is reduced by one level.

Comparisons with the best known alternatives can be done by analyzing the fastest known networks. For data sorting, the latency and the cost of the most widely discussed networks are shown in Table 5.1. The formulae for the table are taken from [4] [27] [9] [36] [44]. For example, if N = 1024 then the latency is equal to D(1024)=55 for the fastest known even-odd merge and bitonic merge networks [24] [19], which is smaller than the number of iterations for the proposed network. However, C(1024) for the less resource consuming even-odd merge network is 24,063 C/S and for the proposed network C(1024) = 1023 C/S. Thus, the difference is a factor of about 24. It means that with the same hardware resources, the proposed networks can process blocks of data with significantly larger number N of data items. Indeed, the resources C(1024) = 24,063 of the known even-odd merge network are the same as for 24 proposed networks each of which sorts the same number of data items, i.e. 1024. This

means that the proposed network occupies less than 5% of the resources of the known network and the number of sorted items is exactly the same.

TABLE 5.1. Cost C(N) and latency D(N) of the most widely discussed networks

Type of the network	C(N)	D(N)
Bubble and insertion sort	$N\times(N-1)/2$	$2\times N-3$
Even-odd transposition	$N\times(N-1)/2$	N
Even-odd merge	$(p^2-p+4)\times 2^{p-2}-1$, $N=2^p$	$p\times(p+1)/2$, $N=2^p$
Bitonic merge	$(p^2+p)\times 2^{p-2}$, $N=2^p$	$p\times(p+1)/2$, $N=2^p$
The proposed network (see Figure 3.1)	$N-1$	$\leq N$

The experiments done for the board Xilinx vc707 [146] have shown that for the networks [24] [19] $N\leq 128$, while for the proposed networks $N \leq 2048$. Thus, the proposed networks may handle about 16 times larger blocks. The blocks created in hardware are further merged in software, thus the number of levels in software will be increased in the known networks by a factor of $\lceil \log_2 16 \rceil = 4$ (comparing to the proposed network). The following experiments were done:

1. Blocks with two sizes (that are 128 and 2048 32-bit words) have been sorted in software using the known (for the size 128) and the proposed (for the size 2048) networks. The measured times are T_{128} and T_{2048}.
2. Since the known networks cannot be used for N=2048, the same results have been obtained through a subsequent merge in software of blocks with N=128 to get blocks with N=2048. The measured time is $T_{128} + T_{merge}$.
3. Finally we measured the value $(T_{128} + T_{merge}) / T_{2048}$. The fastest method was used i.e. pre-sort in the PL with subsequent merge in the PC. The result that was an average of 64 runs exceeds 5. Note that additional delays appeared also in data transmission through PCI-express of smaller blocks of data items.

For subsequent merging required for larger data sets all the conditions for the proposed and known methods are the same. Thus, the proposed methods are always faster because merging in software begins with significantly larger pre-sorted blocks. Clearly, threshold values for maximum sizes of sorted sets are the same as for general-purpose software running in a PC.

5.1.2. Hardware merging of sorted subsets

The system with network-based sorter with subsequent merge in hardware for smaller data fragments and software merge for larger fragments described in section 3.1.2 was also implemented. It was compared with both software sorting and hardware sorting with PC merge. Figure 5.5 demonstrates organization of experiments with data sorters for the three-level system.

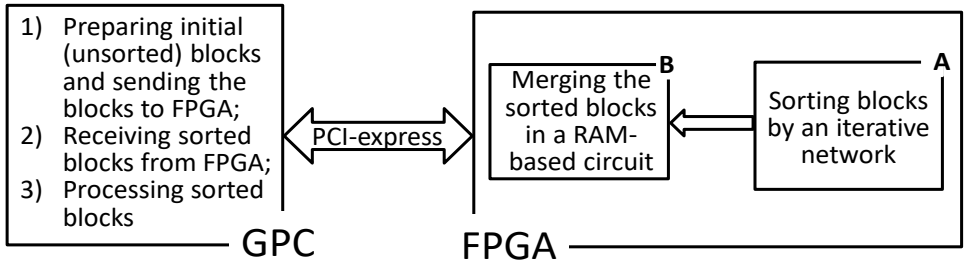

Figure 5.5 Organization of experiments with data sorters (the size of the input data is 256KB of 32-bit data items) [118]

The system for data transfers between a host PC and an FPGA has been designed, implemented, and tested. Experiments were done in the VC707 prototyping board that contains Virtex-7 XC7VX485T FPGA from the Xilinx 7th series with PCI express endpoint connectivity "Gen1 8-lane (x8)". All circuits were synthesized from the specification in VHDL and implemented in the Xilinx Vivado 2016.2 design suite. Software programs in the host PC run under Linux operating system and they were developed in C language. Data were transferred from the host PC to the VC707 and back through PCI express. The host PC is based on Intel core i7 3820 3.60 GHz.

Experiments have been done in accordance with Figure 5.5. The maximum size of data that are entirely sorted in FPGA is 256 KB. For larger size of data additional merging is done in the host PC. The results and comparison with sorting in the host PC are presented in Figure 5.6. It is clearly seen that sorting throughput for the proposed systems is significantly better than in the host PC. For example, 1,024 KB data can be sorted in the proposed system in 0.016 s and in the host PC in 0.11 s. Comparing the time of sorting reported in the referenced papers and the results of Figure 5.6 clearly demonstrate that the proposed solutions are faster.

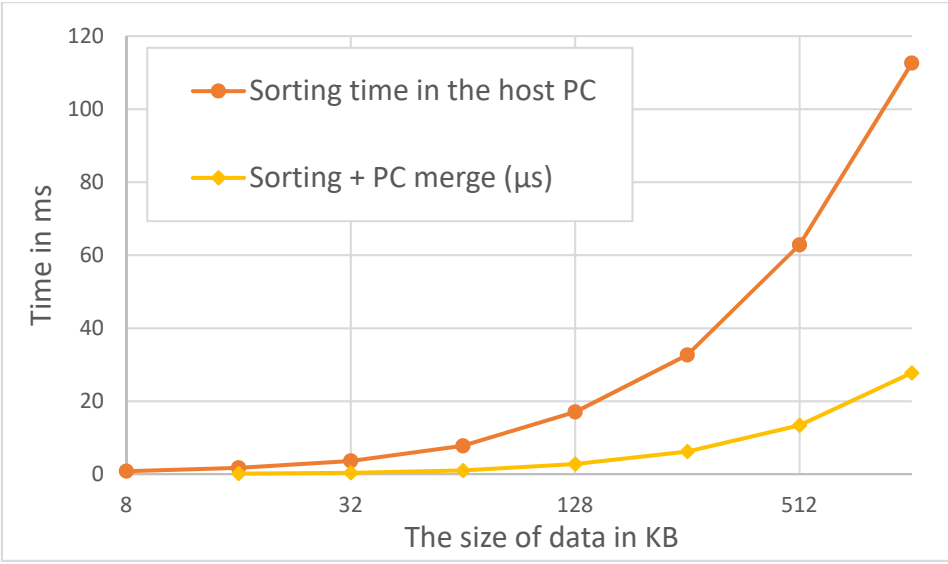

Figure 5.6 Comparison with software sorting

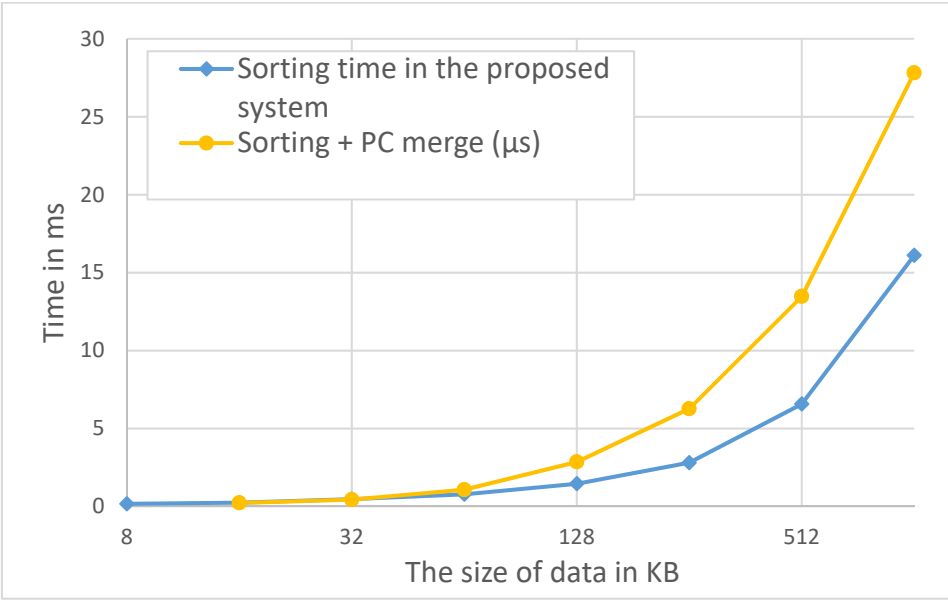

Figure 5.7 Comparison with hardware sorting with PC merge.

The comparison with software merge solution with larger sorting network proposed above is presented in Figure 5.7. The results demonstrate that the proposed sorting with subsequent data merge in hardware is faster than sorting with PC merge, but both solutions perform better than the software sorting. In hardware merge-based solution we have utilized twice smaller sorting network, because of resource limitations of the VC707 device. Both solutions perform

almost identically with small data sets, but the hardware merge performs better starting with 64KB date set and peaks with 256KB set, which is maximal possible set to be sorted solely in hardware. After 256KB threshold the system starts using software merge similar to the software merge-based solution, but it merges data blocks of 256KB instead of 8KB. 1,024 KB data can be sorted in the hardware merge-based system in 0.016 s and in the software merge-based in 0.027 s. Also it is important to mention that hardware-merge based solution utilized more than 70% of the device RAM blocks, while software merge solution doesn't necessitate the usage of RAM blocks.

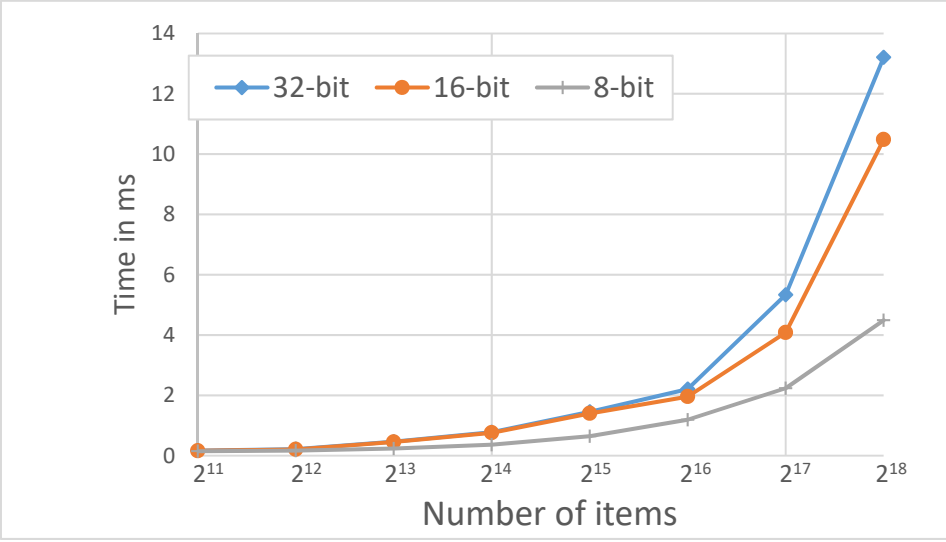

Figure 5.8. Experimental results of sorting data sets with simultaneous item counting for different item sizes [119]

Merging with item counting was performed for 32-, 16- and 8-bit items. The 36-bit size of the word for 32-bit items in the BRAM was chosen. It means that the item count part of the word is 4-bit and capable of counting up to 15 repetitions, which is enough for experiments with randomly generated data. The system was configured to work with 32-bit words with 16-bit size of both the item and the count parts for counting and merging of 16- and 8-bit data.

The experiments were conducted with randomly generated numbers. The merging with counting 32-bit items didn't show any noticeable speedup over simple merging, since 2^{16} of randomly generated numbers do not have significant number of repetitions. The merging with counting of the same number of 16-bit data items is 1,45 times faster than the simple merge and merge of 8-bit items is 27,28 times faster.

We experimented with different volumes of 8-, 16- and 32-bit data items and compared them with software sorting. The host PC was used for merging the data sets larger than volume of data that can be processed with the FPGA. In

addition to data sorting and merging, PCI express throughput and operating system overhead were also taken into account.

Figure 5.8 depicts comparison of sorting data sets in the proposed system with 8-, 16-, and 32-bit item sizes.

5.2. Partial sorting

We have implemented all the proposed in section 3.2 methods of partial sorters for minimal and maximal subset extractors using all platforms discussed in Chapter 4. Initially all these solutions were designed for PS/PL PSoC implementation, but we have conducted experiments for 2- and 3-level architectures involving host PC for exploration of additional features and comparison with known hardware alternatives.

5.2.1. Hardware/software implementation of simultaneous min/max extractors

The hardware/software systems for min/max subset extraction were designed as it was proposed in section 4.1.1. Xilinx PSoC Zynq-7000 was chosen as a platform for this implementation.

Figure 5.9 shows the organization of experiments. Initial (source) data are either generated randomly in software of the PS with the aid of C language `rand` function (see number 1 in Figure 5.9) or prepared in the host PC (see number 2 in Figure 5.9). 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 PSoC xc7z020-1clg484c housed on ZedBoard [143] with the aid the software/hardware architecture described in section 4.1. 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 Figure 5.9) or in the host PC (see number 4 in Figure 5.9). Functions for verification of the results are given in [131]. Verification time is not taken into account in the measurements below.

Synthesis and implementation of hardware modules were done in Xilinx Vivado 2016.2 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 2016.2) using methods described in [131]. Interactions with PSoC are done through the SDK console window.

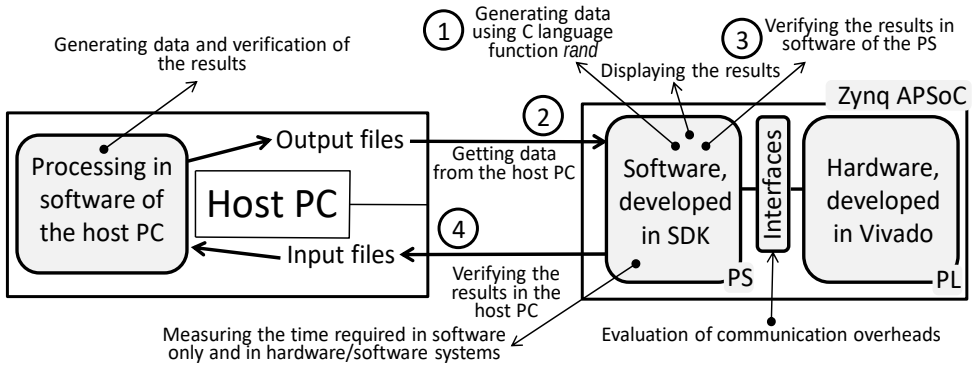

Figure 5.9 Experimental setup [120]

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 [131] [136] [137]. 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 [135], 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. Figure 5.10 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. Figure 5.11 shows the acceleration of software/hardware systems comparing to the software only sorting in the PS. Note that Figures 5.10, 5.11 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.

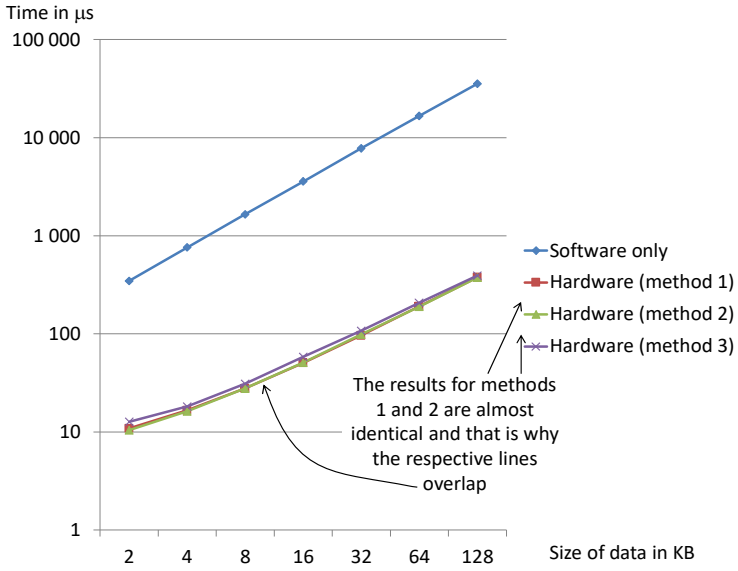

Figure 5.10 Computing time in software only and software/hardware systems. Method 1 – three sorting networks (section 3.2.1), method 2 – swapping networks (section 3.2.2), method 3 – switchable comparators (section 3.2.3) [120]

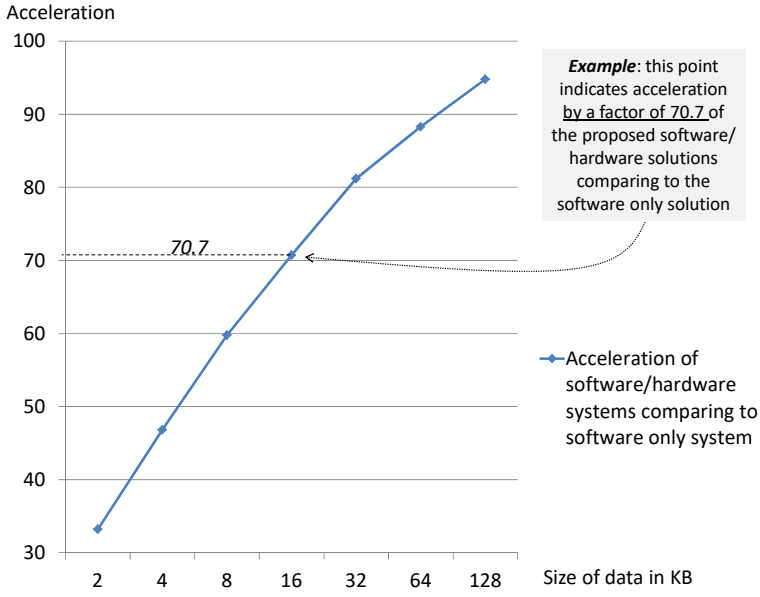

Figure 5.11 Acceleration of software/hardware systems comparing to software only system [120]

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 3.2.5.) then acceleration is decreased. Table 5.2 presents the results for extracting larger subsets (containing from 127 to 505 32-bit data items) from 128 KB set.

Table 5.2. 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 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 Figure 5.11 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 [131] (in all experiments M=32). We found that for small and moderate subsets the proposed here methods provide significantly better acceleration.

5.2.2. Three-level system for min/max extractors

The next experiments were done extracting the maximum and the minimum sorted subsets using the system described in section 4.3, which involves usage of general purpose PC and PCI express communication. We found that the acceleration is better than for complete data sorters described in section 5.1, which use the same approach. This is because the number of data transferred through PCI express is significantly decreased and almost all operations are done in the PSoC/FPGA. We implemented and tested the iterative circuit presented in section 3.1 (Figure 3.1) in the PL of PSoC, which takes data from the DDR memory and extracts the maximum and minimum subsets with L_{max}/L_{min} data items, where L_{max}/L_{min} varies from 128 to 1024 (as before M = 32, L varies from 2 KB to 1024 KB). Table 5.3 presents the results for $L_{max}/L_{min} = 128$.

TABLE 5.3. The results of experiments extracting the maximum/minimum subsets

Data (KB)	Time (μs)	Data (KB)	Time (μs)
2	70	64	254
4	75	128	425
8	89	256	916
16	112	512	1543
32	157	1024	3535

Table 5.3 presents the results for larger numbers of data items in extracted subsets (from 128 to 1024) for L = 256 KB.

For very large subset extraction the approach described in section 3.2.5 was used. Table 5.4 represents experimental results for very large scale extraction based on 3-level for subsets up to 512 data items.

TABLE 5.4.: The results of experiments with extracting subsets with different number of data items

Data	Time (µs)	Data	Time (µs)
128+128	916	640+640	4481
256+256	1808	768+768	5372
384+384	2698	896+896	6261
512+512	3589	1024+1024	7152

5.2.3. Separate min/max extractors and comparison with known hardware alternatives

If only the maximum or only the minimum subsets have to be computed the acceleration is almost the same as with maximal and minimal extraction, but the occupied hardware resources are reduced.

We implemented only minimum or only maximum subsets extractors with an aim to compare it with known alternatives. For this implementation Xilinx Virtex-7 FPGA was chosen and the two level-based architecture from section 4.2 was used. We compared it with software sorting and a hardware solution from [39] (OEM/BM). Software solution is the most obvious and the most widely used quicksort implementation from C++ language (sort function). With this approach a whole data set is being sorted with subsequent extraction of the maximal (or minimal) subset. For comparison in hardware area, the system from [39] was implemented. After some experiments we found the optimal configuration for implementation for Virtex-7 device which extracts 128-item data sets. Any implementation for extracting 256-item data sets utilizes more than 100% resources of the device. We used suggested in the section 3.2.4 concept of iterative max-set-selection units. The basis of this system is constructed from the two following blocks: 256-to-128 odd-even merge max-selection units and reduced bitonic 256-to-128 unit which starts with core max-selection unit. Inputs for core max selection units are outputs of OEM 256-to-128 and outputs of BM sorter (which contains results from the previous iteration).

For our methods we implemented two different systems. One for finding 128-item data subset in order to compare with OEM/BM method, and another for finding 1024-item data sets which is the maximal possible circuit that fits in the chosen Virtex-7 device. Post-implementation resource usage is shown in Table 5.5. Methods A and B in this table refer to the methods described in section 3.2.4 and depicted in Figure 3.15. Method A is a method based on two sorting networks and Method B is a method based on swapping networks.

Table 5.5. Resource utilization for methods A and B from Figure 3.15

Method	Resources	
	FF	LUT
Method A 128	9%	22%
Method B 128	8%	19%
Method A 1024 (max)	38%	94%
Method B 1024 (max)	22%	70%
OEM/BM 128 (max)	52%	78%

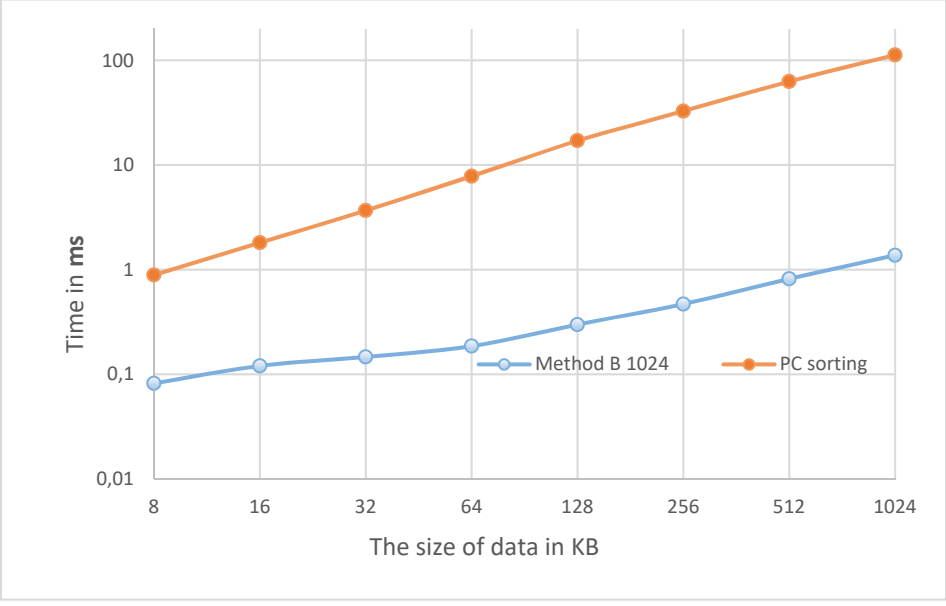

Figure 5.12 Experimental results. Hardware subset extraction based on swapping network compared to software solution [122].

Lookup table (LUT) usage for the method A is 3,5 times smaller and for the method B is 4 times smaller than OEM/BM based solution. The method A requires 5,7 times fewer amount of flip-flop (FF) than OEM/BM and the method B requires 6,5 times fewer FFs. Also it is necessary to mention that all modules required for PCIe DMA system utilize about 15% of LUTs. By subtracting these resources we see that pure min/max system for the method A requires 9 times fewer LUTs and the method B requires 15,7 times fewer LUTs.

Available resources of Virtex-7 device allow us to expand our circuits for extracting larger maximum or minimum subsets. Both proposed architectures were expanded to extract subsets of 1024 items which is 10 times more than with OEM/BM approach. Although for simultaneous extracting of maximum and minimum subsets both proposed methods are identical in terms of resource usage and performance, the method B is better for extraction of maximum or minimum subset alone.

Fig. 5.12 shows experimental results. With Virtex-7 and the proposed PCI express transfer system all hardware implementations showed approximately identical results. With architectures that allow faster data transfer OEM/BM approach may show better results, because for the proposed methods A and B the worst case performance is K/2 clock cycles for K inputs and OEM/BM performance is dependent on the number of pipeline stages. But because of significant economy of resources with the proposed methods (especially the method B) it is possible to speed up sorting by placing two or more instances of the sorting circuit that will sort parts of the whole data simultaneously.

Comparison of the proposed methods for extracting the maximum and minimum sorted subsets with the results in [39] demonstrates that the proposed method permits significantly larger subsets to be constructed. Indeed, the maximum size of extracted subsets in [39] is smaller and the maximum size of initial set is only 256 items. This is because the methods [39] are based on even-odd merge and bitonic merge networks for which the complexity of the circuits, i.e. the value of C(N), is limited. In our case, the maximum size of extracted subsets is 1024 (which exceeds the size of initial data sets in [39]) and the size of initial set is up to 1024 KB. The size of each item is 32 bits. The conclusion is the following: 1) the proposed methods enable data sets with significantly larger numbers of items to be processed; 2) the size of the extracted (minimum, maximum, or both) subsets may be increased in the proposed networks; 3) the performance (throughput) for processing large subsets in the proposed methods is better because complex tasks cannot be entirely solved in hardware using the methods [39] and the necessary software introduces large additional delays.

5.3. Hamming weight and matrix covering

The charts in Figure 5.13(a) permit to compare the suggested architectures with the best known alternatives, such as [89], [88], [87]. All the circuits were

synthesized, implemented in the Xilinx Zynq xc7z020 microchip, and tested in two prototyping boards: 1) Xilinx Zynq-7000 EPP ZC702; and 2) ZedBoard.

The first chart (Figure 5.13(a)) shows the maximum combinational path delay and the second chart indicates the number of FPGA slices for different designs. The total number of available slices in the microchip xc7z020 is 13 300. For our circuits we also considered pipelined implementations which include additional registers between layers (see PLR in Figure 3.18). We found that the maximum delay between the registers can be as little as 1.253 ns. Thus, potential throughput can be less than 2 ns per weight.

Implementation of matrix covering circuit which includes HW counters described in section 3.3 was done in the Xilinx Zynq-7000 PSoC ZC702 evaluation kit. Software for the ARM was developed in C language and hardware for the PL was synthesized from specification in VHDL. Experiments were done with two types of matrices 32 × 32 and 64 × 64. Thus, either 32 + 32 = 64 or 64 + 64 = 128 HW counters have been implemented in the PL section and all these circuits can run in parallel. Since C_{min} and R_{max} are found at different steps of the algorithm, only half of the HW counters work in parallel enabling either the minimal column C_{min} or the maximal row R_{max} to be found.

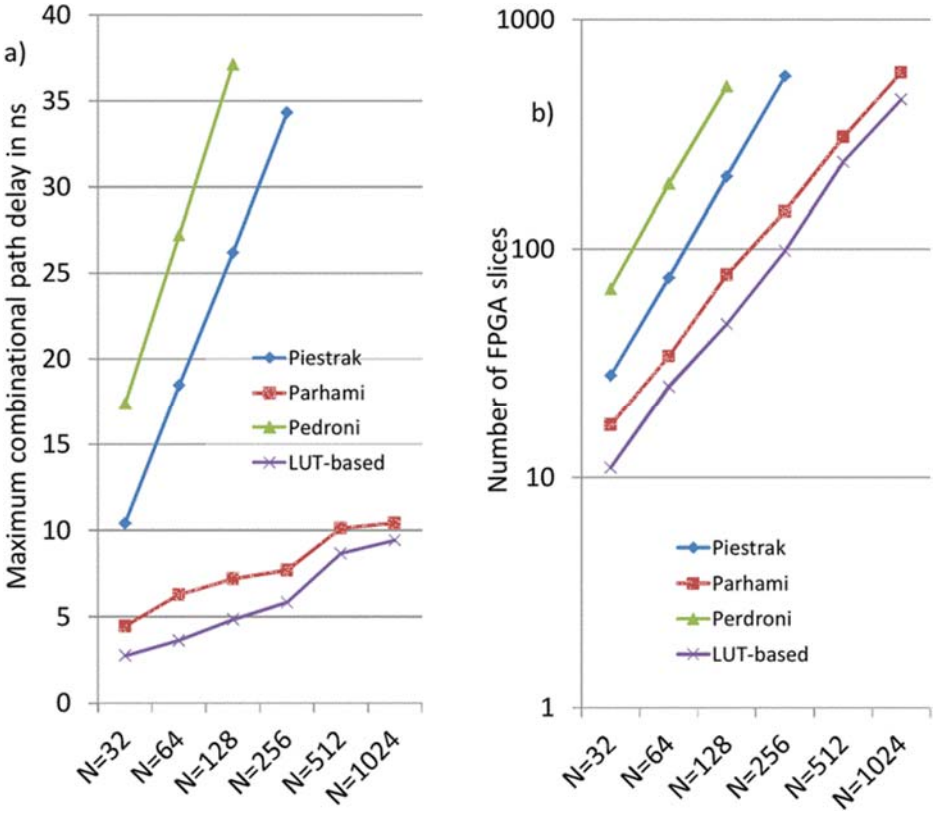

Figure 5.13 Latency (a) and cost (b) comparison with comptitive solutions by Piestrak [88], Parhami [89] amd Perdroni [87]. [123]

We compared three different implementations in which the covering algorithm is either:

1. Described in C language program running in PC with Intel i7 2.66 GHz processor;
2. Described in C language program running in ARM Cortex-A9;
3. Implemented in the PS and in the PL of Zynq-7000 PSoC.

Initial matrices have been generated randomly using the C rand function and identically for all the described above implementations. The number of instances (examples) was chosen to be 100,000.

In the last case (see the point 3 above) that is the original contribution of the thesis the following results have been obtained:

1. Generating in the PS and transmitting the matrices from the PS to the PL requires about 31 μs for 32 rows and 32 columns and about 34 μs for 64 rows and 64 columns. Only one AXI 32-bit (for the matrices 32 × 32) or 64-bit (for the matrices 64 × 64) port from the 4 available ports has been used. Clearly, additional ports permit the indicated time to be reduced;
2. Each iteration in the PL is executed in about 28 ns for the matrices 64 × 64 and about 24 ns for the matrices 32 × 32;
3. Communications between the PS and the PL (through interrupts and general-purpose ports) at any iteration of the algorithm require negligible time comparing to other operations.

The covering is found significantly faster than in software. The acceleration comparing with the PS only (see point 2 above) is from 30 to 50 times and comparing with the PC (see point 1 above) is from 5 to 10 times. This is because operations of the covering algorithm in software require many cycles and frequent transmission of data between processors and memories. For example, if we consider 64 × 64 matrices then a single matrix transfer from the PS to the DDR takes 33,300 ns on average and this is the most time consuming operation. Data transfer from the DDR to the PL is done in 284 ns on average. Once the PL receives the matrix data, no more interaction with the DDR is required for further processing.

5.4. Summary

In this chapter experimental results are presented. We conducted experiments for methods presented in Chapter 3 and implemented using approaches described in Chapter 4.

The experiments were done with an advanced prototyping systems of Xilinx 7th series FPGA and PSoC devices, allowing data processing in complex hardware/software systems.

We implemented, verified and tested methods proposed in this work and compared them with software running on general purpose computer and hardware solutions known from publications.

6. CONCLUSIONS

This thesis explored different methods of network-based accelerators for parallel data processing in several subjects. This chapter summarizes the main thesis contributions and outlines the directions for the future work.

The accelerators for solutions of the problems proposed in this thesis are in very high demand in many areas and especially in those where time and resource consumption is critical. Fast sorting of high volumes of incoming streaming data with simultaneous subsets extraction and processing are vital tasks in many real-time systems where the information must be quickly analyzed.

The architectures for building such accelerators in multi-level hardware/software systems were also proposed. These approaches can be modified for large variety of different data processing tasks which require fast analysis of the streaming data.

The main contributions of the presented work are summarized below.

- Hardware/software architectures for fast extraction of minimum and maximum sorted subsets from large data sets and three 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. The extracted subsets may be filtered and this feature is useful for control applications. The proposed solutions are highly parallel permitting capabilities of programmable logic to be used very efficiently. All the suggested methods were implemented in commercial microchips, tested, evaluated, and compared with alternatives. The results of experiments have shown significant speed-up of the proposed software/hardware systems comparing to software only systems and to competitive hardware/software implementations. The advantages of the proposed techniques over competitive hardware/software techniques include the ability to sort significantly larger data fragments and significantly more efficient resource utilization. The proposed techniques require from 9 to 15 times less FPGA LUTs than known alternatives for extracting subsets of the same sizes. The acceleration over GPP-only solutions is significant.

- Hardware/software architectures for data sorting that involve sorting and merging operations.

The distinctive feature of these architectures is parallelization at several stages with the adjusted time. The first stage is data sorting in hardware using periodic pipelined sorting networks and it is done in such a way that data acquisition, sorting and transferring the sorted data are carried out at the same time. The last stage is merging of hardware sorted subsets in software. The first architecture consists of these two stages, while the second architecture has the middle stage, which is a hardware pipelined RAM-based merger that enables merging at

different levels to be done in parallel and it can also be combined with the first stage. Such type of processing is efficient for sorting large sets (tens and hundreds millions of data items). The experiments were done with an advanced prototyping system (allowing data processing in a general-purpose computer and in recent FPGA from the Virtex-7 and PSoC Zynq-7000 of Xilinx). The results of experiments demonstrate significant acceleration comparing to general-purpose software and the results reported in publications. In comparison with other sorting networks the proposed sorters occupy significantly less hardware resources and therefore can sort larger amounts of data. The proposed network occupies less than 5% of the resources of the known network and the number of sorted items is exactly the same. Therefore the proposed system with subsequent merging is always faster than the alternatives because the merging starts with significantly larger sorted data subsets. Additionally the solution which involves simultaneous data sorting and item counting was proposed. This approach demonstrates even better performance with data sets with high number of repeated items and requires approximately the same amount of the hardware resources. It provides both fully sorted data set and a list of repeated data items.

- Hamming weight/distance counters/comparators based on FPGA LUTs.

The results of experiments confirm correctness and effectiveness of the proposed technique. The proposed approach showed better results in both performance and resource utilization in comparison with other known alternatives.

- A novel technique for implementation of matrix/set covering algorithms in hardware and software of recent all programmable systems-on-chip.

A new method that permits the known approximate algorithm to be executed over suggested unrolled matrices is discussed and the relevant hardware accelerator is developed. It is shown that the covering algorithm can efficiently be partitioned in software and hardware modules that finally have been completely implemented and tested in Xilinx Zynq microchips. The results of experiments and comparisons with two different software implementations demonstrate significant speedup which is very important for various practical applications that are also mentioned in the paper. The comparison with PS only implementation showed the acceleration from 30 to 50 times and with PC – 5 to 10 times.

6.1. Future Work

This section outlines tasks and directions that need further investigation in the scope the studied topic.

The investigation of different properties of network-based algorithms should be continued in order to increase hardware acceleration even further. The ability of swapping networks to generate bitonic sequences was higlighted and their properties can be integrated in the merge-tree structure for merging sorted data sets. Different methods proposed in this thesis could be also integrated for acceleration of more complex practical applications. One of those possible applications is integration of data sorting with simultaneous item counting and subsets extraction for solving the most frequent item computation.

The new generation of PSoC devices which combine FPGA, CPU, real-time CPU and GPU on the same microchip should be analyzed and utilized for the applications discussed in this research. The distributed methods proposed in this thesis will definitely benefit from these newly emerged platforms.

REFERENCES

[1] V. Sklyarov, I. Skliarova, "Digital Hamming Weight and Distance Analyzers for Binary," *International Journal of Innovative,* vol. 9, no. 12, pp. 4825-4849, 2013.

[2] R. Mueller, J. Teubner, G. Alonso, "Data processing on FPGAs," *Proceedings of the VLDB Endowment,* vol. 2, no. 1, pp. 910-921, 2009.

[3] G. Steiner, B. Philofsky, "Managing Power and Performance with the Zynq UltraScale+ MPSoC," October 2016. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp482-zu-pwr-perf.pdf. [Accessed June 2017].

[4] D. E. Knuth, The Art of Computer Programming, Sorting and Searching, vol. III, Addison-Wesley, 2011.

[5] V. Sklyarov,A. Rjabov, I. Skliarova, A. Sudnitson, "High-performance Information Processing in Distributed Computing Systems," *International Journal of Innovative Computing, Information and Control,* vol. 12, no. 1, pp. 139-160, 2016.

[6] L. V. Kalé, E. Solomonik, "Sorting," in *Encyclopedia of Parallel Computing*, Springer Science+Business Media, 2011, pp. 1855-1862.

[7] G. Bilardi, F. P. Preparata, "Area-time lower-bound techniques with applications to sorting," *Algorithmica,* vol. 1, no. 1-4, pp. 65-91, 1986.

[8] S. Sengupta et al., "Scan primitives for GPU computing," *Graphics hardware,* pp. 97-106, 2007.

[9] R. Mueller, Data Stream Processing on Embedded Devices, Ph.D. thesis., Zurich: ETH, 2010.

[10] S. Kestur, J. D. Davis, O, Williams, "Blas comparison on fpga, cpu and gpu," EEE computer society annual symposium on VLSI (ISVLSI), 2010.

[11] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, J. Lemeire, "Comparing and combining GPU and FPGA accelerators in an image processing context," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2013.

[12] S. Asano, T. Maruyama, Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *International Conference on Field Programmable Logic and Applications*, 2009.

[13] V. Venugopal, D. M. Shila, "High throughput implementations of cryptography algorithms on GPU and FPGA," in *IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, 2013.

[14] P. Sanders, T. Hansch, "Efficient massively parallel quicksort," in *Solving Irregularly Structured Problems in Parallel*, vol. 4, Springer Berlin Heidelberg, 1997, pp. 13-24.

[15] M. Zagha, G. E. Blelloch, "Radix sort for vector multiprocessors," in *ACM/IEEE conference on Supercomputing*, New York, 1991.

[16] J. S. Huang, Y. C. Chow, "Parallel sorting and data partitioning by sampling," in *International Computer Software and Application Conference*, 1983.

[17] D. R. Helman, D. A. Bader, J. JáJá, "A randomized parallel sorting algorithm with an experimental study," *journal of parallel and distributed computing,* vol. 52, no. 1, pp. 1-23, 1998.

[18] L. V. Kale, S. Krishnan, "A comparison based parallel sorting algorithm," in *International Conference on Parallel Processing*, 1993.

[19] S. W. A. Baddar, K.E. Batcher, Designing Sorting Networks. A New Paradigm., Springer, 2011.

[20] M. Codish, L. Cruz-Filipe, P. Schneider-Kamp, "The Quest for Optimal Sorting Networks: Efficient Generation of Two-Layer Prefixes," in *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2014.

[21] D. Bundala, J. Závodný, "Optimal sorting networks," *Language and Automata Theory and Applications,* pp. 236-247, 2014.

[22] M. Ajtai, J. Komlós, E. Szemerédi, "An 0 (n log n) sorting network," in *ACM symposium on Theory of computing*, 1983.

[23] F. T. Leighton, "Tight Bounds on the Complexity of Parallel," *IEEE Trans. Computers,* vol. 34, pp. 344-354, 1985.

[24] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Computer Conference*, 1968.

[25] S. Lacey, R. Box, "Box, A Fast, Easy Sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines," *Byte,* vol. 16, no. 4, pp. 315-320, 1991.

[26] R. D. Chamberlain, N. Ganesan, "Sorting on Architecturally Diverse

Computer Systems," 2009.

[27] R. Mueller, J. Teubner, G. Alonso, "Sorting networks on FPGAs," *The International Journal on Very Large Data Bases,* pp. 1-21, 2012.

[28] M. Zuluada, P. Milder, M. Puschel, "Computer Generation of Streaming Sorting Networks," in *Proc. 49th Design Automation Conference*, 2012.

[29] V. Sklyarov, I. Skliarova, "High-performance implementation of regular and easily scalable sorting," *Microprocessors and Microsystems,* vol. 38, no. 5, pp. 470-484, 2014.

[30] M. Zuluaga, P. Milder, M. Püschel, "Streaming Sorting Networks," *ACM Transactions on Design Automation of Electronic Systems (TODAES),* vol. 21, no. 4, pp. 55:1-55:30, 2016.

[31] H. Schröder, "Partition sorts for VLSI," in *13th GI-Jahrestagung*, 1983.

[32] A. Grasselli, "Control units for sequencing complex asynchronous operations," *IEEE Transactions on Electronic Computers,* vol. 4, no. EC-11, pp. 483-498, 1962.

[33] W. H. Kautz, K. N. Levitt, A. Waksman, "Cellular interconnection arrays.," *IEEE Transactions on Computers,* vol. 100, no. 5, pp. 443-451, 1968.

[34] S. N. Salloum, D. H. Wang, "Fault tolerance analysis of odd-even transposition sorting networks with single pass and multiple passes," in *IEEE Pacific Rim Conference on Communications, Computers and signal Processing*, 2003.

[35] A. Hematian, S. Chuprat, A. A. Manaf, N. Parsazadeh, "Zero-Delay FPGA-Based Odd-Even Sorting Network," in *IEEE Symposium on Computers & Informatics*, 2013.

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

[37] J. Ortiz, J, D. Andrews, D, "A configurable high-throughput linear sorter system," in *Workshops and Phd Forum In Parallel & Distributed Processing*, 2010.

[38] R. Marcelino, H. C. Neto, J. M. P. Cardoso, "A comparison of three representative hardware sorting units," in *35th Annual Conference of Industrial Electronics*, 2009.

[39] A. Farmahini-Farahani, H. J. Duwe, M. J. Schulte, K. Compton,

"Modular design of high-throughput, low-latency sorting units.," *IEEE Transactions on Computers,* vol. 62, no. 7, pp. 1389-1402, 2013.

[40] W. Song, D. Koch, M. Luján, J. Garside, "Parallel Hardware Merge Sorter," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, Washington DC, USA, 2016.

[41] R. Chen, V. Prasanna, "Accelerating Equi-Join on a CPU-FPGA," Ming Hsieh Department of Electrical Engineering – Systems, University of Southern California, Los Angeles, California, 2016.

[42] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, "GPU computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, 2008.

[43] I. Buck, T. Purcell, "A toolkit for computation on GPUs," *GPU Gems,* vol. 1, p. 621–636, 2004.

[44] P. Kipfer, R. Westermann, "Improved GPU sorting," *GPU gems,* vol. 2, pp. 733-746, 2005.

[45] A. Greb, G. Zachmann, "GPU-ABiSort: Optimal parallel sorting on stream architectures," in *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[46] M. Harris, S. Sengupta, J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU gems,* vol. 3, no. 39, pp. 851-876, 2007.

[47] E. Sintorn, U. Assarsson, "Real-time approximate sorting for self shadowing and transparency in hair rendering," in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008.

[48] E. Sintorn, U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing,* vol. 68, no. 10, pp. 1381-1388, 2008.

[49] N. Satish, M. Harris, M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *IEEE International Symposium Parallel & Distributed Processing*, 2009.

[50] N. Leischner, V. Osipov, P. Sanders, "GPU sample sort," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[51] X. Ye, D. Fan, W. Lin, N. Yuan, P. Ienne, "High performance comparison-based sorting algorithm on many-core GPUs," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[52] I. Tanasic, L. Vilanova, M. Jordà, J. Cabezas, I. Gelado, N. Navarro, W. M. Hwu, "Comparison based sorting for systems with multiple GPUs," in *6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013.

[53] S. Zezza, S. Nooshabadi, M. Martina, G. Masera, "Efficient implementation techniques for maximum likelihood-based error correction for jpeg2000," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 19, no. 4, p. 591–596, 2009.

[54] S. Goren, G. Dundar, B. Yuce, H. F. Ugurdag, "A fast circuit topology for finding the maximum of N k-bit numbers," in *Symp. on Computer Arithmetic*, 2013.

[55] C. Wey, M. Shieh, S. Lin, "Algorithms of finding the first two minimum values and their hardware implementation," *IEEE Trans. Circuits and Systems I,* vol. 55, no. 11, p. 3430–3437, 2008.

[56] A. D. G. Biroli, J. C. Wang, "A fast architecture for finding maximum (or minimum) values in a set. In Acoustics,," in *2014 IEEE International Conference on Speech and Signal Processing (ICASSP)*.

[57] V. E. Alekseyev, "Sorting Algorithms with Minimum Memory," *Kibernetica,* vol. 5, pp. 99-103, 1969.

[58] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, R. E. Tarjan, "Time Bounds for Selection," *Computer and System Sciences,* vol. 7, pp. 448-461, 1973.

[59] J. M. Chambers, "Algorithm 410: Partial sorting," *Commun. ACM,* vol. 14, no. 5, pp. 357-358, 1971.

[60] D. Wang, A. Mazumdar, G. W. Wornell, "Compression in the Space of Permutations," *IEEE Transactions on Information Theory,* pp. 6417 - 6431, 2015.

[61] A. Bertossi, S. Olariu, M. C. Pinotti, S. Q. Zheng, "Classifying matrices separating rows and columns," *IEEE Transactions on Parallel and Distributed Systems,* vol. 15, no. 7, pp. 654-665, 2004.

[62] S. Olariu, M. C. Pinotti, S. Q. Zheng, "An optimal hardware-algorithm for selection using a fixed-size parallel classifier device," *In High Performance Computing–HiPC,* pp. 284-288, 1999.

[63] G. Cormode, M. Hadjieleftheriou, "Finding Frequent Items in Data Streams," *VLDB Endowment,* vol. 1, no. 2, pp. 1530-1541, 2008.

[64] J. Teubner, R. Mueller, G. Alonso, "FPGA Acceleration for the Frequent

Item Problem," *Proc. 26th Int',l Conf. Data Eng,* 2010.

[65] J. Teubner, R. Muller, G. Alonso, "Frequent item computation on a chip," *IEEE Trans. Knowl. Data Eng,* vol. 238, pp. 1169-1181, 2011.

[66] S. Shaobo, Y. Qi; Q. Wang, "FPGA Acceleration for Intersection Computation in Frequent Itemset Mining," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2013.

[67] K. H. Rosen, J. G. Michaels, J. L. Gross, J. W. Grossman, D. R. Shier, Handbook of Discrete and Combinatorial Mathematics, Boca Raton: CRC Press, 2000.

[68] J. D. Davis, Z. Tan, F. Yu, L. Zhang, "A practical reconfigurable hardware accelerator for Boolean satisfiability solvers," in *Proc. 45th ACM/IEEE Design Automation Conference – DAC'2008*, Anaheim, California, USA, June, 2008.

[69] J. P. Marques-Silva, K. A. Sakallah, "Boolean Satisfiability in electronic design automation," in *Proceedings of DAC*, USA, Los, 2000.

[70] "The International SAT competitions web page," [Online]. Available: http://www.satcompetition.org.

[71] Y. Hamadi, S. Jabbour, "ManySAT: A parallel SAT solver," *Journal on Satisfiability, Boolean Modelling and,* vol. 6, pp. 245-262, 2009.

[72] I. Skliarova, A.B. Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," *IEEE Transactions on Computers,* vol. 53, no. 11, pp. 1449-1461, 2004.

[73] K. Kanazawa, T. Maruyama, "An Approach for Solving Large SAT Problems on FPGA," *ACM Transactions on Reconfigurable Technology and Systems,* vol. 4, no. 1, p. 10, 2010.

[74] K. Kanazawa, T. Maruyama, "An FPGA Solver for SAT-Encoded Formal Verification Problems.," in *International Conference on Field Programmable Logic and Applications*, 2011.

[75] K. Gulati, M. Waghmode, S. P. Khatr, W. Shi, "Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction," *Computers & Digital Techniques,* vol. 2, no. 3, pp. 214-229, 2008.

[76] K. Gulati, S. Paul, S. P. Khatri, S. Patil, A. Jas, "FPGA-based Hardware Accelerator for Boolean Satisfiability," *ACM Transactions on Design Automation of Electronic Systems,* vol. 14, no. 2, 2009.

[77] L. Haller, S. Singh, "Relieving capacity limits on FPGA-based SAT-solvers," *Formal Methods in Computer-Aided Design,* pp. 217-220, 2010.

[78] M. Suzuki, T. Maruyama, "Variable and clause elimination in SAT problems using an FPGA," in *International Conference on Field-Programmable Technology*, 2011.

[79] Z. Luo, H. Liu, "Cellular genetic algorithms and local search for 3-SAT problem on graphic hardware," in *IEEE Congress on Evolutionary Computation*, 2006.

[80] H. Deleau, C. Jaillet, M. Krajecki, "GPU4SAT: solving the SAT problem on GPU," in *PARA 2008 9th International Workshop on State–of–the–Art in Scientific and Parallel Computing*, Trondheim, Norway, 2008.

[81] Q. Meyer, F. Schönfeld, M. Stamminger, R. Wanka, "3-SAT on CUDA: Towards a massively parallel SAT solver," in *International Conference on High Performance Computing and Simulation (HPCS)*, 2010.

[82] S. Beckers, G. De Samblanx, F. De Smedt, T. Goedemé, L. Struyf, J. Vennekens, "Parallel SAT-solving with OpenCL," in *Proceedings of the IADIS International Conference on Applied Computing*, 2011.

[83] H. Fujii, N. Fujimoto, "GPU acceleration of BCP procedure for SAT algorithms," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2012.

[84] "Intel® SSE4 Programming Reference," Intel, July 2007. [Online]. Available: https://software.intel.com/sites/default/files/m/d/4/1/d/8/d9156103.pdf.

[85] "NEON Programmer's Guide," ARM Limited, 2013. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html.

[86] D. B. S. King, R. J. Simpson, C. Moore, I. P. MacDiarmid, "Digital n-tuple Hamming comparator for weightless systems," *Electronics Letters,* vol. 34, no. 22, pp. 2103-2104, 1998.

[87] V. Pedroni, "Compact Hamming-comparator-based rank order filter for digital VLSI and FPGA implementatio," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2004.

[88] S. J. Piestrak, "Efficient Hamming weight comparators of binary," *Electronics Letters,* vol. 39, no. 11, pp. 661-612, 2007.

[89] B. Parhami, "Efficient Hamming Weight Comparators for Binary Vectors Based on Accumulative and Up/Down Parallel Counters," *IEEE Trans. on Circuits and Systems—II: Express Briefs,* vol. 56, no. 2, pp. 167-171, 2009.

[90] K. Appiah, A. Hunter, P. Dickinson, H. Meng, "Binary object recognition system on FPGA with bSOM," in *SOC Conference (SOCC), 2010 IEEE International*, 2010.

[91] S. Jin, D. Kim, D. D. Nguyen, J. W. Jeon, "Pipelined Hardware Architecture for High-Speed Optical Flow Estimation using FPGA," in *Annual International Symposium on Field-Programmable Custom Computing Machines*, 2010.

[92] V. B. Kovačević, A. M. Gavrovska, M. P. Paskaš, "High-speed implementation of Hamming neural network," in *Symposium on Neural Network Applications in Electrical Engineering (NEUREL), 2010 10th*, 2010.

[93] F. Gioachin, A. Sharma, S. Chakravorty, C. L. Mendes, L. V. Kale, T. Quinn, "Scalable cosmological simulations on parallel machines," in *High Performance Computing for Computational Science-VECPAR*, Springer Berlin Heidelberg, 2006, pp. 476-489.

[94] D. Baily, E. Barscz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, D. Fineberg, P. Frederickson, S. Weeratunga, "The NAS Parallel Benchmarks," NASA Ames Research Center, Moffett Field, CA, 1994.

[95] C. Chakrabarti, S. Dhanani, "Median filter architecture based on sorting networks," in *IEEE International Symposium on Circuits and Systems*, 1992.

[96] K. Vasanth, S. N. Raj, S. Karthik, P. P. Mol, "Fpga implementation of optimized sorting network algorithm for median filters," in *International Conference on Emerging Trends in Robotics and Communication Technologies*, 2010.

[97] S. M. Meena, K. Linganagouda, "Rank based merge sorting network architecture for 2D median and morphological filters.," *IEEE International Advance Computing Conference,* pp. 473-479, 2009.

[98] J. Scott, M. Pusateri, M. U. Mushtaq, "Comparison of 2D median filter hardware implementations for real-time stereo video.," in *Applied Imagery Pattern Recognition Workshop*, 2008.

[99] D. Zmaranda, H. Silaghi, G. Gabor, C. Vancea, "Issues on Applying Knowledge-Based Techniques in Real-Time Control Systems,"

International Journal of Computers, Communications and Control, vol. 8, no. 1, pp. 166-175, 2013.

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

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

[102] D. Verber, "Hardware implementation of an earliest deadline first task scheduling algorithm.," *Informacije MIDEM,* vol. 41, no. 4, pp. 257-263, 2011.

[103] A. Gregerson, M. Schulte, K. Compton,, "High-Energy physics," *Handbook of Signal Processing Systems,* pp. 179-211, 2010.

[104] Z. K. Baker, V. Prasanna, "An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems," in *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, USA, 2006.

[105] S. Sun, Analysis and acceleration of data mining algorithms on high performance reconfigurable computing platforms. Ph.D. thesis, Iowa: Iowa State University., 2011.

[106] X. Wu, V. Kumar, J. R. Quinlan, "Top 10 algorithms in data mining," *Knowledge and Information Systems,* vol. 14, no. 1, pp. 1-37, 2014.

[107] M. F. M. Firdhous, "Automating Legal Research through Data Mining," *International Journal of Advanced Computer Science and Applications,* vol. 1, no. 6, pp. 9-16, 2010.

[108] M. Kik, M. Kutylowski, M. Piotrów, "Correction networks," in *International Conference on Parallel Processing*, 1999.

[109] M. Kik, "Periodic correction networks.," *Parallel Processing,* pp. 471-478, 2000.

[110] M. Piotrów, "Periodic, random-fault-tolerant correction networks," in *ACM symposium on Parallel algorithms and architectures*.

[111] G. Stachowiak, "Fibonacci correction networks," *In Algorithm Theory-SWAT,* pp. 535-548, 2000.

[112] G. Stachowiak, "Fast periodic correction networks. Theoretical computer science," *Theoretical computer science,* vol. 3, no. 354, pp. 354-366, 2006.

[113] H. Heo, J. Lee, C. Lee, "FPGA based Implementation of FAST and BRIEF algorithm for object Recognition," in *IEEE Region 10 Conference TENCON*, 2013.

[114] R. Hentati, M. Abid, B. Dorizzi, "Software implementation of the OSIRIS iris recognition algorithm in FPGA," in *International Conference on Microelectronics (ICM)*, 2011.

[115] C. Gu, M. O'Neill, "Ultra-compact and robust FPGA-based PUF identification generator," in *International Symposium on Circuits and Systems (ISCAS)*, 2015.

[116] S. Gehrer, G. Sigl, "Using the reconfigurability of modern FPGAs for highly efficient PUF-based key generation," in *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2015.

[117] R. P. Lippmann, "An introduction to computing with neural nets," *ASSP Magazine,* vol. 4, no. 2, pp. 4-22, 1987.

[118] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, "Fast Iterative Circuits and RAM-based Mergers to Accelerate Data Sort in Software/Hardware Systems," *Proceedings of the Estonian Academy of Sciences,* vol. 66, no. 4, 2017.

[119] A. Rjabov, V. Sklyarov, I. Skliarova, A. Sudnitson, "RAM-based mergers for data sort and frequent item computation," in *Conforence on Information and Communication Technology, Electronics and Microelectronics*, Opatija, Croatia, 2017.

[120] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, "Zynq-based System for Extracting Sorted Subsets from Large Data Sets," *Journal of Microelectronics, Electronic Components and Materials,* vol. 45, no. 2, pp. 142-152, 2015.

[121] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, "Computing Sorted Subsets for Data Processing in Communicating Software/Hardware Control Systems," *International Journal of Computers Communications & Control,* vol. 11, no. 1, pp. 126-141, 2016.

[122] A. Rjabov, "Hardware-based systems for partial sorting of streaming data," in *15th Biennial Baltic Electronics Conference*, Tallinn, 2016.

[123] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, "Implementation of Parallel Operations over Streams in Extensible Processing Platforms," in *The 56th IEEE International Midwest Symposium on Circuits and Systems*, Columbus, Ohio, USA, 2013.

[124] I. Skliarova, V. Sklyarov, A. Rjabov, A. Sudnitson, "Fast Matrix Covering in All Programmable Systems-on-Chip," *Elektronika ir Elektrotechnika,* vol. 20, no. 5, pp. 150-153, 2014.

[125] V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson, Hardware/Software Co-design for Programmable Systems-on-Chip, Tallinn: TUT Press, 2014.

[126] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2009, p. 1312 .

[127] A. Zakrevskij, Y. Pottosin, L. Cheremisiniva, Combinatorial Algorithms of Discrete Mathematics, Tallinn: TUT Press, 2008, p. 193 .

[128] V. Sklyarov, I. Skliarova, "Fast regular circuits for network-based parallel data processing," *Advances in Electrical and Computer Engineering,* vol. 13, no. 4, p. 47–50, 2013.

[129] Crockett L.H., Elliot R.A., Enderwitz M.A., and Stewart R.W, The Zynq Book, 2014.

[130] V. Sklyarov, I. Skliarova, A. Barkalov, L. Titarenko, Synthesis and Optimization of FPGA-based Systems, Springer, 2014.

[131] V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson, C. Cardoso, Hardware/Software Co-design for Programmable Systems-on-Chip, TUT Press, 2014.

[132] Altera, "Cyclone V Overview," 10 6 2016. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf. [Accessed 12 04 2017].

[133] Intel, "Arria 10 Device Overview," 15 3 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf. [Accessed 12 4 2017].

[134] Microsemi, "SmartFusion2 Product Information," 8 2016. [Online]. Available: https://www.microsemi.com/document-portal/doc_download/131308-smartfusion2-product-information-brochure. [Accessed 12 4 2017].

[135] Xilinx, Inc., "Zynq-7000 All Programmable SoC Technical Reference Manual," 2014. [Online]. Available:

http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

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

[137] Neuendorffer, S., and Martinez-Vallina, F., "Building Zynq Accelerators with Vivado High Level Synthesis," in *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, Monterey, CA, USA,, 2013.

[138] Xilinx, "AXI Central Direct Memory Access v4.1," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf..

[139] Xilinx, "LogiCORE IP AXI Bridge for PCI Express v1.06," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_5/pg055-axi-bridgepcie.pdf.

[140] Xilinx, "PCI Express Endpoint-DMA Initiator Subsystem," 2013. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1171-pcie-central-dma-subsystem.pdf.

[141] J. Corbet, A. Rubini, and G. Kroah-Hartman, Linux Device Drivers.

[142] Xilinx, Inc., "ZC702 Evaluation Board User Guide UG850 (v1.5)," 2015. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf.

[143] Avnet, Inc., "ZedBoard (ZynqTM Evaluation and Development) Hardware User's Guide," [Online]. Available: http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.

[144] Xilinx, Inc., "ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide UG954 (v1.6)," 2016. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf.

[145] Xilinx, Inc., "VC707 Evaluation Board for the Virtex-7 FPGA User Guide UG885 (v1.7.1)," 2016. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vc707/u

g885_VC707_Eval_Bd.pdf.

[146] Xilinx, Inc., "VC707 Evaluation Board for the Virtex-7 FPGA User Guide, UG885 (v1.4)," [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf.

[147] Xilinx, Inc., "Simple AMP Running Linux and," 2013.

[148] Xilinx, Inc., "OS and Libraries Document Collection UG647," 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/oslib_rm.pdf.

[149] J. D. Davis, Z. Tan, F. Yu, L. Zhang, "A practical reconfigurable hardware accelerator for Boolean satisfiability solvers," in *Proceedings of the 45th annual Design Automation Conference*, 2008.

ACKNOWLEDGEMENTS

I would like to thank everybody who helped me during my PhD studies and without whom this work would never appeared.

Particularly, I would like to express my gratitude to my supervisors Assoc.Prof. Aleksander Sunditsõn, Prof. Valery Sklyarov and Assist.Prof. Iouliia Skliarova for their support and for helping me to my first steps in the engineering domain. They have guided me through my PhD studies and provided challenging tasks. I has been a big pleasure to do research work with them.

I would also like to thank all the people who worked with me in Department of Computer Systems who contributed to my work discussions and ideas.

Special thanks to Dr. Margus Kruus, the head of Department of Computer Systems for his support with many administrative issues and to Prof. Andres Keevallik for his constant attention.

Furthermore, I would like to acknowledge several organizations that have supported my PhD studies: Tallinn University of Technology, Information Technology Foundation (HITSA), Estonian Association of Information Technology and Telecommunications (ITL) and Estonian Ministry of Education and Research.

Finally, I would like to thank my family for their patience and support.

Artjom Rjabov

Tallinn, April 2017

ABSTRACT

The thesis explores topics related to hardware acceleration of computationally intensive and resource consuming problems that may be used efficiently in information processing that is frequently needed in electronic, environmental, medical, and biological applications. We propose hardware acceleration methods for problems such as data sorting and merging, filtering and subset extraction, parallel covering of matrices/sets, Hamming weight computation and related tasks. Our solutions are based on highly parallel network-based methods which consist of large numbers of repeated elements.

We use reconfigurable technologies such as field-programmable gate arrays (FPGA) and programmable systems on chip (PSoC) as target platforms for implementation of our data processing methods. Effectivness of these platforms and their combinations were investigated in this research. These platforms are very appropriate for implementation of such systems because of their low cost, flexibility, availability and many other advantages.

The main contributions of this research are techniques for fast extraction of minimum and maximum sorted subsets from large data sets, data processing that involve sorting, merging operations and simultaneous item counting, hamming weight/distance counters/comparators, matrix/set covering and their implementations which involve hardware/software co-design and combinations of reconfigurable platforms.

KOKKUVÕTE

Selles väitekirjas uuritakse teemasid, mis on seotud arvutusmahukate ja ressursikulu probleemide lahendamise riistvarakiirendusega, mida võib kasutada informatsiooni töötlemisel, mis on tihti vajalik elektroonika-, keskkonna-, meditsiini- ja bioloogilistes rakendustes. Pakutakse välja riistvara kiirenduse meetodeid erinevate probleemide nagu informatsiooni sorteerimine ja ühendamine, filtreerimine ja alamhulkade ekstraheerimine, paralleelsete maatriksite/kogumite katmine, Hamming'u kaalu arvutamine ja nendega seotud ülesannete lahendamiseks. Lahendused põhinevad tugevalt paralleelsetel võrgu-põhistel meetoditel, mis koosnevad paljudest korduvatest arvutuselementidest.

Andmetöötlusmeetodite realiseerimiseks kasutatakse riistvaraplatvormina ümberkonfigureeritavaid tehnoloogiaid nagu väliprogrammeeritavad väravamassiivid (FPGA) ja programmeeritavad süsteemid kiipidel (PSoC). Valitud platvormid sobivad väitekirjas väljatöötatud meetodite rakendamiseks oma odavuse, paindlikkuse ja teiste eeliste poolest. Lisaks uuriti antud töös ka platvormide efektiivsust meetodite rakendamiseks.

Väitekirja põhitulemusteks on: kiire minimaalsete ja maksimaalsete sorteeritud alamhulkade leidmine suurtest andmehulkadest; andmetöötlusmeetodid, mis hõlmavad sorteerimist, operatsioonide ühendamist ja samaaegsete objektide loendamist; Hamming'u kaalu/kauguse loendurite/komparaatorite erilahenduse loomine; maatriksite/kogumite katmine; samuti loetletud tulemuste rakendamine, mis hõlmab riistvara/tarkvara koosprojekteerimist ja ümberkonfigureeritavate platformide kombineerimist.

PUBLICATION I

Sklyarov, V.; Skliarova, I.; **Rjabov, A.**; Sudnitson, A. (2013). Implementation of Parallel Operations over Streams in Extensible Processing Platforms. The 56th IEEE International Midwest Symposium on Circuits and Systems (IEEE MWSCAS 2013), Columbus, Ohio, USA, August 4-7, 2013. IEEE, 852−855.

Implementation of Parallel Operations over Streams in Extensible Processing Platforms

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

Artjom Rjabov, Alexander Sudnitson
Computer Department,
TUT,
Tallinn, Estonia
artjom.rjabov@gmail.com, alsu@cc.ttu.ee

***Abstract*—The paper discusses the use of extensible processing platforms for the design of high-performance systems which combine fast parallel operations over data streams in programmable logic and problem-specific software running in advanced RISC machine. The streams contain information that needs to be analyzed and filtered. The main idea is to digitalize frequently changed data from numerous sensors and to represent each data item in form of a binary vector. It is shown that many analysis and filtering problems can be solved through Hamming weight counting for the vectors and comparison of the results with pre-given bounds (thresholds). The proposed architecture takes advantages from fixed plus variable computations and implements novel methods. The results of experiments and evaluations of the architecture in two Zynq-based prototyping boards are also presented.**

I. INTRODUCTION

Combining capabilities of software and hardware permits many characteristics of developed applications to be improved. The earliest work in such direction was done at the University of California at Los Angeles [1]. 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. Today very similar technique has been implemented on a chip such as Xilinx Zynq xc7z020 extensible processing platform (EPP) [2]. A processing system (PS) (dual ARM® Cortex™-A9 MPCore™ of Zynq) executes software programs and C/C++ languages can be used to develop such programs. Programmable logic (PL) is FPGA Artix-7 [3] implemented on the same microchip with PS. PS and PL can exchange data using AXI-based (Advanced eXtensible Interface) high-bandwidth connectivity. Thus, in the same microchip we can implement and test: 1) systems requiring development of software and invoking on-chip processing block (i.e. PS); 2) application-specific hardware in programmable logic, i.e. PL (for such hardware we can also use soft processing cores, embedded blocks, such as DSPs and memories, and arbitrary logic composed of look-up tables and flip-flops); 3) *fixed+variable* structure computing that combines PS and PL with high-speed data exchange between PS and PL through AXI interface. Fig. 1 describes scenarios of interactions between PS and PL which we are going to discuss below.

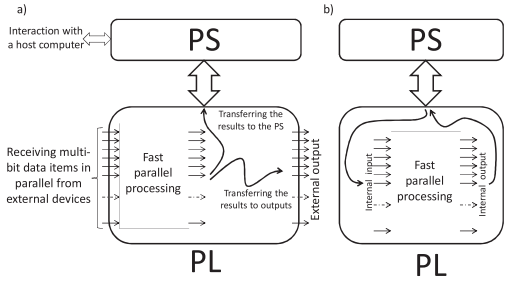

Figure 1. Interactions between PS and PL: PL provides fast parallel processing of external data and PS can use the results (a); PL executes dedicated operations for PS on internal requests from PS (b)

Two types of computations will be considered. In the first type (see Fig. 1a) PL functions as an autonomous system receiving data from external pins, executing operations over the data, and transferring the results either to external pins or to PS. In the second type (see Fig. 1b) PL is considered to be a slave sub-system of PS. As soon as PS needs to accelerate PL-dedicated operations, a request is sent to PL and data associated with the operations are transferred to PL. PL executes the operations using arbitrary logic and embedded components (such as DSPs). As soon as the operations are completed, PL informs PS that the results are ready and they are sent to PS. Both types of computations will be applied to stream processing. In the first type streams are received by PL from external sensors. In the second type a pre-processing is done in PS receiving data from other software programs or from the host computer connected through available input/output ports [2].

A brief summary of what is new and distinctive in this paper is given below:

978-1-4799-0066-4/13/$31.00 ©2013 IEEE

- Model of computations over streams in EPP combining PS and PL (sections II, III).
- Fast parallel look-up table based counters for Hamming weight computations (section IV).
- EPP-based implementation of the entire system and the results of experiments and comparisons (section V).

II. MODEL OF COMPUTATIONS OVER STREAMS

Many electronic, environmental, medical, and biological applications need to process data streams produced by sensors that measure external parameters within the given upper and lower bounds (thresholds). Examples of such measurements are monitoring thermal radiation from volcanic products [4], digital filtering [5], etc. Let us describe the problem in a way shown in Fig. 2a. A set of sensors $S_0,\ldots,S_{N-1}$ (the value N can achieve thousands) measure and output data. A set of data values (SDV) collected at the same time is represented in form of sub-sets such as that include: 1) values that are below the lower bound, 2) values that are between the upper and the lower bounds, and 3) values that are above the upper bound. The number of sub-sets can be just two (i.e. above and below the given threshold) or more than three (i.e. there are more than three groups in which we would like to analyze data values produced by the sensors). Dependently on applications SDV are generated with different frequency which might be very high (measured in megahertz). Thus, processing such SDVs has to be very fast and high-speed accelerators are greatly required. Let us look at the model depicted in Fig. 2b, where the interval of potential values is digitalized. If a measured value falls to a pre-given discrete interval a flag '1' is recorded in the corresponding binary vector, which is zero-filled at the beginning (see Fig. 2b).

In many practical cases we would like to analyze a distribution of potential values between different intervals (see Fig. 2c). For example, we would like to know how often volcanic activity [4] falls to a set of critical values, how the results of measurements in different (environmental, medical, biological) experiments are distributed, how well signals can be filtered in digital and signal processing, etc. Thus, we need to know how many active values fall in certain sub-sets.

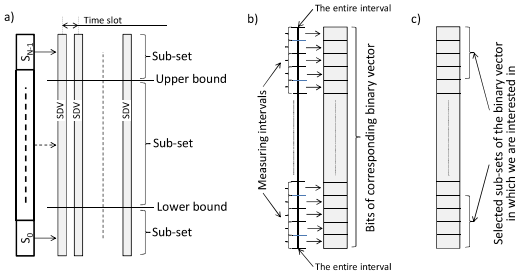

Figure 2. Data streams formed by sensors for different types of measurements (a), potential way to digitalize the measured values (b), selecting sub-sets of values which are needed for further analysis (c)

The Hamming weight w(A) of each sub-set A in Fig. 2c indicates its power (intensity of the relevant signals). Generally, the more the intensity the more critical is the subset. The more sub-sets have critical values the higher probability of an event which might happen (for example, scientists can conclude that there exist a high probability of a volcano eruption [4]). Analyzing the Hamming weights in different time slots (see Fig. 2a) or in different sub-sets (see Fig. 2b) permits to conclude when or where the activity of measured values is higher or lower (see Fig. 3a).

Measuring Hamming distances enables us to check repetitions of activities within the chosen time intervals. Discovering the maximum and the minimum values permits to know in which time (let us say of the day or night) the activity is the highest or the lowest (see Fig. 3b). Sorting of values enables charts showing activities during the chosen period of time to be built (see an example in Fig. 3c). All such details are required for monitoring and making conclusions or decisions in different indicated above areas.

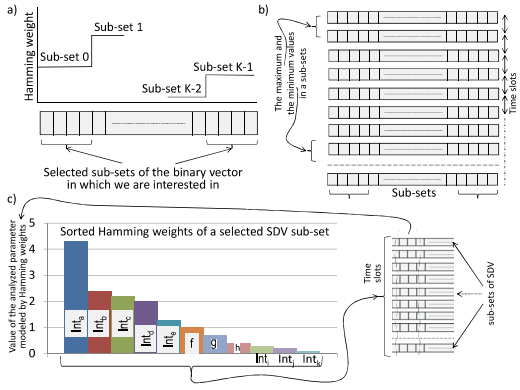

Figure 3. Analysis of Hamming weights in different sub-sets (a), discovering the maximum/minimum values in a sub-set (b), sorting the weights in a selected SDV sub-set (Int_i is a time slot with index i)

Filtering is a kind of processing shown in Fig. 2, 3. Suppose that we are interested only in SDVs with Hamming weights above (or alternatively below) of a given threshold κ and we would like to choose just such values for further processing. So, we need a digital filter which can be built using the Hamming weight counter/comparator. The considered above processing (see Fig. 2, 3) requires high performance computations. To rapidly find Hamming weights of long-size binary vectors, bits of such vectors need to be handled in parallel. Thus, FPGAs (PL of EPP) are very appropriate.

It is practical and efficient to combine processing in software and in hardware. For example, processing system might collect data from different sources and execute such operations over the received data that do not need highly parallel computations (e.g. monitoring and making conclusions). As soon as it is necessary to perform fast parallel operations (such as that are shown in Fig. 2, 3) the relevant data are transferred to PL operating as high-performance accelerator (see Fig. 1). The results are transferred back to the

PS. Programmable logic is faster for such operations as sort/search [6] (especially when we apply hardware-targeted methods [7]), Hamming weight computation/comparison [8] and many others [9].

III. Implementation of computations in EPP

Fig. 4 demonstrates the implemented system that involves both PS and PL. An interaction between PS and PL is organized with the aid of Xillybus Lite IP core [10].

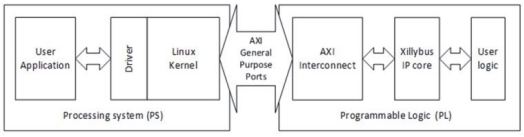

Figure 4. Implemented interaction between PS and PL

The user software applications run in ARM Cortex-A9 under Linux. The user (application-specific) logic is designed in Xilinx ISE 14.4 [11] (the details will be given in the next section) and interacts with the Xillybus IP core. The latter provides data exchange with PS through AXI running under Linux. Software applications were developed in C/C++ and they execute the following tasks: 1) getting data from the host PC and transmitting these data to PL; 2) getting and application-specific analysis of the results from PL; 3) support for experiments with the developed hardware in PL. User application-specific project in PL can be configured to support both types of computations shown in Fig. 1, i.e. initial stream is uploaded either from external pins or internally from PS through AXI. Currently we tested only Hamming weights/distance counters and comparators (see section IV) for streams represented in form of a binary matrix with K lines (modeled data items) containing N-bit vectors used for computations of Hamming weights or distances.

IV. Parallel LUT-based Humming weight counters

An FPGA LUT(n,m) can be used to directly implement arbitrary Boolean functions $f_0,\ldots,f_{m-1}$ of n variables $x_0,\ldots,x_{n-1}$. Clearly, h LUTs(n,m) can be configured to calculate the Hamming weight w(A) of a vector $A=\{a_0,\ldots,a_{n-1}\}$, where $h=\lceil(\log_2(n+1))/m\rceil$. The idea is to build a network from LUTs(n,m) that can find the Hamming weight w(A) for an arbitrary vector A of size N and then to compare this weight with either a fixed threshold κ or with the weight of another binary vector B assuming that the Hamming weight of B has been found similarly. Since Hamming distance d(A,B) = w(A XOR B) we can find d(A,B) as Hamming weights of "*XORed*" arguments A and B. The Hamming weight for a general vector (not obligatory binary) is defined as the number of its non-zero elements [8].

An analysis of practical applications shows that the majority of them require the Hamming weight/distance count/comparison for such values of N that are divisible by 8, 32, or 36. We suggest here two optimized LUT-based designs permitting the Hamming weight to be found for N=8 (Fig. 5a) and N=36 (Fig. 5b). For N=32 either four bits in Fig. 5b are assigned to 0 or the results of Fig. 5a are incrementally added in a tree-based structure much similar to [8] composed of the design in Fig. 5a and adders. The circuit in Fig. 5b without two right adders Σ (see Fig. 5b) has $\lceil(\log_2(n+1))/m\rceil\times(\lceil N/n\rceil+\lceil(N/2)/n\rceil)$ LUTs(n,m). Even for m=1 (the worst case) we need only 27 LUTs for Zynq xc7z020 containing totally 53200 LUTs.

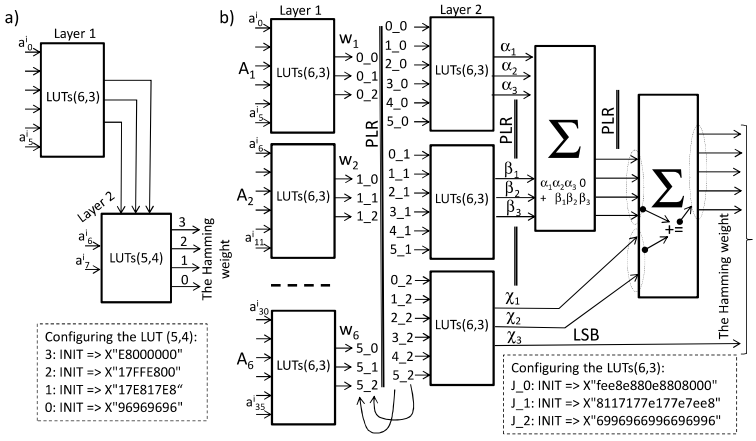

Figure 5. Hamming weight counters for N=8 (a) and N=36 (b)

The Hamming weight for N>36 can be found in a similar tree-based structure. There are two layers in Fig. 5a with LUTs(6,3) and LUTs(5,4). The first layer counts $w(a^i_0,\ldots,a^i_5)$ and the second layer takes the results of the first layer and finally determines the 4-bit weight $w(a^i_0,\ldots,a^i_7)$. The delay from the inputs to the outputs is equal to just 2 LUT delays. There are also two layers in Fig. 5b with LUTs(6,3) and two combinational adders. The first layer is composed of 6 LUTs(6,3) and it outputs six Hamming weights $w_1,\ldots,w_6$ for six sub-vectors $A_1,\ldots,A_6$ of the input vector. The second layer

contains 3 LUTs(6,3) and it outputs Hamming weights $\alpha_1\alpha_2\alpha_3$, $\beta_1\beta_2\beta_3$, $\chi_1\chi_2\chi_3$ of the most significant bits (MSB) in $w_1,\ldots,w_6$ ($\alpha_1\alpha_2\alpha_3$), the middle bits in $w_1,\ldots,w_6$ ($\beta_1\beta_2\beta_3$) and the less significant bits (LSB) in $w_1,\ldots,w_6$ ($\chi_1\chi_2\chi_3$). The final result is computed by two combinational adders as it is shown in Fig. 5b. We found that any layer with index greater than $\lceil \log_n N \rceil$ is not cost-effective because either the size of output weights will be increased compared to the previous layers or LUTs will be used not-efficiently. All LUTs in Fig. 5b are configured identically (see INIT statements in Fig. 5b for configuring LUTs in Xilinx ISE 14.4 environment [11]).

V. Experiments and Comparisons

The charts in Fig. 6 permit to compare the suggested architectures with the best known alternatives, such as [8,12,13]. All the circuits were synthesized, implemented in the Xilinx Zynq xc7z020 microchip, and tested in two prototyping boards: 1) Xilinx Zynq-7000 EPP ZC702; and 2) ZedBoard.

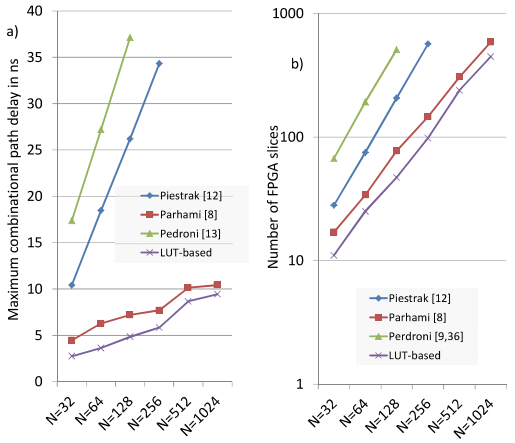

Figure 6. Latency (a) and cost (b) comparison

The first chart (Figure 6a) shows the maximum combinational path delay and the second chart indicates the number of FPGA slices for different designs. The total number of available slices in the microchip xc7z020 is 13 300. For our circuits we also considered pipelined implementations which include additional registers between layers (see PLR in Fig. 5). We found that the maximum delay between the registers can be as little as 1.253 ns. Thus, potential throughput can be less than 2 ns per weight.

The project that includes interactions between PL and PS (see Fig. 5) was entirely implemented and tested. It was used to process 252-bit binary vectors supplied from different sources and for subsequent analysis of the results in software. The block "User logic", shown in Fig. 4, implements the Hamming weight/distance counters/comparators with the tree-based structure composed of 7 blocks depicted in Fig. 5b and the final adders that were built from two embedded to xc7z020 DSP slices 48E1 [14]. There are totally 220 DSP slices in Zynq xc7z020 microchip. Each slice was configured as 4 independent 12-bit adders with blocked internal carry propagation between the adders to ensure independent addition operations [14]. The "User logic" block (see Fig. 4) can easily be replaced with a new block for implementing other operations shown in Fig. 3. Thus, the system is reusable and can be seen as a very good base for further experiments.

VI. Conclusion

The paper suggests novel methods and a reusable system for analysis of streams involving Hamming weight/ distance counters/comparators. The system is reusable and it is composed of problem-specific software running in advanced RISC machine and a hardware accelerator mapped to programmable logic. The results of experiments confirm correctness and effectiveness of the proposed technique.

Acknowledgment

This research was supported by FCT (Portugal), the EU through the European Regional Development Fund and by the Estonian Science Foundation Grant No. 9251.

References

[1] G. Estrin, "Organization of Computer Systems – The Fixed Plus Variable Structure Computer", Proc. Western Joint Computer Conf., New York, 1960, pp. 33-40.

[2] M. Santarini. Zynq-7000 EPP Sets Stage for New Era of Innovations. *Xcell journal*, issue 75, second quarter, 2011.

[3] Zynq-7000 All Programmable SoC First Generation Architecture: http://www.xilinx.com/support/documentation/data_sheets/ds188-XA-Zynq-7000-Overview.pdf.

[4] L. Field, T. Barnie, J. Blundy, R.A. Brooker, D. Keir, E. Lewi, K. Saunders, Integrated field, satellite and petrological observations of the November 2010 eruption of Erta Ale, Bull Volcanol (2012) 74:2251–2271.

[5] P. D. Wendt, E. J. Coyle, and N. C. Gallagher, "Stack filters," IEEE Trans. on Acoust., Speech, Signal Processing, vol. 34, no. 4, pp. 898-908, Aug. 1986.

[6] R. Mueller, J. Teubner, G. Alonso, Sorting Networks on FPGAs, The International Journal on Very Large Data Bases, vol. 21, no. 1, 2012, pp. 1-23.

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

[8] B. Parhami, Efficient Hamming Weight Comparators for Binary Vectors Based on Accumulative and Up/Down Parallel Counters, IEEE Trans. on Circuits and Systems—II: Express Briefs, vol. 56, no. 2, pp. 167-171, Feb. 2009.

[9] D.G. Bailey, Design for Embedded Image Processing on FPGAs, John Wiley and Sons, 2011.

[10] Xillybus Lite for Zynq-7000: Easy FPGA registers with Linux. Available at http://xillybus.com/xillybus-lite.

[11] ISE 14.4 design tools and documentation. Available at: www.xilinx.com.

[12] S.J. Piestrak, Efficient hamming weight comparators of binary vectors, Electronic Letters, vol. 43, no. 11, pp. 611–612, May 2007.

[13] V. Pedroni, Compact Hamming-comparator-based rank order filter for digital VLSI and FPGA implementations, Proc. of the IEEE International Symposium on Circuits and Systems - ISCAS'2004, pp. 585–588.

[14] Xilinx, "7 Series DSP48E1 Slice User Guide", 2012.

PUBLICATION II

Skliarova, I.; Sklyarov, V.; **Rjabov, A.**; Sudnitson, A. (2014). Fast Matrix Covering in All Programmable Systems-on-Chip. Elektronika ir Elektrotechnika, 20 (5), 150−153.

Fast Matrix Covering in All Programmable Systems-on-Chip

V. Sklyarov[1], I. Skliarova[1], A. Rjabov[2], A. Sudnitson[2]
[1]*Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, 3810-193 Aveiro, Portugal*
[2]*Department of Computer Engineering, Tallinn University of Technology, 12617 Tallinn, Estonia*
skl@ua.pt

[1]***Abstract*—The paper suggests a technique for solving the matrix/set covering problem in all programmable systems-on-chip. A novel very fast hardware accelerator is proposed and implemented in the programmable logic (PL) of a Xilinx Zynq microchip. The accelerator is managed by software running in the processing system (ARM Cortex-A9) available on the same microchip and communicating with the PL through high-speed interfaces. The results of implementation, experiments, and comparisons demonstrate significant speedup comparing to software running in general-purpose PC and in the ARM.**

***Index Terms*—Accelerator architectures, concurrent computing, parallel processing, field programmable gate arrays, system-on-chip.**

I. INTRODUCTION

Combinatorial search algorithms are frequently involved to solve optimization problems. Examples are matrix/set covering, the Boolean satisfiability, graph coloring and many others described and reviewed in [1]–[4]. Many tasks are NP-complete and, thus, they are time consuming. We consider here the matrix/set covering which belongs to partitioning problems [1] arising in such practical applications as scheduling aircrafts, location emergency stations in urban areas, fault testing of electronic circuits, resource distribution in multi-core systems, and many others [1]. For many applications high performance is required and it may be achieved in hardware accelerators for which FPGA-based solutions are especially promising. It is shown and proved in the paper that recently appeared on the market all programmable systems-on-chip (APSoC) of Xilinx Zynq family [5] are very appropriate for implementation of combinatorial search algorithms enabling the problem to be decomposed into two sub-problems that are 1) higher-level activation of primary sub-tasks in which the algorithm has been decomposed, and 2) fast execution of the sub-tasks in the hardware accelerator. According to the proposals, the first sub-problem is assigned to a processing system (PS) implemented on the basis of industry-standard dual-core ARM Cortex-A9 in Zynq APSoC. The acceleration is done in a programmable logic - PL (Xilinx Artix-7 FPGA) that is available on the same microchip with the ARM. It is shown that such type of hardware/software co-design permits elegant and efficient solutions to be found that are faster than the best known alternatives.

The remainder of the paper is organized in six sections. Section II defines the problem and presents an example. Section III suggests architecture of the hardware accelerator. Section IV is dedicated to software/hardware co-design. Experimental setup is discussed in Section V. The results and comparisons are reported in section VI. The conclusion is given in Section VII.

Manuscript received October 28, 2013; accepted January 6, 2014.

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

II. PROBLEM DEFINITION

The covering problem can identically be formulated on either sets [1], [2] or matrices [1]. Let $\mathbf{A} = (a_{ij})$ be a 0-1 incidence matrix. The sub-set $A_i = \{j \mid a_{ij} = 1\}$ contains all columns covered by row i (*i.e.* the row i has value 1 in all columns of the sub-set A_i). The minimal row cover is composed of the minimal number of the sub-sets $\mathbf{A}_i$ that cover all the matrix columns. Clearly, for such sub-sets there is at least one value 1 in each column of the matrix. Let us consider an example from [2] of a set S and sub-sets $S_1,\ldots,S_6$ (Fig. 1), which can be represented in the form of the following matrix **A**:

	1	2	3	4	5	6	7	8	9	10	11	12
S_1:	1	1	0	0	1	1	0	0	1	1	0	0
S_2:	0	0	0	0	0	1	1	0	0	1	1	0
S_3:	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
S_4:	*0*	*0*	*1*	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*
S_5:	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*
S_6:	0	0	0	1	0	0	0	1	0	0	0	0

Different algorithms have been proposed to solve the covering problem [1]–[3], such as greedy heuristic [1], [2] and a very similar method [3]. An analysis of the known algorithms has shown the following:

1. The majority of them are approximate since the problem is NP-complete;

2. The algorithms are very similar and they differ insignificantly;
3. The algorithms may equally be applied to either sets or matrices and any of such models can be chosen because they are directly convertible to each other.

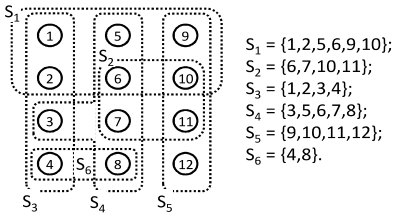

Fig. 1. An example of a set S with sub-sets S_1,...,S_6 from [2].

We consider below a slightly modified method from [3] that is applied to binary matrices exemplified above and the matrix from Fig. 1 [2] will be used to illustrate the steps of the chosen method that are the following:

1. Finding the column C_{min} with the minimum Hamming weight (HW) that is the number of ones. If there are many columns with the same (minimum) HW, selecting such one for which the maximum row is larger, where the maximum row contains 1 in the considered column and the maximum number of ones;
2. If HW = 0 then the desired covering does not exist, otherwise from the set of rows containing ones in the column C_{min} finding and including in the covering the row R_{max} with the maximum HW;
3. Removing the row R_{max} and all the columns from the matrix that contain ones in the row R_{max}. If there are no columns then the covering is found otherwise go to the step 1.

Let us apply the step 1–3 to the matrix **A** above:

1. The column 12 is chosen;
2. The row S_5 is included in the covering;
3. The row S_5 and the columns 9, 10, 11, 12 are removed from the matrix.
1. The remaining columns contain the following number of ones: 2, 2, 2, 2, 2, 3, 2, 2. The column 3 is chosen because for this column the row S_4 has the maximum HW equal to 5;
2. The row S_4 is chosen and included in the covering;
3. The row S_4 and the columns 3, 5, 6, 7, 8 are removed from the matrix.
1. The remaining matrix contains rows S_1, S_2, S_3, S_6 and columns 1, 2, 4 with the following HWs: 2, 2, 2. The column 1 is chosen;
2. The row S_3 is chosen and included in the covering;
3. After removing the row S_3 the covering is found and it includes the rows S_3,S_4,S_5 shown in italic font in the matrix above. The minimum covering is the same as in [2] that was found with a different algorithm.

III. ARCHITECTURE OF HARDWARE ACCELERATOR

This section presents the proposed architecture of the hardware accelerator executing the steps 1 and 2 from Section II. We suggest the given matrix to be unrolled in such a way that all its rows and columns are saved in the PL registers. Note that more than a hundred of thousands of such registers are available in the recent low-cost FPGAs. This technique permits all rows and columns to be accessed and processed in parallel.

Figure 2 demonstrates the unrolled matrix **A** shown above in Section II (and repeated in Fig. 2 for convenience). HW counters compute HW for all the rows/columns in parallel using combinational circuits, such as that are proposed in [6], [7]. These circuits are very fast allowing HWs to be computed in less than 20 ns even in low-cost FPGAs.

The MIN column and MAX row circuits permit to find out the minimal column C_{min} and the maximum row R_{max}. It is shown in [8] that these circuits can be built as MAX-MIN fully combinational networks producing the results faster than in 20 ns. Since all the circuits (computing HW and the maximum/minimum values) are functioning in parallel, the steps 1 and 2 may be completed faster than in 20 + 20 = 40 ns even in low-cost FPGAs. So, a very significant acceleration can be expected.

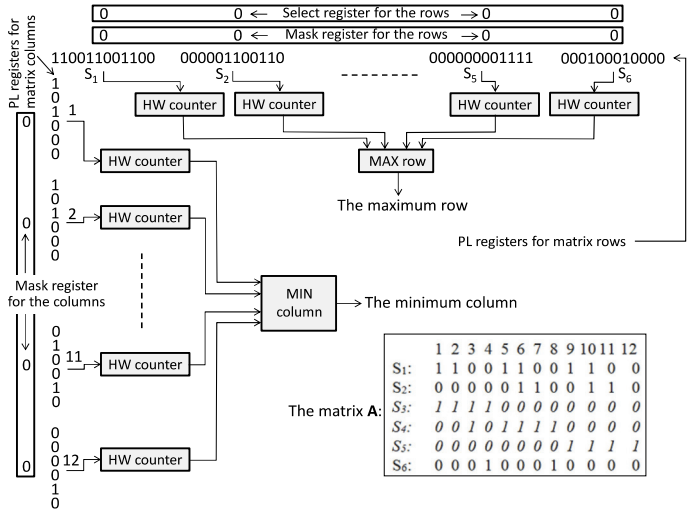

Fig. 2. Architecture of the proposed hardware accelerator on an example of unrolled matrix A from Section II.

In accordance with the proposals, the matrix is unrolled only once and any reduced matrix is formed by masking previously selected rows and columns. One select register and two mask registers (one for rows and another one for columns) shown in Fig. 2 are additionally allocated in the PL. The select register is zero-filled at the beginning of the step 1 and after the step 1 it indicates by values 1 those rows that have to be chosen by the selected column (i.e. such rows have values 1 in the selected column). The mask registers are filled in with zeroes at the beginning of the algorithm and they mask (by the values 1) those rows and columns that have been removed from the matrix in each iteration. For example, the select register contains the value 000010 after the first step in the example of Section II. The mask registers after the first iteration in the example are set to 000000001111 for the columns and 000010 for the rows. After the second iteration they are updated as 001011111111 for the columns and 000110 for the rows.

IV. SOFTWARE/HARDWARE CO-DESIGN

Figure 3 presents the proposed partitioning in software and hardware modules (assuming implementation in Zynq APSoC) of the considered algorithm that enables the minimal covering to be found.

Software in the PS is responsible for the following steps:

1. Getting from a host computer or generating the matrix, unrolling it, and saving in external DDR memory as a set of rows and a set of columns;
2. As soon as C_{min} is found, the PL generates an interrupt of type *a*. The PS receives the C_{min} and sets the select register in the PL through general-purpose ports [5];
3. As soon as R_{max} is found, the PL generates an interrupt of type *b*. The PS receives the R_{max} and sets the mask registers in the PL through general-purpose ports [5];
4. At any iteration it is checked if the solution is found or if it does not exist. If the solution is found it is indicated by the PS or transmitted to the host computer and the algorithm is completed.

Hardware in the PL implements the architecture in Fig. 2 and is responsible for the following steps:

1. Getting the unrolled matrix from external DDR through high-performance Advanced eXtensible Interface (AXI) [5] and saving the rows and columns in slice registers as it is shown in Fig. 2.
2. Getting from the PS select/mask vectors and setting/updating the select and the mask registers.
3. Finding out the value C_{min} at each iteration and as soon as the value of C_{min} is ready, generating an interrupt of type *a*.
4. Finding out the value R_{max} at any iteration and as soon as the value of R_{max} is ready, generating an interrupt of type *b*.

V. EXPERIMENTAL SETUP

Implementation was done in the Xilinx Zynq-7000 APSoC ZC702 evaluation kit [9] containing a microchip (APSoC) Zynq xc7z020. The PS is the dual-core ARM Cortex-A9 and the PL is Artix-7 FPGA from the 7th series of Xilinx. Currently only AXI, general-purpose ports and interrupts have been used (from 16 available interrupts we selected only two assigned above as type *a* and *b*). Software for the ARM was developed in C language and hardware for the PL was synthesized from specification in VHDL. Computing HW was done in LUT-based circuits from [7] that are very economical and fast. Experiments were done with two types of matrices 32 × 32 and 64 × 64. Thus, either 32 + 32 = 64 or 64 + 64 = 128 HW counters have been implemented in the PL section and all these circuits can run in parallel. Since C_{min} and R_{max} are found at different steps in the current implementation, only half of the HW counters work in parallel enabling either the minimal column C_{min} or the maximal row R_{max} to be found.

Because the occupied resources are indeed very small [7], we implemented all the required HW counters assuming that in future improvements all of them might function in parallel. The MIN and MAX circuits are built as combinational networks and they are described in detail in [8].

Figure 4(a) presents such a circuit for a matrix 32 × 32 for which the number of bits in any HW is 6 (because the maximum number of ones in a 32-bit vector is 32 that can be represented by a 6-bit code).

A particular (simplified) example for only 6 input items 3, 14, 21, 11, 14, 27 is given in Fig. 4(b). The maximum value (27) is found in a combinational circuit with only 3 gate level delays. Clearly, there is 5 gate level delay for matrices 32 × 32 and 6 gate level delay for matrices 64 × 64.

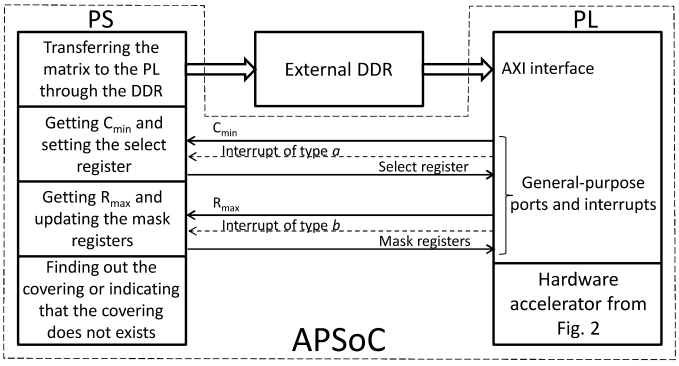

Fig. 3. Partitioning of the algorithm in software and hardware modules.

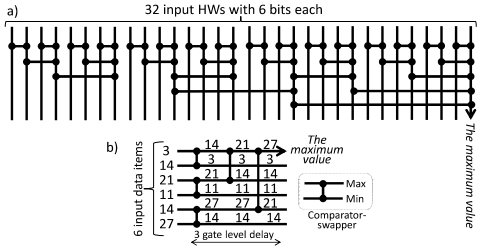

Fig. 4. MAX circuit from [8] for 32 × 32 matrix (a); an example (b).

VI. RESULTS AND COMPARISONS

We compared three different implementations in which the covering algorithm is either:

1. Described in C language program running in PC with Intel i7 2.66 GHz processor;
2. Described in C language program running in ARM Cortex-A9 (for evaluation kit [9]);
3. Implemented in the PS and in the PL of Zynq-7000 APSoC as it is shown in Fig. 3 (for evaluation kit [9]).

Initial matrices have been generated randomly using the C *rand* function and identically for all the described above implementations. The number of instances (examples) was chosen to be 100,000.

In the last case (see the point 3 above) that is the original contribution of the paper, the following results have been obtained:

1. Generating in the PS and transmitting the matrices from the PS to the PL requires about 31 μs for 32 rows and 32 columns and about 34 μs for 64 rows and 64 columns. Only one AXI 32-bit (for the matrices 32 × 32) or 64-bit (for the matrices 64 × 64) port from the 4 available ports has been used. Clearly, additional ports permit the indicated time to be reduced;
2. Each iteration in the PL is executed in about 28 ns for the matrices 64 × 64 and about 24 ns for the matrices 32 × 32;
3. Communications between the PS and the PL (through interrupts and general-purpose ports) at any iteration of the algorithm require negligible time comparing to other operations.

The covering is found significantly faster than in software. The acceleration comparing with the PS only (see point 2 above) is from 30 to 50 times and comparing with the PC (see point 1 above) is from 5 to 10 times. This is because operations of the covering algorithm in software require many cycles and frequent transmission of data between processors and memories. For example, if we consider 64 × 64 matrices then a single matrix transfer from the PS to the DDR takes 33,300 ns on average and this is the most time consuming operation. Data transfer from the DDR to the PL is done in 284 ns on average. Once the PL receives the matrix data, no more interaction with the DDR is required for further processing.

For future work we will use accelerator coherency ports available for Zynq microchips and allowing data exchange directly with the processor cache memory [10], [11]. Besides, an additional optimization technique will be provided looking for the better distribution of different sub-tasks between the PS and the PL.

VII. CONCLUSIONS

The paper presents a novel technique for implementation of matrix/set covering algorithms in hardware and software of recent all programmable systems-on-chip. A new method that permits the known approximate algorithm to be executed over suggested unrolled matrices is discussed and the relevant hardware accelerator is developed. It is shown that the covering algorithm can efficiently be partitioned in software and hardware modules that finally have been completely implemented and tested in Xilinx Zynq microchips. The results of experiments and comparisons with two different software implementations demonstrate significant speedup which is very important for various practical applications that are also mentioned in the paper.

REFERENCES

[1] K. H. Rosen, J. G. Michaels, J. L. Gross, J. W. Grossman, D. R. Shier, *Handbook of Discrete and Combinatorial Mathematics*. Boca Raton, FL: CRC Press, p. 1232, 2000.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*. USA: MIT Press, p. 1312, 2009.

[3] A. Zakrevskij, Y. Pottosin, L. Cheremisiniva. *Combinatorial Algorithms of Discrete Mathematics*. Tallinn: TUT Press, p. 193, 2008.

[4] I. Skliarova, A. B. Ferrari, "Reconfigurable hardware SAT solvers: a survey of systems", *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1449–1461, 2004. [Online]. Available: http://dx.doi.org/10.1109/TC.2004.102

[5] *Zynq-7000 All Programmable SoC First Generation Architecture*, Xilinx Inc., USA, 2012. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds188-XA-Zynq-7000-Overview.pdf

[6] V. Sklyarov, I. Skliarova, "Design and implementation of counting networks", *Computing*, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00607-013-0360-y

[7] V. Sklyarov, I. Skliarova, "Digital Hamming weight and distance analyzers for binary vectors and matrices", *Int. Journal of Innovative Computing, Information and Control*, vol. 9, no. 12, pp. 4825–4849, 2013. [Online]. Available: http://www.ijicic.org/ijicic-12-12021.pdf

[8] V. Sklyarov, I. Skliarova, "Fast regular circuits for network-based parallel data processing", *Advances in Electrical and Computer Engineering*, vol. 13, no. 4, pp. 47–50, 2013. [Online]. Available: http://dx.doi.org/10.4316/AECE.2013.04008

[9] *Zynq-7000 EPP ZC702 evaluation kit*, Xilinx Inc., USA, 2014. [Online]. Available: http://www.xilinx.com/products/boards-and-kits/EK-Z7-ZC702-G.htm

[10] M. Sadri, C. Weis, N. When, L. Benini, "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ", *Proc. 10th FPGAWorld Conf.*, Stockholm, Sweden, 2013. [Online]. Available: http://dx.doi.org/10.1145/2513683.2513688

[11] D. Mihhailov, A. Sudnitson, V. Sklyarov, I. Skliarova, "Acceleration of Recursive Data Sorting over Tree-based Structures", *Elektronika ir Elektrotechnika*, no. 7, pp. 51–56, 2011. [Online]. Available: http://dx.doi.org/10.5755/j01.eee.113.7.612

PUBLICATION III

Rjabov, A.; Sklyarov, V.; Skliarova, I.; Sudnitson, A. (2015). Processing Sorted Subsets in a Multi-level Reconfigurable Computing System. Elektronika ir Elektrotechnika, 21 (2), 30−33, 10.5755/j01.eee.21.2.11509.

Processing Sorted Subsets in a Multi-level Reconfigurable Computing System

Artjom Rjabov[1], Valery Sklyarov[2], Iouliia Skliarova[2], Alexander Sudnitson[1]
[1]*Department of Computer Engineering, Tallinn University of Technology, Tallinn, Estonia*
[2]*Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, Aveiro, Portugal*
artjom.rjabov@gmail.com

***Abstract*—The paper suggests a technique for extracting and filtering sorted subsets in a three-level computing system with such sub-systems as general-purpose computer (level 1), ARM Cortex-A9 (level 2), and reconfigurable logic (level 3). The last two levels are implemented in Zynq-7000 device available on the prototyping board ZC706. Communications between the levels 1 and 2-3 are organized through PCI express bus and interactions between components of levels 2 and 3 - through on-chip AXI interfaces. We studied two levels of software programs (running in PC and ARM), high-performance hardware accelerators implemented in Zynq-7000 programmable logic, and architecture enabling interactions and exchange of data between different levels. The selected for analysis sorting problem has high computational complexity and is widely required in data processing (data mining and statistical data manipulation, in particular). The results of experiments demonstrate that the elaborated architecture is efficient and permits fast solutions to be found. Proposals for potential further improvements are also given.**

***Index Terms*—Computing sorted subsets, communicating software/hardware systems, sorting networks, filtering, programmable systems-on-chip.**

I. INTRODUCTION

Many practical applications require acquisition, analysis, and filtering of large data sets. Let us consider some examples. In [1] a data mining problem is explained 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 appear together in a basket either a maximum or a minimum number of times within the database [1]. Similar problems appear to determine frequent queries at the Internet, customer transactions, credit card purchases, etc. requiring processing very large volumes of data in the span of a day [1]. Fast extraction of the most frequent or the less frequent items from large sets permits data mining algorithms to be accelerated and may be used in many known methods from this scope, *e.g.* [2]–[4]. Another example can be taken from the area of control. Applying the technique [5] in real-time applications requires knowledge acquisition from the controlled systems. For example, signals from sensors may be filtered and analysed to prevent error conditions [5]. To provide more exact and reliable conclusion, combination of different values need to be extracted, ordered, and analysed. Similar tasks appear in monitoring thermal radiation from volcanic products [6], filtering and integrating information from a variety of different sources in medical applications [7] and so on.

Since many systems have hard real-time constraints, performance is important and hardware accelerators may provide significant assistance for software products (such as [5]). Similar problems appear in so-called straight selection sorting (in such applications where we need to find the task with the shortest deadline in scheduling algorithms [8]).

The paper suggests a new method to design high-performance accelerators based on all programmable systems-on-chip (APSoC) from the Xilinx Zynq-7000 family [9] communicating with a general-purpose computer through PCI express bus. APSoCs are recently developed field-configurable devices integrating the most advanced programmable logic (PL) and a widely used processing system (PS): the dual-core ARM® Cortex™ MPCore™. The available interfaces between the PS and PL are supported by ready-to-use intellectual property (IP) cores. These, combined with numerous architectural and technological advances, have enabled APSoCs to open a new era in the development of highly optimized computational systems [10].

The remainder of the paper is organized in four sections. Section II describes the problem and suggests an architecture of a 3-level system. Section III considers different modes of functionality of hardware accelerators. Section IV reports the results of experiments and compares them with alternative computations in general-purpose software. The conclusion is given in Section V.

II. PROBLEM DEFINITION AND SYSTEM ARCHITECTURE

Let A be a set of data items that can be of any predefined type common for general-purpose languages (*e.g.* integer). We consider here such computations that permit:

Manuscript received November 22, 2014; accepted January 16, 2015.
This research was supported by the European Union through the European Regional Development Fund, the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, the Estonian Science Foundation Grant No. 9251, and Portuguese National Funds through FCT - Foundation for Science and Technology, in the context of the project PEst-OE/EEI/UI0127/2014.

– Extract subsets of A containing L_{max} (L_{min}) items with the maximum (the minimum) values;

– Extract subsets containing filtered values of A that fall within the given upper (u) and lower (l) bounds.

The set A can be very large and we would like to execute the computations indicated above as fast as possible.

The proposed system architecture combines the following three levels (Fig. 1):

1. Software of a host computer (such as PC) developed in a general-purpose programming language (*e.g.* C/C++ or Java). Since such software has a number of known constraints (such as the maximum number of parallel threads, and architecture-specific limitations) we would like to develop a more flexible and parallel acceleration system taking advantages of field-programmable technology.
2. APSoC PL enabling broad parallelism to be provided and eliminating architectural constraints (i.e. the most appropriate accelerator architecture can be proposed and realized).
3. APSoC software that permits interactions between different levels to be simplified and optimized with the aid of available efficient IP cores.

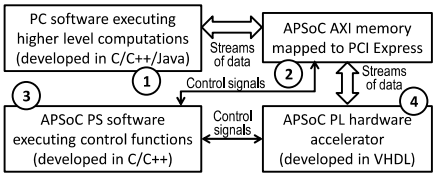

Fig. 1. Elaborated architecture.

In the proposed designs software (in the host PC and in the PS of APSoC) is running under Linux operating system. The following functionality (Fig. 1) is provided:

– As soon as some acceleration is needed, the program (in the host PC, see block 1) copies data from the set A through PCI express bus to DDR memory (see block 2) communicating with the APSoC (see blocks 3, 4) and controlled by the APSoC (ZC706 prototyping system [11] of Xilinx will be used in further experiments).

– As soon as data are transferred to the DDR, an interrupt (see block 1) is generated and handled in the APSoC PS (block 3). C/C++ function, that handles the interrupt in the PS, requests the acceleration operation in the PL and supplies necessary data (such as the number and the size of items in the received set A: see blocks 3 and 4) through AXI (Advanced eXtensible Interface [9]) GP (general-purpose [9]) port. Basic functionality of the function that handles interrupts is similar to [12].

– The PL accelerator (see block 4) executes highly parallel operations over the set A and copies the extracted subset to the same DDR memory.

– As soon as all items that form the result are transferred to the DDR, the PL generates an interrupt to the PS (see blocks 3 and 4) which is handled in the PS software.

– Interrupt handler in software of the PS sets a special flag indicating that the requested acceleration operation has been completed (see blocks 1 and 3). The flag is tested in the PC software and as soon as it is set, the resulting data are copied to the PC (see blocks 1 and 2).

Configuration of the APSoC, specifying the requested acceleration operation such as finding the minimum/maximum subsets or filtering using bounds (and consequently enabling the required operation to be chosen), is done before the execution time. It is also possible to choose operations during run-time providing necessary details from the PC to the PS and further to the PL. Data exchange between different sub-systems (PC, DDR memory, PS and PL) is initiated as follows (Fig. 2):

1. PC/PS (memory): a) software of the host PC executes C library function `memcpy` which copies data from the set A (kept in the host PC memory) to the DDR memory through the following blocks: Xilinx IP core for working with PCI express [13] (see the block AXI memory mapped to PCI express), AXI interconnect and PS memory controller (Fig. 2); b) software of the host PC generates an interrupt (through additional `memcpy` function) indicating completion of data transfer and handled in the PS (see PCI Control Unit and interrupt IRQ in Fig. 2).
2. Memory, PS/PL: a) software of the APSoC PS transfers control signals to the PL through an AXI GP master port using `Xil_Out32` function of Xilinx [14] (see GP Control Unit in Fig. 2); b) software of the APSoC PS sends a request to the PL (once again through an AXI GP master port) to execute the chosen operation.
3. The PL carries out the indicated operation getting blocks of data from the DDR memory and transferring the results to the DDR memory through AXI high-performance (HP) ports (see HP Control Unit and interrupt IRQ in Fig. 2).
4. When the results are ready and copied by the PL to the DDR, the PL generates an interrupt handled by the PS.
5. Interrupt handler in the PS sets the flag for the host PC (see PCI Control Unit in Fig. 2).
6. The PC transfers the resulting subset using `memcpy` function and the Xilinx IP core for working with PCI express.

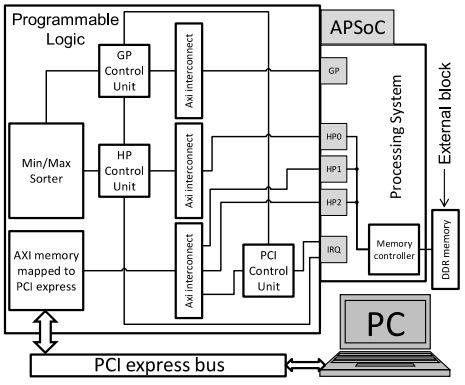

Fig. 2. Interactions between different system components.

III. FUNCTIONALITY OF THE HARDWARE ACCELERATOR

Let N be the number of elements in the given set A. We consider such tasks for which $L_{max} << N$ and $L_{min} << N$ which are more common for practical applications. Accelerating circuits implement partial sort that is done in highly parallel networks [15]. Since N may be large, it

cannot be processed completely in hardware due to the lack of sufficient resources.

We suggest solving the problem iteratively using hardware architecture shown in Fig. 3. Data are incrementally received in blocks containing up to K items and then processed by the sorting circuits [15] which iteratively execute many parallel operations and can be applied for significantly larger number of data items within the same hardware than other known sorting networks. The proposed method enables sorted subsets to be incrementally constructed as follows:

1. At initialization step (preceding the execution step) the maximum and the minimum subsets are filled in with the minimum and the maximum values as it is shown in Fig. 3.
2. Blocks with data items are sequentially supplied to the inputs of the sorting circuits located between the circuits which compute the maximum and the minimum subsets. All data items from one block are supplied in parallel. A new block arrives only when all data items from the previous block are processed (i.e. items, which satisfy criteria of the bottom block go down and items, which satisfy criteria of the upper block go up as it is shown in Fig. 3).
3. As soon as all the blocks (in which the set A is decomposed) are processed, the maximum and the minimum subsets are ready and they are transferred to the DDR memory.

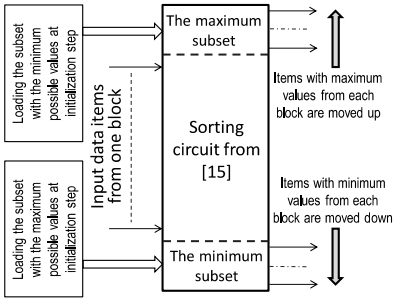

Fig. 3. Basic structure of the hardware accelerator.

It is easy to show that the circuit in Fig. 3 permits very large sets A to be processed. The sizes of the minimum/maximum subsets and the size of the blocks may vary (the details are given in the next section).

If data items need to be filtered then the circuit shown in Fig. 4 is invoked. Now we would like to use the circuit in Fig. 3 only for such data items that are within the bounds l and u. The circuit in Fig. 4 enables data items to be filtered in real-time (i.e. during data exchange between the PL and the DDR memory). The block "*l* and/or *u*" admits only those data items from AXI HP port that fall within the given bounds l/u. If and only if the item I_k is admitted, the address counter is incremented and the write enable (WE) signal is asserted allowing the value I_k to be written to the input register with the number chosen by the address counter. Data items from the input registers are inputs of the circuit shown in Fig. 3.

The filtered values may be: a) sent back to the DDR memory; b) sorted using the projects from [12]; c) used to extract the minimum/maximum subsets (Fig. 3).

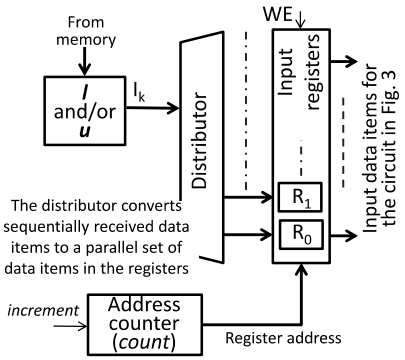

Fig. 4. Filtering circuit.

Note that additional circuits (such as problem-targeted control finite state machines) are needed for executing operations in Fig. 3 and Fig. 4. They are implemented similarly to [12], [15], [16].

IV. EXPERIMENTS AND COMPARISONS

Experiments were done with the Xilinx ZC706 prototyping system [11] containing the Zynq-7000 XC7Z045 APSoC device with PCI express endpoint connectivity "Gen1 4-lane (x4)". The PS is the dual-core ARM Cortex-A9 and the PL is Kintex-7 FPGA from the Xilinx 7th series. In all the experiments data from the set A are generated in the host PC randomly and the results are analyzed and verified also in the host PC. The size of data was chosen to be 256 KB. In the experiments the host PC generates 32-bit integer values and 64K (65,536) of 32-bit words are processed. The values of L_{min}/L_{max} varied from 128 to 1024 bytes (i.e. from 32 to 256 32-bit words).

A similar task was also solved in software only of the host PC where data from the set A were preliminary sorted and then the maximum and minimum subsets for different values of L_{min}/L_{max} were extracted. The bound values (L_{min}/L_{max}) in the host PC almost do not influence the results because the time is mainly consumed by the sort function. The following simple Java code was used:

```
long time=System.nanoTime();
   Arrays.sort(A);
long time_end=System.nanoTime();
```

where A is an array representing the set A. The array is generated randomly as:

```
for(int i = 0; i < A.length; i++)
      A[i] = rand.nextInt(Integer.MAX_VALUE);
```

The time is measured as (**double**) (time_end - time)/1000000. μs. Sorting 64 K of 32-bit data in PC with i7-4770 CPU 3.4 GHz and 8 GB of RAM requires approximately 18,000 μs. Similar results were also obtained for C/C++ programs running in the same PC. Transferring 256 KB from PC to DDR memory requires approximately 1,800 μs. Table I indicates the time consumed in the APSoC for extracting subsets with different number of data items and the resulting acceleration (taking into account the indicated above communication overhead of 1,800 μs).

TABLE I. THE CONSUMED TIME BY THE DEVELOPED HARDWARE ACCELERATOR AND ACCELERATION ACHIEVED COMPARED TO GENERAL-PURPOSE SOFTWARE RUNNING IN PC WITH MULTICORE I7 PROCESSOR AND 8 GB OF RAM.

L_{min} and L_{max}	The consumed time in μs	Acceleration
128 and 128	2,908	3.8 (6.2)
256 and 256	4,041	3.1 (4.5)
384 and 384	5,090	2.6 (3.5)
512 and 512	6,201	2.2 (2.9)
640 and 640	7,284	2.0 (2.5)
768 and 768	8,348	1.8 (2.2)
896 and 896	9,477	1.6 (1.9)
1024 and 1024	10,544	1.5 (1.7)

The column "L_{min} and L_{max}" includes two values because the analysed circuit permits the maximum subset with L_{max} elements and the minimum subset with L_{min} elements to be extracted at the same time.

The second column indicates the consumed time for all necessary operations in the PS and PL of the APSoC.

The column "Acceleration" also shows (see the values in parentheses) acceleration without taking into account communication overheads (i.e. without the mentioned above value 1,800 μs). Analysis of this case permits to estimate potential acceleration when data are transferred only once and then used for different computations in the APSoC.

We also evaluated potentialities for further accelerations and came to the following conclusions.

Software in the host PC is running in a high-performance multicore processor operating at significantly higher clock frequency than APSoC. To achieve additional acceleration in generally slower reconfigurable logic: a) high-level parallelism has to be used enabling hundreds of operations needed in software programs to be executed at the same time; b) the depth of combinational circuits implemented in the PL cannot be large because deep circuits involve extensive combinational path delays. The chosen sorted network [15] is not deep and operates at significantly higher clock frequency than alternative known circuits (see experiments in [15]). Higher-level parallelism can be achieved by joining data exchange and data processing operations. Let us look at Fig. 3, Fig. 4. Data are received from AXI HP ports sequentially and the maximum size of data items from one port is 64 bit. Such data need to be unrolled with the aid of the distributor circuit shown in Fig. 4. Thus, while a current block of data is being processed in the circuit in Fig. 3, the next block can be received and unrolled.

From Table I we can see that the larger are the values L_{min}/L_{max}, the smaller is the achieved acceleration. However, for the majority of practical applications very large values of L_{min}/L_{max} are not needed. Additional experiments demonstrated that the larger are the blocks of data handled in the PL the higher is the acceleration. We found that the size of such blocks for the ZC706 system could be increased from the considered 256 32-bit words to 2,048 32-bit words. Thus, the results may be additionally improved.

There are 4 AXI HP and one AXI accelerator coherency ports in Zynq devices [9]. Using many ports in parallel can be seen as another opportunity to increase throughput of hardware accelerators in the PL.

V. CONCLUSIONS

The paper suggests novel solutions for extracting subsets with the desired properties from large data sets and evaluates capabilities of a 3-level computing system that combines general-purpose software, application-specific software and reconfigurable hardware. We elaborated architecture of such a system and evaluated different implementations of the proposed solutions making a number of experiments with the Xilinx ZC706 prototyping system which interacts with a general-purpose computer through PCI express bus. We found that the considered PC-PS-PL computing system is faster by a factor ranging from 1.5 to 3.8 comparing to general-purpose software running in i7-based multicore PC.

REFERENCES

[1] Z. K. Baker, V. K. Prasanna, "An architecture for efficient hardware data mining using reconfigurable computing systems", *Proc. 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, USA, 2006, pp. 67–75. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2006.22

[2] S. Sun, "Analysis and acceleration of data mining algorithms on high performance reconfigurable computing platforms", *Ph.D. Dissertation*, Iowa State University, 2011. [Online]. Available: http://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=1421&context=etd

[3] X. Wu, V. Kumar, J. R. Quinlan, *et al*., "Top 10 algorithms in data mining", *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10115-007-0114-2

[4] M. F. M. Firdhous, "Automating legal research through data mining", *Int. Journal of Advanced Computer Science and Applications*, vol. 1, no. 6, pp. 9–16, 2010.

[5] D. Zmaranda, H. Silaghi, G. Gabor, C. Vancea, "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, 2013. [Online]. Available: http://dx.doi.org/10.15837/ijccc.2013.1.181

[6] L. Field, T. Barnie, J. Blundy, R. A. Brooker, D. Keir, E. Lewi, K. Saunders, "Integrated field, satellite and petrological observations of the November 2010 eruption of Erta Ale", *Bulletin of Volcanology*, vol. 74, no. 10, pp. 2251–2271, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00445-012-0660-7

[7] W. Zhang, K. Thurow, R. Stoll, "A knowledge-based telemonitoring platform for application in remote healthcare", *Int. Journal of Computers, Communications and Control*, vol. 9, no. 5, pp. 644–654, 2014. [Online]. Available: http://dx.doi.org/10.15837/ijccc.2014.5.661

[8] D. Verber, "Hardware implementation of an earliest deadline first task scheduling algorithm", *Informacije MIDEM*, vol. 41, no. 4, pp. 257–263, 2011.

[9] Xilinx, Inc., "Zynq-7000 All Programmable SoC Technical Reference Manual", 2014. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[10] Xilinx, Inc., *XCell Journal*, vol. 83, second quarter, 2013.

[11] Xilinx, Inc., "ZC706 evaluation board for the Zynq-7000 XC7Z045 all programmable SoC. user guide", 2013. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf

[12] V. Sklyarov, I. Skliarova, J. Silva, A. Rjabov, A. Sudnitson, C. Cardoso, *Hardware/Software Co-Design for Programmable Systems-on-Chip*, Tallinn: TUT Press, 306 p., 2014.

[13] Xilinx, Inc., "AXI Bridge for PCI Express v2.5", 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_5/pg055-axi-bridge-pcie.pdf

[14] Xilinx, Inc., "OS and Libraries Document Collection", 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/oslib_rm.pdf

[15] V. Sklyarov, I. Skliarova, "High-performance implementation of regular and easily scalable sorting networks on an FPGA", *Microprocessors and Microsystems*, vol. 38, no. 5, pp. 470–484, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2014.03.003

[16] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, "Fast matrix covering in all programmable systems-on-chip", *Elektronika ir Elektrotechnika*, vol. 20, no. 5, pp. 150–153, 2014. [Online]. Available: http://dx.doi.org/10.5755/j01.eee.20.5.7116

PUBLICATION IV

Sklyarov, V.; Skliarova, I.; **Rjabov, A.**; Sudnitson, A. (2015). Zynq-based System for Extracting Sorted Subsets from Large Data Sets. Journal of Microelectronics, Electronic Components and Materials, 45, 142−152.

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: $T_s > T_{sch} \le T_{sh} + T_h + T_c$, 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, $T_{sch} \le T_{sh} + T_h + T_c$. 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.

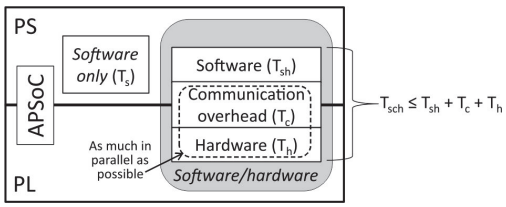

Figure 1: Software only and software/hardware systems

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.

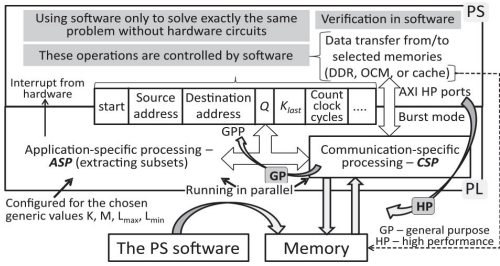

Figure 2: The proposed software/hardware architecture

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.

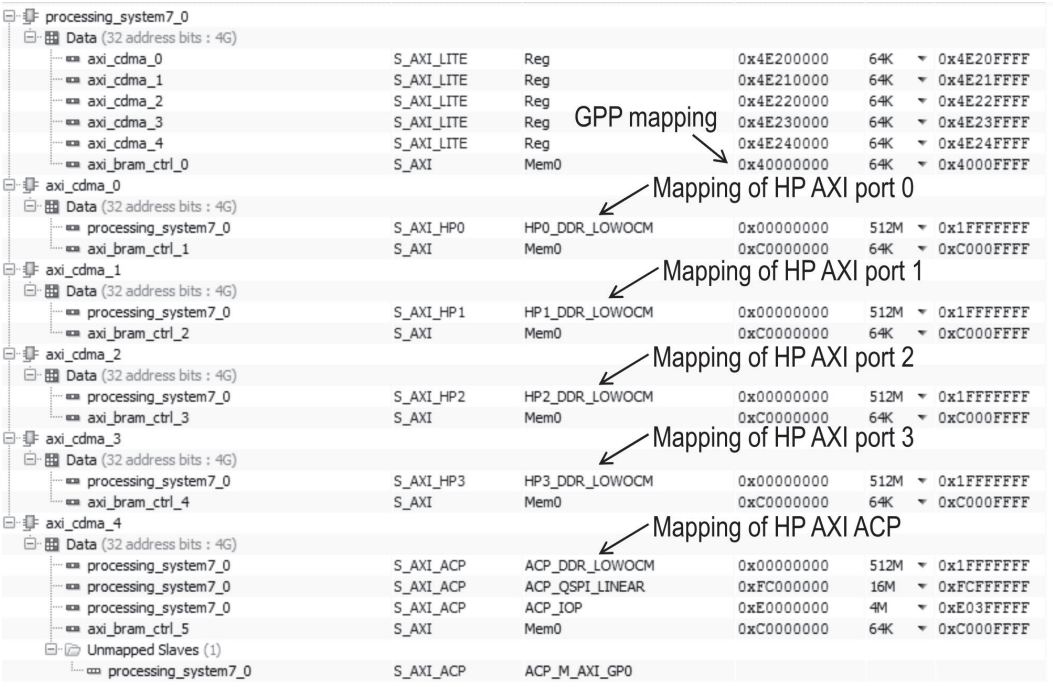

Figure 3: Address mapping from Vivado 2014.2 block design editor

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:

```
#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
```

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].

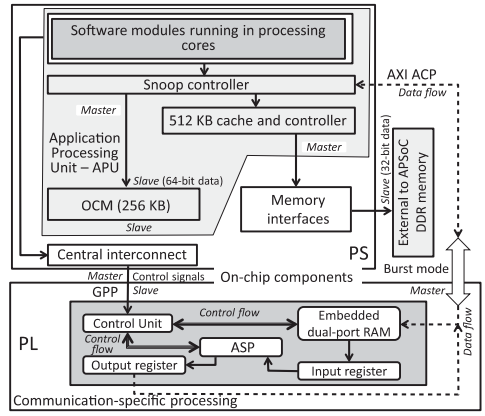

Figure 4: Hardware/software interactions

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.

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).

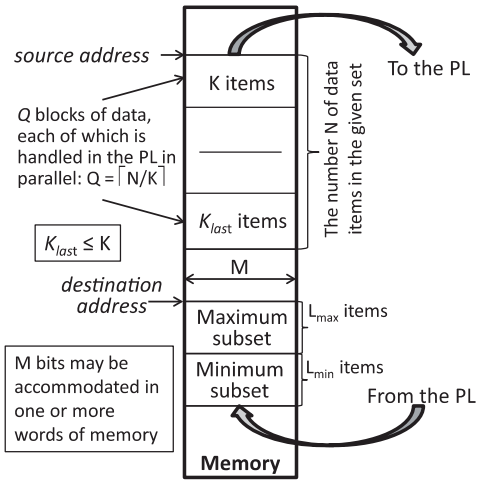

Figure 5: Accommodation of the initial data set and the extracted subsets in memory

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.

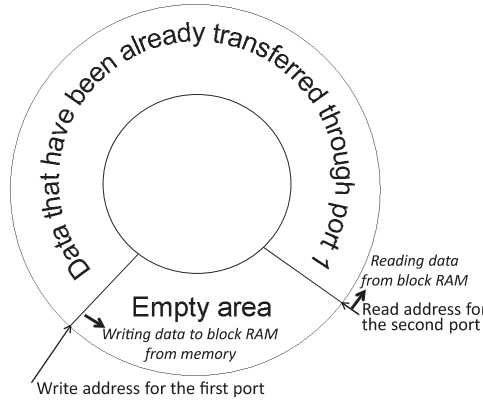

Figure 6: Block RAM organized as a circular buffer

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 ($L_{max} \le N$ and $L_{min} \le N$). 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.

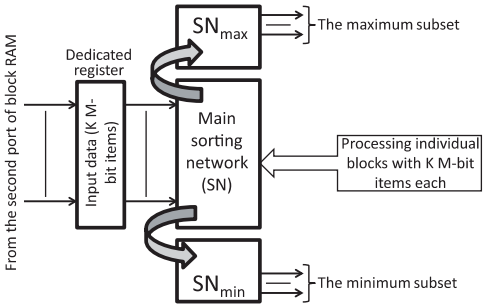

Figure 7: Basic hardware architecture for ASP

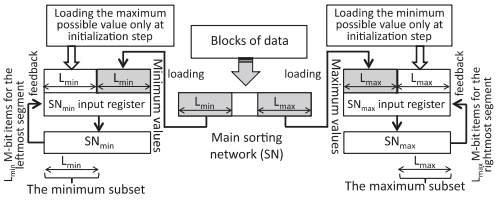

Figure 8: The first method of extracting the maximum and minimum sorted subsets

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.

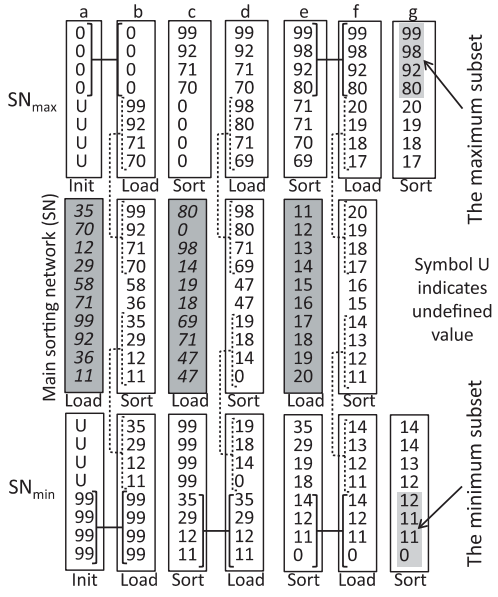

Figure 9: Example of extracting sorted subsets using the first method

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.

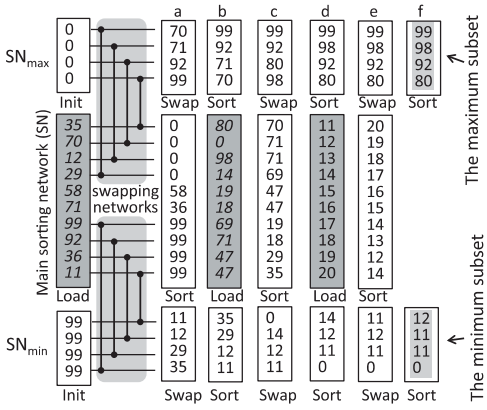

Figure 10: Example of extracting sorted subsets using the second method

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-

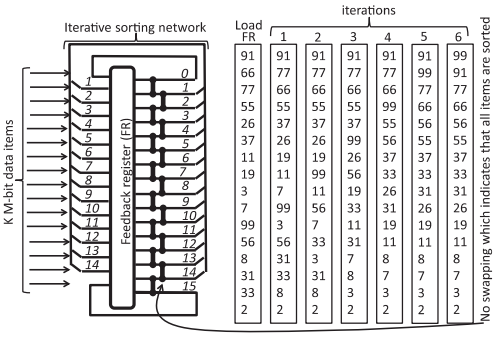

Figure 11: An example of iterative sorting network from [27] for K=16 data items

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 I_{max} and I_{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 I_{max} and I_{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 I_{max} and I_{min}, i.e. $L_{max} > I_{max}$ and $L_{min} > I_{min}$. In such case the maximum and minimum subsets can be computed iteratively as follows:

1. At the first iteration, the maximum subset containing I_{max} items and the minimum subset containing I_{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 I_{max}/I_{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.

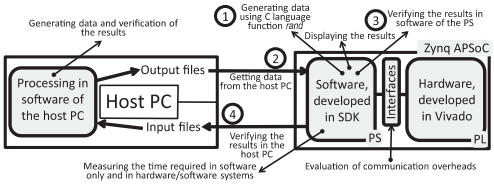

Figure 12: Experimental setup

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.

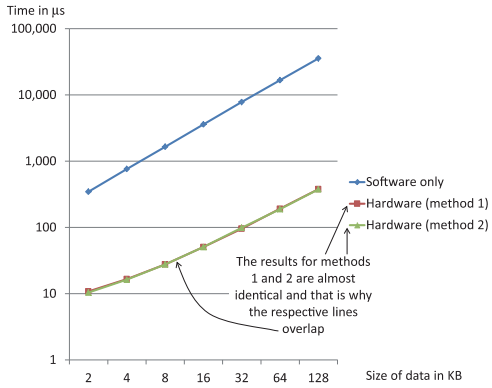

Figure 13: Computing time in software only and software/hardware systems

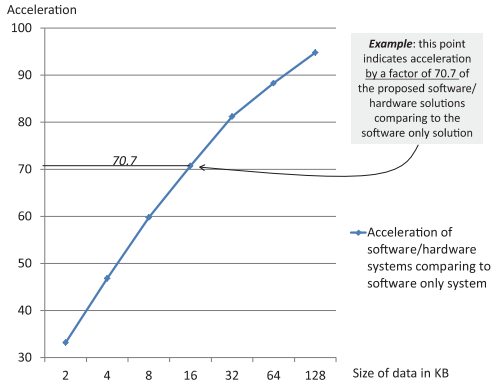

Figure 14: Acceleration of software/hardware systems comparing to software only system

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

PUBLICATION V

Artjom Rjabov (2016). Hardware-based systems for partial sorting of streaming data. 15th Biennial Baltic Electronics Conference (BEC2016), Tallinn, Estonia, October 3-5, 2016. IEEE, 59−62.

Hardware-based Systems for Partial Sorting of Streaming Data

Artjom Rjabov
Department of Computer Engineering,
Tallinn University of Technology,
Tallinn, Estonia
artjom.rjabov@ttu.ee

***Abstract*—Fast data sorting is an essential component of many high-performance computing systems. High-throughput and highly parallel sorting algorithms are very appropriate for devices which provide massive parallelism like FPGAs and APSoCs. One of the major drawbacks of these platforms is amount of available resources which is a serious obstacle for design of hardware sorters. However, for many practical applications complete sorting is not important and only partial sorting for extraction of maximum and minimum subsets of the data is required. In this paper we investigate the maximum and minimum extraction problem, present a hardware-based extracting circuit based on pipelined periodic sorting networks and compare it with known alternatives.**

Keywords—High-performance computing systems, Information processing; Sorting networks; Parallel sorting; Partial sorting; reconfigurable computing.

I. INTRODUCTION

Parallel algorithms for data sorting have been studied in computer science for decades. There are many different parallel sorting algorithms. The most notable of them are Parallel QuickSort, Parallel Radix Sort, Sample Sort, Histogram Sort [1] and a family of algorithmic methods known as sorting networks [2]. The latter presents a great interest for hardware acceleration. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagate through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line. Sorting is a very computationally expensive and time consuming operation which requires a lot of hardware resources. There are different approaches to overcome these limitations. Utilizing iterative networks with reusable comparators permits to process significantly larger data sets. Another two possibilities to overcome these problems are utilization of a relatively small parallel sorter along with a merging circuit or implementation of partial sorting. In this paper we investigate the latter possibility.

Many applications do not require all inputs to be sorted. Some of them necessitate only maximal and minimal values to be selected. 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). Maximum and minimum subsets extraction is required in searching, statistical data manipulation and data mining (e.g. [3] [4]). To describe one of the problems from data mining informally let us consider an example [3] 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 [3]. 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 [3]. 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.

The paper suggests a method and high-performance hardware implementation of a partial sorter for maximum (or minimum) subset extraction and maximum (or minimum) unsorted subset selection based on parallel sorting network. The system is designed for working over streaming data. It utilizes AXI interfaces and is suggested as a PCI express peripheral.

The remainder of the paper contains 6 sections. Section II analyzes the related work. Section III describes highly parallel networks for sorting and explores hardware co-design. Section IV describes experimental setup and hardware accelerator architecture. Section V presents the results of experiments and comparisons with known alternatives. The conclusion is given in Section IV.

II. RELATED WORK

The problem of finding subsets of minimum and maximum values is well known, but very small number of solutions exist. The majority of works in this area are focused on finding 1 or 2 maximal or minimal values in data sets ([5] [6]), but only few works are focused on subsets.

Frarmahini-Farahani et al. investigated the problem of partial sorting and max-set-selection in [7]. They proposed a modular design of a partial sorting system based on Batcher's

978-1-5090-1392-0/16/$31.00 ©2016 IEEE

odd-even and bitonic sorting networks. Their system is built on sorting blocks constructed from Batcher's odd-even merge (OEM) and bitonic sorting networks (BM), where bitonic sorters are reduced in order to get sorted maximal (or minimal) subset. They also proposed an approach to select unsorted maximal subset by replacing bitonic sorters with maximum selection units. In theory this technique is extendable to 2^n-to-2^m size (where m<n). Also they proposed an architecture for iterative max selection units that can potentially work with data streams. Another solution of this problem was developed by Biroli and Wang in [8]. Their approach is not based on sorting networks, but still uses parallel comparators. They applied fast circuit topologies for single max/min value search by Goren et al. [5] to find a subset of the largest or smallest values. In contradiction to Frarmahini-Farahani they didn't use Batcher's networks. Both works focused on finding relatively small subsets. The work by Frarmahini-Farahani is more suitable for work with large subsets, but its expansion will lead to large resource consumption.

In our previous works we also explored minimum and maximum extractors. In [9] and [10] we proposed different hardware/software methods for simultanious minimal and maximal subset and implemented them on Zynq-7000 APSoC devices. In [11] we proposed a multi-level architecture for minimal/maximal subset extraction which utilizes a general purpose processor of a host PC and the programmable logic and processing system of Zynq device.

III. Computing Sorted Subsets

As it was suggested in [9], some practical applications don't require maximal and minimal subsets simultaniously. For this purpose a reduced partial sorter that contains one main and one additional sorting network was discussed. In this paper we furher investigate this suggested approach because of its applicability and possible comparison with known alternatives. Additionally it is a pure hardware implementation which does not require embedded processing system like in our previous projects and it is designed as a hardware accelerator with PCI-express interface for ease on practical usage over streaming data.

Let set S containing N M-bit data items be given. The maximum subset contains L_{max} largest items in S. We mainly consider such tasks for which $L_{max} \ll N$ which are more common for practical applications (also applicable to a minimum subset). Since N may have very large value (millions of items) it cannot be processed completely in hardware due to the limited resources. It is shown in [12, 13] that even for relatively complex Field-Programmable Gate Arrays (FPGAs) the size N is limited. For example, for even-odd merge and bitonic merge networks [2] N 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).

As a basis for our sorting circuit we use a periodic pipelined Odd-Even Transition Sorting Network (also known as Odd-Even Transposition Sorting Network or OETS). A periodic network is a type of network which consists of identical sequences of comparators. Traditional implementation of OETS is less efficient than Batcher's networks, but it is more reliable and its implementation is simpler. Salloum and Wang proved that OETS has good fault-tolerant properties [14].

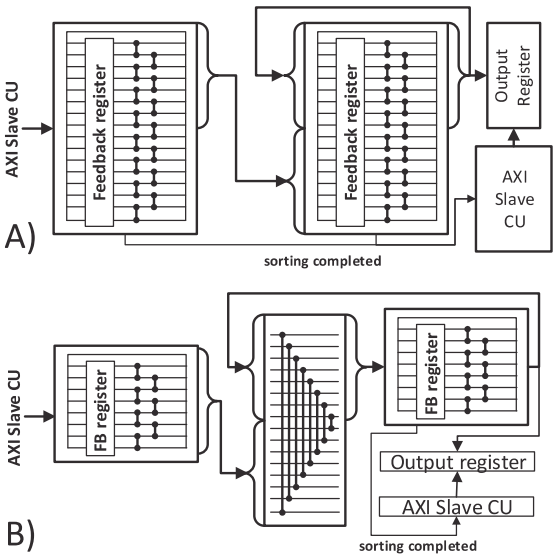

Fig. 1. Proposed methods for partial sorting circuits.

Like in our previous works, we use pipelined approach with reusable comparators presented in [13]. 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 [13] according to which if there are no swaps of data on the right-most line of comparators then sorting is completed. Note that the network [13] possesses significantly smaller combinational delays than networks from [2]. Besides, in the proposed architecture 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.

Two methods are depicted in Fig. 1. The first method utilizes two sorting networks of the same size. The first sorting network receives blocks of data and sorts them. After the sorting is completed, the maximal (or minimal) half loads into the second sorting network along with maximal (or minimal) half of outputs of the second network. For maximal set selection, in the initial step the second network is loaded with zeros. For minimal set selection, it is loaded with maximal possible value. After all the data is transmitted, the system waits for the completion of sorting in both sorting networks. The maximal (or minimal) half of the outputs of the second network is loaded in the output register and waits for read request.

The second method is also based on sorting networks, but these networks are two times smaller than in the first method,

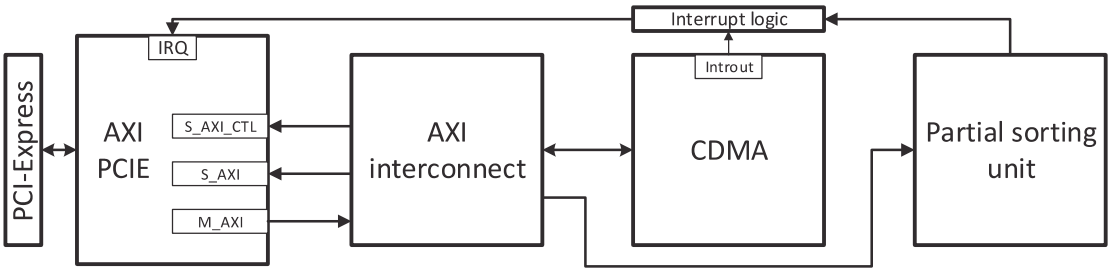

Fig. 2. Basic hardware architecture.

because we don't need here to sort results of the previous iteration with results of the current iteration. Both networks are connected here with a swapping network. All outputs of the first sorting network are connected to the swapping network along with all outputs of the second sorting network. On the outputs of the second network we receive unsorted maximal and minimum subsets of the input data, where all items of the upper half of the network are larger than all items of the lower half. In some practical applications receiving sorted maximal and minimal subsets is not required and only unsorted ones are needed. In that case we can turn the second sorting network off during the last iteration of the algorithm. It will reduce execution time which will be noticeable for relatively small amounts of data.

IV. Hardware Architecture and Experimental Setup

The system was designed as a hardware accelerator for a host PC which communicates through PCI-express interface in Direct Memory Access (DMA) mode. Fig. 2 depicts this architecture.

Software in the host PC runs the 32-bit Linux operating system (kernel 3.16) and executes programs (written in C language) that take results from PCI-express (from the FPGA) for further processing. We assume that the data collected in the FPGA are preprocessed in the programmable logic by applying various highly parallel networks (see Section III), and the results are transferred to the host PC through the PCI-express bus. To support data exchange through PCI-express, a dedicated driver was developed. The programmable logic uses the Intellectual Property (IP) core of the central direct memory access (Xilinx CDMA) module to copy data through AXI PCI express (Xilinx AXI-PCIE). Data transfer in the host PC is organized through direct memory access (DMA). To work with different devices, a driver (kernel module) was developed. The driver creates in the directory /dev a character device file that can be accessed through read and write functions, for example write(file, data array, data size). The PC BIOS assigns a number (an address) to the selected base address register (BAR) and a corresponding interrupt number that will be later used to indicate the completion of a data transfer. As soon as the driver is loaded, a special operation (probe) is activated and the availability of the device with the given identification number (ID) is verified (the ID is chosen during the customization of the AXI-PCIE). Then a sequence of additional steps is performed (see [15, pp. 302-326] for necessary details). A number of file operations are executed in addition to the probe function. In our particular case, access to the file is done through read/write operations.

V. Experimental Results and Comparison

All hardware solutions were implemented, evaluated and tested in Xilinx Virtex-7 XC7VX485T FPGA. The implementation in this paper was designed with an aim to compare it with known alternatives. We compared it with software sorting and a hardware solution from [7] (OEM/BM). Software solution is the most obvious and the most widely used quicksort implementation from C++ language (sort function). With this approach a whole data set is being sorted with subsequent extraction of the maximal (or minimal) subset. For comparison in hardware area, the system from [7] was implemented. After some experiments we found the most optimal configuration for implementation for Virtex-7 device which extracts 128-item data sets. Any implementation for extracting 256-item data sets utilizes more than 100% resources of the device. We implemented suggested in the paper concept of iterative max-set-selection units. The basis of this system is constructed from the two following blocks: 256-to-128 odd-even merge max-selection units and reduced bitonic 256-to-128 unit which starts with core max-selection unit. Inputs for core max selection units are outputs of OEM 256-to-128 and outputs of BM sorter (which contains results from the previous iteration).

For our methods we implemented two different systems. One for finding 128-item data subset in order to compare with OEM/BM method, and another for finding 1024-item data sets which is the maximal possible circuit that fits in Virtex-7 device. Post-implementation resource usage is shown in Table 1.

TABLE I. RESOURCE UTILIZATION

Method	Resources	
	FF	*LUT*
Method A 128	9%	22%
Method B 128	8%	19%
Method A 1024 (max)	38%	94%
Method B 1024 (max)	22%	70%
OEM/BM 128 (max)	52%	78%

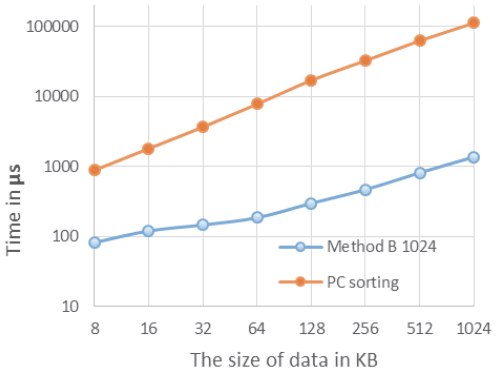

Fig. 3. Experimental results.

Lookup table (LUT) usage for the method A is 3,5 times smaller and for the method B is 4 times smaller than OEM/BM based solution. The method A requires 5,7 times fewer amount of flip-flop (FF) than OEM/BM and the method B requires 6,5 times fewer FFs. Also it is necessary to mention that all modules required for PCIe DMA system utilize about 15% of LUTs. By subtracting these resources we see that pure min/max system for the method A requires 9 times fewer LUTs and the method B requires 15,7 times fewer LUTs.

Available resources of Virtex-7 device allow us to expand our circuits for extracting larger maximum or minimum subsets. Both proposed architectures were expanded to extract subsets of 1024 items which is 10 times more than with OEM/BM approach. Although for simultaneous extracting of maximum and minimum subsets both proposed methods are identical in terms of resource usage and performance, the method B is better for extraction of maximum or minimum subset alone.

Fig. 3. shows experimental results. With Virtex-7 and the proposed PCI express transfer system all hardware implementations showed approximately identical results. With architectures that allow faster data transfer OEM/BM approach may show better results, because for the proposed methods A and B worst case performance is K/2 clock cycles for K inputs and OEM/BM performance is dependent on the number of pipeline stages. But because of significant economy of resources with the proposed methods (especially the method B) it is possible to speed up sorting by placing two or more instances of the sorting circuit that will sort parts of the whole data simultaneously.

VI. Conclusion

The paper suggests hardware-based methods of partial sorting for computing the maximum and minimum subsets of large sets of streaming data. The proposed solutions are highly parallel permitting capabilities of programmable logic to be used very efficiently. All the proposed methods were implemented in commercial microchips, tested, evaluated, and compared with alternatives. The results of experiments have shown significant advantages over other software and hardware solutions.

Acknowledgment

This research was supported by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, the Study IT in Estonia Programme.

References

[1] E.V. Kalé, E. Solomonik, "Sorting," in Encyclopedia of Parallel Computing, Springer Science+Business Media, 2011, pp. 1855-1862.

[2] S.W. Aj-Haj Baddar, K.E. Batcher, Designing Sorting Networks. A New Paradigm., Springer, 2011.

[3] Z.K. Baker, V.K. Prasanna, "An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems," in Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, USA, 2006.

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

[5] S. Goren, G. Dundar, B. Yuce, H.F. Ugurdag, "A fast circuit topology for finding the maximum of N k-bit numbers," in Symp. on Computer Arithmetic, 2013.

[6] C. Wey, M. Shieh, S. Lin, "Algorithms of finding the first two minimum values and their hardware implementation," IEEE Trans. Circuits and Systems I, vol. 55, no. 11, p. 3430–3437, 2008.

[7] A. Farmahini-Farahani, H.J. Duwe, M.J. Schulte, K. Compton, "Modular design of high-throughput, low-latency sorting units.," IEEE Transactions on Computers, vol. 62, no. 7, pp. 1389-1402, 2013.

[8] A.D.G Biroli, J.C. Wang, "A fast architecture for finding maximum (or minimum) values in a set. In Acoustics,," in 2014 IEEE International Conference on Speech and Signal Processing (ICASSP).

[9] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, (2015). Zynq-based System for Extracting Sorted Subsets from Large Data Sets. Informacije MIDEM, 45(2), 142-152.

[10] V. Sklyarov, A. Rjabov, I. Skliarova, A. Sudnitson, (2016). High-performance Information Processing in Distributed Computing Systems. International Journal of Innovative Computing, Information and Control, 12 (1), 139─160.

[11] A. Rjabov, V. Sklyarov, I. Skliarova, A. Sudnitson, (2015). Processing Sorted Subsets in a Multi-level Reconfigurable Computing System. Elektronika ir Elektrotechnika, 21(2), 30-33.

[12] R. Mueller, J. Teubner, G. Alonso, (2012); Sorting networks on FPGAs, The VLDB Journal—The International Journal on Very Large Data Bases, 21(1), 1-23.

[13] V. Sklyarov, I. Skliarova,. (2014); High-performance implementation of regular and easily scalable sorting networks on an FPGA, Microprocessors and Microsystems, 38(5): 470-484.

[14] S.N. Salloum, D.H. Wang,, "Fault tolerance analysis of odd-even transposition sorting networks with single pass and multiple passes," in IEEE Pacific Rim Conference on Communications, Computers and signal Processing, 2003.

[15] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, http://lwn.net/Kernel/

PUBLICATION VI

Sklyarov, V.; **Rjabov, A.**; Skliarova, I.; Sudnitson, A. (2016). High-performance Information Processing in Distributed Computing Systems. International Journal of Innovative Computing, Information and Control, 12 (1), 139−160.

International Journal of Innovative Computing, Information and Control
Volume **12**, Number **1**, February **2016**

 ISSN 1349-4198
pp. 139–160

HIGH-PERFORMANCE INFORMATION PROCESSING IN DISTRIBUTED COMPUTING SYSTEMS

Valery Skliarov[1], Artjom Rjabov[2], Iouliia Skliarova[1] and Alexander Sudnitson[2]

[1]Department of Electronics, Telecommunications and Informatics
Institute of Electronics and Informatics Engineering of Aveiro
University of Aveiro
Aveiro 3810-193, Portugal
{ skl; iouliia }@ua.pt

[2]Department of Computer Engineering
Tallinn University of Technology
Tallinn 19086, Estonia
aleksander.sudnitson@ttu.ee

Received July 2015; revised December 2015

Abstract. *This paper explores distributed computing systems that may be used efficiently in information processing that is frequently needed in electronic, environmental, medical, and biological applications. Three major components of such systems are: 1) data acquisition and preprocessing; 2) transmitting the results of preprocessing to a higher level computing system that is a PC; and 3) post processing in higher level computing system (in the PC). Preprocessing can be done in highly parallel accelerators that are mapped to reconfigurable hardware. The core of an accelerator is a sorting/searching network that is implemented either in an FPGA or in a programmable system-on-chip (such as Zynq devices). Data is transmitted to a PC through a high-bandwidth PCI-express bus. The paper suggests novel solutions for sorting/searching networks that enable the number of data items that can be handled to be significantly increased compared to the best known alternatives, maintaining a very high processing speed that is either similar to, or higher than in the best known alternatives. Preprocessing can also include supplementary tasks, such as extracting the minimum/maximum sorted subsets, finding the most frequently occurring items, and filtering the data. A higher level computing system executes final operations, such as merging the blocks produced by the sorting networks, implementing higher level algorithms that use the results of preprocessing, statistical manipulation, analysis of existing and acquired sets, data mining. It is shown through numerous experiments that the proposed solutions are very effective and enable a more diverse range of problems to be solved with better performance.*
Keywords: High-performance computing systems, Information processing, Sorting, Searching, Merging, Reconfigurable hardware, PCI-express bus, Programmable systems-on-chip

1. **Introduction.** Sorting and searching procedures are needed in numerous computing systems [1]. They can be used efficiently for data extraction and ordering in information processing. Some common problems that they apply to are (see also Figure 1):

1. Extracting sorted maximum/minimum subsets from a given set;
2. Filtering data, i.e., extracting subsets with values that fall within given limits;
3. Dividing data items into subsets and finding the minimum/maximum/average values in each subset, or sorting each subset;

4. Finding the value that is repeated most often, or finding the set of n values that are repeated most often;
5. Removing all duplicated items from a given set;
6. Computing medians;
7. Solving the problems indicated in points 1-6 above for matrices (for rows/columns of the matrices).

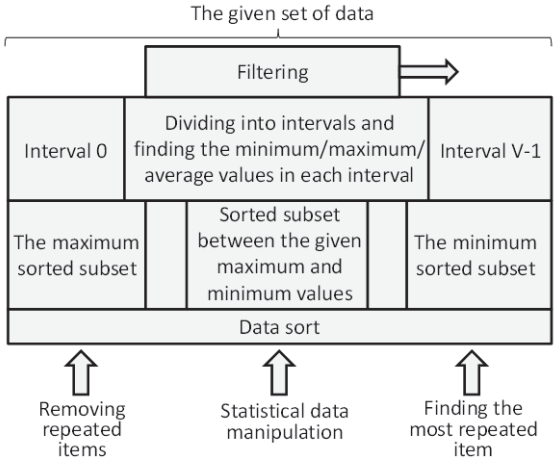

FIGURE 1. Common problems that are frequently solved in information processing systems

These problems are important because many electronic, environmental, medical, chemical, and biological applications need to process data streams produced by sensors and calculate certain parameters [2]. Let us consider some examples. Applying the technique [3] in real-time applications requires data acquisition from control systems such as in a manufacturing or process plant. Signals from sensors may need to be filtered and analyzed to prevent error conditions (see [3] for additional details). To provide a more precise and reliable conclusion, combinations of different values need to be extracted, ordered, and analyzed. Similar tasks arise in monitoring thermal radiation from volcanic eruptions [4], filtering and integrating information from a variety of sources in medical applications [5], in data mining [6], and so on. Since many control systems are real-time, performance is important and hardware accelerators can provide significant assistance for software.

The problems listed above can be solved as shown in Figure 2. Measured data items are handled in such a way that they are optionally filtered before various types of data processing algorithms are applied. Performance can be increased by employing broad parallelism and we suggest providing support for such parallelism in networks for sorting and searching.

Let us introduce a set of high-level operations, each of which is dedicated to one of the problems listed in points 1-6 above. The paper suggests methods for high-performance implementation of such operations in a distributed computing system with the architecture depicted in Figure 3.

Two basic subsystems (a pre-processor implemented in reconfigurable devices and a host PC) communicate through a high-bandwidth PCI-express (Peripheral Components Interface) bus. Two types of reconfigurable devices are studied: advanced field-programmable gate arrays (FPGAs), such as those from the Xilinx Virtex-7 family, and all-programmable

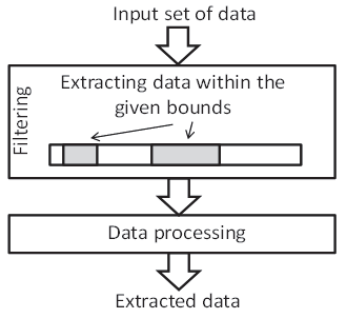

FIGURE 2. General architecture of data processing

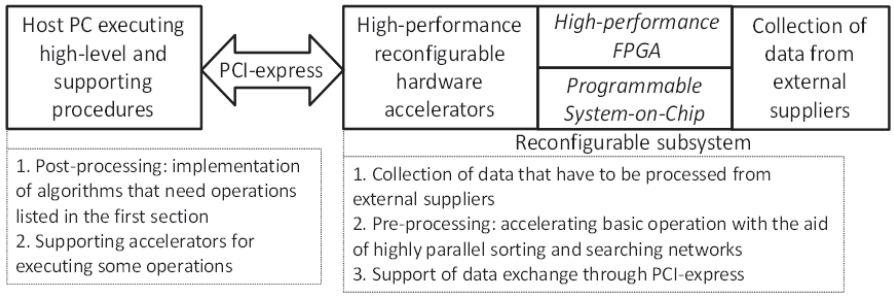

FIGURE 3. Architecture of a distributed computing system

systems-on-chip (APSoCs), such as those from the Xilinx Zynq-7000 family. In the first case, two-levels of processing are involved: accelerators implemented in reconfigurable hardware, and general-purpose software running in a PC. In the latter case, a third level of processing is added that is application-specific software running in a multi-core processing unit embedded to APSoC. The basic tasks of the host PC and the reconfigurable subsystem are listed in Figure 3.

The main contributions of the paper can be summarized as follows:

1) Selecting widely reusable operations for various types of information processing, and developing a methodology for using such operations in engineering applications;
2) Highly parallel networks for various sorting and searching algorithms and their thorough evaluation;
3) Multi-level implementation of the selected operations in general-purpose and application-specific software, and in reconfigurable hardware;
4) Experimental evaluation of the proposed technique in two advanced prototyping boards – the ZC706 [7] and the VC707 [8].

The remainder of the paper contains 6 sections. Section 2 analyzes related work. Section 3 identifies potential practical applications of the results and gives a number of real world examples from different areas. Section 4 describes highly parallel networks for sorting and searching. Section 5 explores software/hardware co-design. Section 6 presents the results of experiments and comparisons. The conclusion is given in Section 7.

2. **Related Work.** The majority of known methods and designs that are relevant to this paper are in three main areas: 1) high-performance distributed systems that include

a general-purpose computer (such as a PC) and a reconfigurable subsystem interacting through a high-bandwidth PCI-express bus; 2) sorting and searching using highly parallel networks; 3) software/hardware co-design targeted to combine reconfigurable hardware with general-purpose and application-specific software. The following subsections discuss the related work that has been done in each area separately.

2.1. **High-performance distributed systems.** High-performance distributed systems are used in many areas such as in medical equipment, aerospace, transportation vehicles, intelligent highways, defense, robotics, process control, factory automation, and building and environmental management [9]. A number of such systems for different application areas are described in [10-12]. Let us consider one of them from [12]. Typical adaptive cruise control and collision avoidance systems receive periodic inputs from sensors such as radar, lidar (light identification detection and ranging), and cameras, and then process the data to extract the necessary information. The system then executes a control algorithm to decide how much acceleration or deceleration is required, and sends commands to actuators to execute the appropriate actions. Since this is time-critical functionality, the end-to-end latency from sensing to actuation must be bounded.

The application domains define different requirements for systems. Independently of the domains, the majority of applications need data processing that may be organized in different ways. For example, a researcher may require support for finding data items that occur together in a certain situation, either a maximum or a minimum number of times. Such problems arise in determining the frequency of inquiries over the Internet, for customer transactions such as credit card purchases, which typically produce very large volumes of data in the course of a day [6]. Data processing is involved in software systems [13], priority buffering in scheduling algorithms [10], information retrieval [14], extracting data from sensors within predefined ranges [2], video processing [15], knowledge acquisition from controlled environments [3], and so on. Satisfying the real-time requirements for these applications can be achieved in on-chip devices that combine a multi-core processing system (PS) running multi-thread software with programmable logic (PL) that can be used to implement hardware accelerators. For example, devices from the Xilinx Zynq-7000 family of APSoC have already been successfully used in a number of engineering designs [11,12]. The Zynq APSoC combines the dual-core ARM®Cortex™-A9 central processing unit with the 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 enable complete solutions to be implemented on a single microchip running software that may be enhanced with easily customizable hardware. Various advantages of the APSoC platform are summarized in [16,17]. Interactions between the ARM-based 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) [15]. There are a number of prototyping systems available, some of which (e.g., [7]) allow additional data exchange with higher level computers such as PCs through PCI-express. On-chip interfaces [18] enable hardware accelerators and other circuits (supporting, for example, communications with sensors and actuators) in the PL to be linked with multi-core software running in the PS. The PCI-express bus permits multilevel systems to be developed that combine software running in a general-purpose computer (e.g., PC), software running in the PS, and hardware implemented in the PL.

Satisfying real-time requirements is critical in many of the systems referenced above and this can be a problem for the software-only systems that, perhaps, are the most frequently applied nowadays [11,13]. Hardware-only systems are also widely used in a number of

areas (e.g., [12]). It is concluded in [12] that the most sensible approach is for the data intensive portions of an application to be implemented in hardware, thus providing a high degree of determinism and lower execution time, while the high-level decision making is implemented in software, supporting easy customization. It is shown in [19] that on-chip interactions between software and hardware may be seen as a bottleneck (even if HP ports are used) especially for applications that require the exchange of high volumes of data.

The architectures and functionalities of various PCI-express systems are described in [20]. A PCI connection has one or more data transmission lanes, each of which consists of two pairs of wires: one for receiving and one for sending data. The maximum theoretical bandwidth of a single lane is up to 2.5 Giga transfers per second (GT/s) in each direction simultaneously [21], which is the same as Gb per second except that some bits are lost as a result of interface overhead and consequently the theoretical bandwidth is reduced by approximately 20% [21,22]. The bandwidth of χ lanes is the bandwidth of one lane multiplied by χ.

There are many systems that involve high-performance on-chip communications and interactions with PC through a PCI-express bus. The distinctive feature of this paper is the study and evaluation of such systems for a particular area of data processing targeted to problems that were described in Section 1. We will show in the next section that there exist a very large number of potential practical applications for such processing.

2.2. **Sorting and searching networks.** Highly parallel networks for sorting and searching enable numerous operations to be executed simultaneously, which is very appropriate for FPGAs and APSoCs. Two of the most frequently investigated parallel sorters are based on sorting [23] and linear [24] networks. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagate through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line. Three types of such networks have been studied: pure combinational (e.g., [23,25,26]), pipelined (e.g., [23,25,26]), and combined (partially combinational and partially sequential) (e.g., [27]). The linear networks, which are often referred to as linear sorters [24], take a sorted list and insert new incoming items in the proper positions. The method is the same as the insertion sort [1] that compares a new item with all the items in parallel, then inserts the new item at the appropriate position and shifts the existing elements in the entire multi-item vector. Additional capabilities of parallelization are demonstrated in the interleaved linear sorter system proposed in [24]. The main problem with this is that it is applicable only for small data sets (see, for example, the designs discussed in [24], which accommodate only tens of items).

The majority of sorting networks that are implemented in hardware use Batcher even-odd and bitonic mergers [28,29]. Other types are rarer (see for example the comb sort [30] in [31], the bubble and insertion sort in [23,25], and the even-odd transition sort in [32]). Research efforts are concentrated mainly on networks with a minimal depth or number of comparators and on co-design, rationally splitting the problem between software and hardware. The regularity of the circuits and interconnections are studied in [26,27] where networks with iteratively reusable components were proposed. We target our results towards FPGAs and APSoCs because they are regarded more and more as a universal platform incorporating many complex components that were used autonomously not so long ago. The majority of modern FPGAs contain embedded DSP slices and embedded multi-port memories, which are very appropriate for sorting. GPU (graphics processing unit) cores have also been placed inside an APSoC in recent devices [17], but even without

such cores, streaming SIMD (single instruction multiple data) applications can be supported with the existing programmable logic. Comparing FPGA-based implementations with alternative systems [33,34] clearly demonstrates the potential of reconfigurable hardware, which encourages further research in this area. FPGAs still operate at a lower clock frequency than non-configurable ASICs (application-specific integrated circuits) and ASSPs (application-specific standard products) and broad parallelism is evidently required to compete with potential alternatives. Thus, sorting and linear networks can be seen as very adequate models. Unfortunately, they have many limitations. Suppose N data items, each of size M bits, need to be sorted. The results of [23,25] show that sorting networks cannot be built for $N > 64$ ($M = 32$), even in the relatively advanced FPGA FX130T from the Xilinx Virtex-5 family because the hardware resources are not sufficient. When N is increased, the complexity of the networks (the number of comparators $C(N)$) grows rapidly (see Figure 1 in [26]). Besides, propagation delays through long combinational paths in FPGA networks are significant [26]. Such delays are caused not only by comparators, but also by multiplexers that have to be inserted even in partially regular circuits [27], and by interconnections.

It is shown in [26] that very regular even-odd transition networks with two sequentially reusable vertical lines of comparators are more practical because they operate at a higher clock frequency, provide sufficient throughput, and enable a significantly larger number of items to be processed in programmable logic.

2.3. **Multi-level software/hardware co-design.** Multi-level software/hardware co-design is a new kind of system design that combines on-chip hardware/software co-design and co-design at the level of PC interacting with the on-chip subsystem, such as that implemented in Zynq-7000 devices. To our knowledge just a few publications (such as [35,36]) briefly discuss such multi-level co-designs. On the other hand existing prototyping boards, such as [7] permit such designs to be evaluated. Besides, they are valuable for numerous practical applications that were listed in Section 1.

3. **Potential Practical Applications.** Let us discuss now applicability of the considered design technique. An ordinary sorting is required in many types of information processing. For a large number of data items, the known procedures that are used are time consuming and can be accelerated using the methods proposed in this paper. This is especially important for portable embedded applications. In the latter case, even sorting thousands of items can be done significantly faster in software/hardware (e.g., in APSoC) than in software only.

One common problem is clustering objects in accordance with their attributes. Different methods have been proposed for solving this problem and many of them involve sorting and searching as frequently used operations [37-40]. For example, in the CPES (Clustering with Prototype Entity Selection) method [41], a fitness function is proposed to decide if given objects can be clustered. Sorting the results produced by the fitness function enables solutions to be found faster. Actually, for many practical applications it is important to make sorting built-in, much as in operations such as computing the Hamming weights of binary vectors, e.g., POPCNT (population count) [42] and VCNT (Vector Count Set Bits) [43]. Similar proposals were made in [44].

Another example is extracting a required number of items through the Internet. For example, suppose we would like to find the N cheapest products from very large number of available options. An FPGA/APSoC based system requests and receives data from different suppliers and extracts the maximum/minimum subsets with the indicated number of items. Different search criteria can be applied and very large volumes of data can be

analyzed. Other practical applications (in which high throughput is very important) are described in [2]. One particular example can be taken from [2], which requires deciding how often the data collected falls within a set of critical values that are above or below a given threshold. Generally, the greater the intensity, the more critical is the subset and the higher the probability of an event which might happen. By discovering the maximum and minimum subsets you can determine when the activity is the highest or the lowest.

The algorithm [45] discovers rules associated with a set of classes and it has been tested on a real world application data set related to website phishing. In this algorithm the classifier sorts classes within each rule based on their frequency. Thus, sorting is also needed.

In [25] 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 [46] and in data analysis [47], which are tasks that occur often in control engineering.

The method proposed in [48] (based on the Monte Carlo method coupled with a sorting algorithm [49] and gradient search [50]) systematically evaluates possible behaviors of a closed-loop system by analyzing its time response. This permits various techniques to be applied for solving problems that are commonly encountered in networked control systems.

The software/hardware solutions proposed in this paper are faster. They are based on two (PC - FPGA) or three (PC - APSoC: PS - PL) level systems. The effectiveness of the hardware/software solutions is underlined in [51], addressing the importance of portable computing hardware environments to handle massive data.

4. **Highly Parallel Networks for Sorting and Searching.** It is shown in Section 2.2 that sorting networks are widely used in data [25] and vector [52] processing and they enable comparison and swapping operations over multiple data items to be executed in parallel. A review of recent results in this area can be found in [26] where it is shown that many researchers and engineers consider such technique as very beneficial for data and vector processing in FPGAs and APSoCs. Although the methods [28,29] enable the fastest theoretical throughput, the actual performance is limited by interfacing circuits supplying initial data and transmitting the results and the communication overheads do not allow theoretical results to be achieved in practical designs [19].

The proposed circuits require a significantly smaller number of comparators/swappers (C/S) than networks from [28,29] and many C/S are active in parallel and reused in different iterations. The first circuit (see Figure 4) contains N M-bit registers $Rg_0, \ldots, Rg_{N-1}$. Unsorted input data are loaded to the circuit through N M-bit lines $d_0, d_1, \ldots, d_{N-1}$. For the fragment on the left-hand side of Figure 4, the number N of data items is even, but it may also be odd which is shown in the example in the same Figure 4. Each C/S is shown in Knuth notation (↕) [1] and it compares items in the upper and lower registers and transfers the item with the larger value to the upper register and the item with the smaller value to the lower register (see the upper right-hand corner of Figure 4). Such operations are applied simultaneously to all the registers linked to even C/S in one clock cycle (indicated by the letter α) and to all the registers linked to odd C/S in a subsequent clock cycle (indicated by the letter β). This implementation may be unrolled to an even-odd transition network [32], but vertical lines of C/S in Figure 4 are activated sequentially and the number of C/S is reduced compared to [32] by a factor of $N/2$. For example, if the number N is even then the circuit from [32] requires $N \times (N-1)/2$ C/S and the circuit in Figure 4 – only $N-1$ C/S. The circuit in [32] is combinational and the circuit in Figure 4 may require up to N iterations. The number N of iterations can

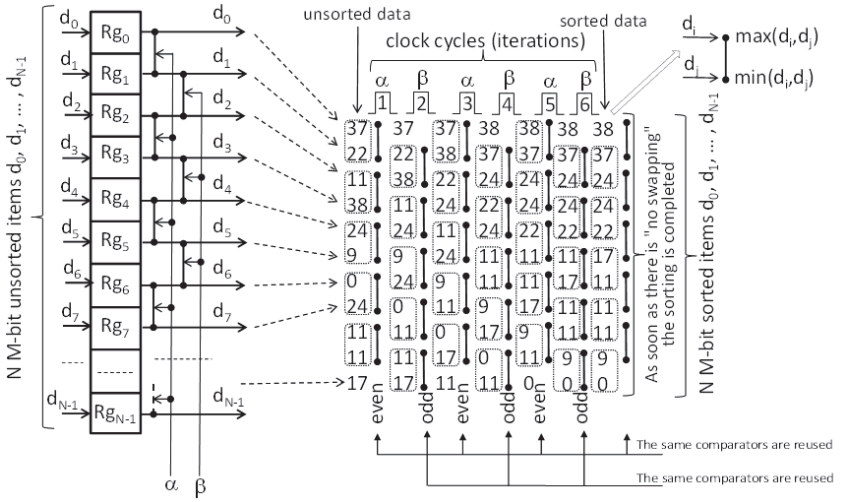

FIGURE 4. The first sorting circuit with an example

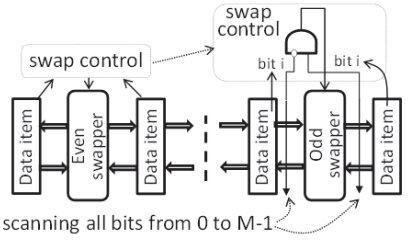

FIGURE 5. The second sorting circuit

be reduced very similarly to [26]. Indeed, if beginning from the second iteration, there is no data exchange in either even or odd C/S, then all data items are sorted. If there is no data swapping for even C/S in the first iteration, data swaps for odd C/S may still take place. Note that the network [32] possesses a long combinational delay from inputs to outputs. The circuit in Figure 4 can operate at a high clock frequency because it involves a delay of just one C/S per iteration (i.e., in each rising/falling edge of the clock).

Let us look at the example shown in Figure 4 ($N = 11$, $M = 6$). Initially, unsorted data $d_0, d_1, \ldots, d_{10}$ are copied to $Rg_0, \ldots, Rg_{10}$. Each iteration (6 iterations in total) is forced by an edge (either rising or falling) of a clock. The signal α activates the C/S between the registers $Rg_0, Rg_1, Rg_2, Rg_3, \ldots, Rg_8, Rg_9$. The signal β activates the C/S between the registers $Rg_1, Rg_2, Rg_3, Rg_4, \ldots, Rg_9, Rg_{10}$. There are 10 C/S in total. Rounded rectangles in Figure 4 indicate elements that are compared at iterations 1-6. Data are sorted in 6 clock cycles and $6 < N = 11$. Unrolled circuits from [32] would require 50 C/S with the total delay equal to the delay of N sequentially connected C/S.

The second circuit is shown in Figure 5 and it combines the first circuit with the radix sort. Now only single bits from the registers of Figure 4 are compared and if the upper bit is 0 and the lower bit is 1, then M-bit data in the relevant upper and lower registers are swapped. Thus, any C/S is built on only one gate (see Figure 5) much like in [52]. Data are sorted in such a way that at the first step bit 0 (the least significant bit) is chosen.

At the second step bit 1 is chosen and at the last step bit $M-1$ is chosen. Hence, bits in all registers are chosen sequentially, i.e., they are scanned. Clearly, the number of clock cycles is increased but the resources occupied are reduced because one-bit C/S are smaller than M-bit C/S.

The maximum number of clock cycles for sorting N M-bit items is increased up to $N \times M$ but in practical designs it is smaller because the method described above enables sorting to be completed as soon as there is no need for swapping data items.

Figure 6 demonstrates an example. The same data as in Figure 4 are chosen. Now we consider binary codes of the items. For the first sort (0) bit 0 is analyzed, which is the least significant bit. All the remaining sorts (1-5) and the relevant (analyzed) bits are shown in Figure 6. After the last step (sort 5), all data are sorted and for clarity binary codes are shown as decimal numbers.

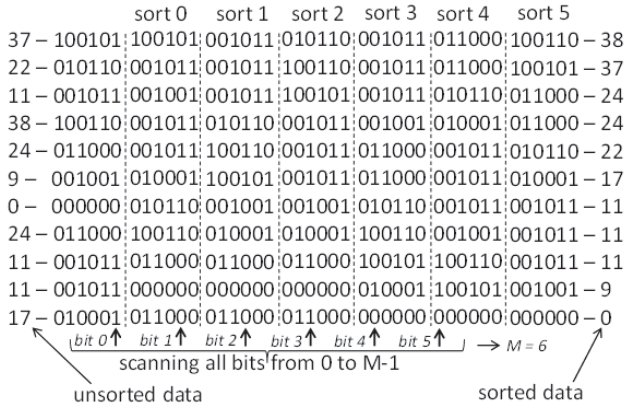

FIGURE 6. An example

The networks described above can be used efficiently for solving numerous supplementary tasks. One of these tasks is the extraction of the maximum and/or minimum subsets from the sorted sets [53-55]. Let set S containing L 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 ($L_{\max} \le L$ and $L_{\min} \le L$). We mainly consider such tasks for which $L_{\max} << L$ and $L_{\min} << L$, which are more common for practical applications. Since L may be very large (millions of items), the set cannot be completely processed in hardware because the resources required are not available. Figure 7 shows the top-level architecture from [55] in which the proposed iterative data sorter is used. Much like [55], the given set S is decomposed on $Q = \lceil L/K \rceil$ subsets, all of which contain exactly K M-bit items except the last, which may have less than K M-bit items. Subsets are computed incrementally in Q steps (we assume below that $K \le L$).

At the first step, the first K M-bit data items are sorted in the network (such as that shown in Figure 4) which handles $L_{\max} + K + L_{\min}$ data items in total but comparators linking the upper part (handling $L_{\max}$ M-bit data items) and the lower part (handling $L_{\min}$ M-bit data items) are deactivated (i.e., the links with the upper and lower parts are broken). So, sorting is done only in the middle part handling K M-bit items. As soon as sorting is completed, the maximum subset is copied to the upper part of the network and the minimum subset is copied to the lower part.

From the second step, all the comparators are active, i.e., the network (see Figure 4) handles $L_{\max} + K + L_{\min}$ items. Now for each new K M-bit items, the maximum and

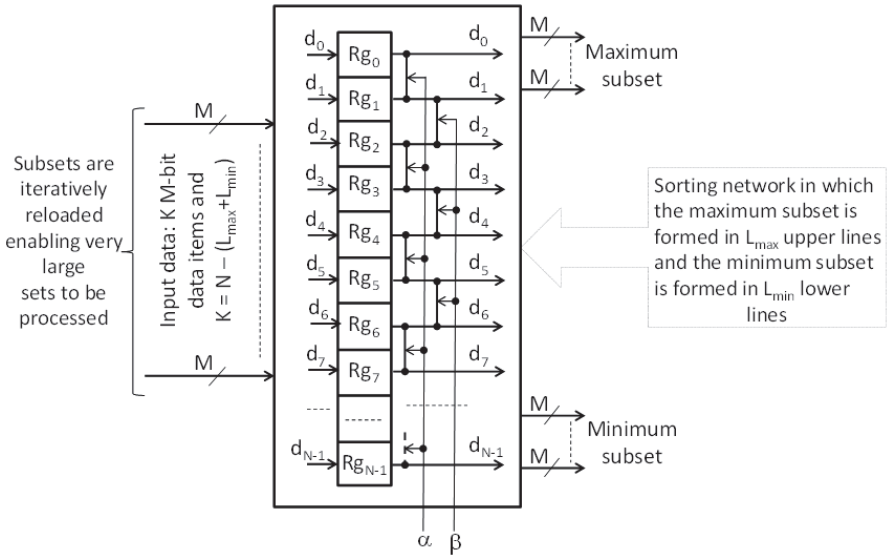

FIGURE 7. Computing the maximum and minimum sorted subsets

minimum sorted subsets are properly corrected, i.e., new items may be inserted. All other details relevant to top-level architecture in Figure 7 can be found in [55].

The next potential task that can be solved with the aid of the proposed methods is filtering [55]. Let B_u and B_l be predefined upper (B_u) and lower (B_l) bounds for the given set S. We would like to use the circuit in Figure 7 only for such data items D that fall within the bounds B_u and B_l, i.e., $B_l \leq D \leq B_u$ (or, possibly, $B_l < D < B_u$). Once again the basic component is the sorting network (see Figure 4) and it can be used in the method [55] that enables data items to be filtered at run-time (i.e., during data exchange).

Clearly, the operations described above can be implemented in software. For example, the C function qsort permits large data sets to be sorted. After that, extracting the maximum and minimum subsets is easily done. Filtering may be done by testing and eliminating items that do not fall within the predefined constraints. However, for many practical applications the performance of the operations described above is important. The results of thorough experiments and comparisons have shown that software/hardware solutions are significantly faster than software only solutions.

Many other problems can also be solved applying the proposed networks. For example, in [2] a highly parallel architecture was proposed that permits repeated items to be found efficiently. The basic component of the architecture [2] is a sorting network, and using the proposed technique for such a network permits the hardware resources to be reduced and the performance to be increased. Thus, the results of this paper are useful for a large number of practical applications.

5. **Software/Hardware Co-design.** Figure 8 shows the basic architecture for data transfer between a host PC and an APSoC through PCI-express.

Figure 9 presents a more detailed architecture of a three-level system for the example of distributed data sort. Software in the host PC runs the 32-bit Linux operating system (kernel 3.16) and executes programs (written in the C language) that take results from PCI-express (from the APSoC) for further processing. We assume that the data collected

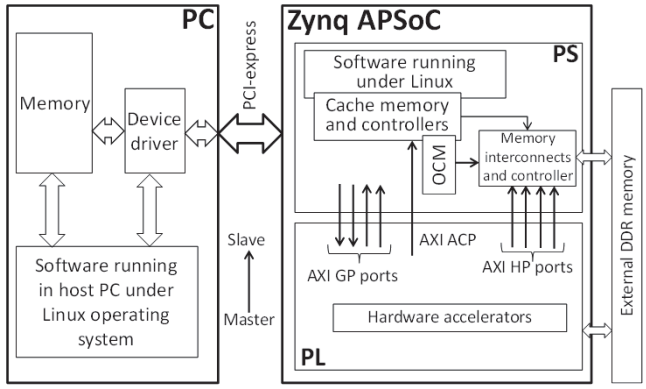

FIGURE 8. Basic architecture for data transfer between a host PC and an APSoC through PCI-express

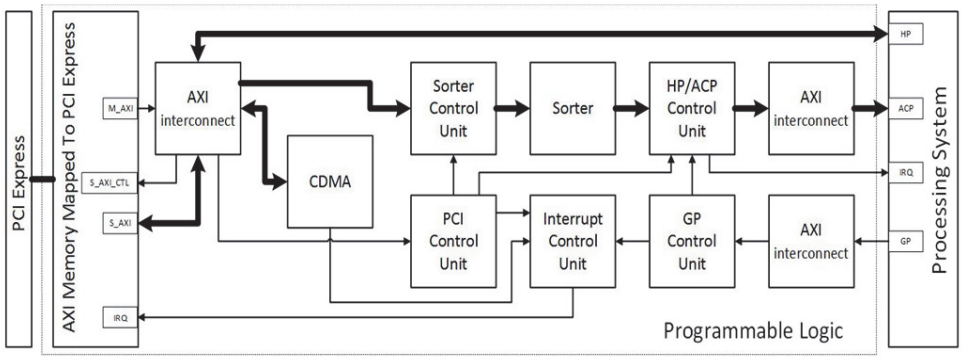

FIGURE 9. Architecture of a three-level APSoC-based system for data sorting

in the APSoC are preprocessed in the APSoC by applying various highly parallel networks (see Section 4), and the results are transferred to the host PC through the PCI-express bus. To support data exchange through PCI-express, a dedicated driver was developed. The APSoC uses the Intellectual Property (IP) core of the central direct memory access (CDMA) module [56] to copy data through AXI PCI express (AXI-PCIE) [57]. The project is similar to [58] and links CDMA and AXI-PCIE modules based on a simple data mover (i.e., the mode "scatter gather" [58] is not used). A master port (M-AXI) of the AXI-PCIE operates similarly to GP ports in [19] and supplies control instructions from the PC to customize data transfers. The instructions indicate the physical address of data for PC memory, the size of transferred data, etc. The CDMA module can be connected to either AXI HP or AXI ACP interfaces in APSoC and transmits data from either on-chip memory (OCM) or external DDR. After supplying the addresses, the number of data bytes (that need to be transferred) is indicated and the data transmission is started. As soon as the data transmission is completed, the CDMA module triggers an interrupt that has to be properly handled (the interrupt number is determined by the BIOS of the host PC). The following customization is done for 1) AXI-PCIE: legacy interrupts, 128 bits data width, and 2) CDMA: 256 bytes burst size, 128 bits data width. Note that the

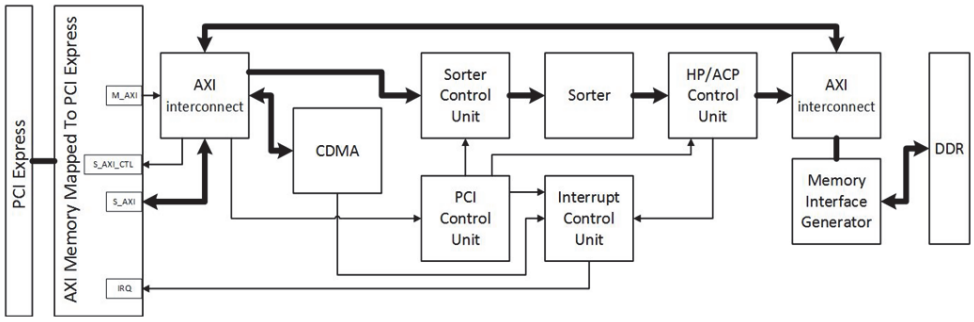

FIGURE 10. Architecture of a two-level FPGA-based system for data sorting

architecture in Figure 9 allows data transfers in both directions, i.e., data from the PC may also be received.

The architecture for the case when an FPGA is used (instead of an APSoC) is similar, and is shown in Figure 10 (now there are just two levels). The only difference is the transfer of data items from an external source to FPGA DDR memory where the data are preliminarily collected by the FPGA and stored. The DDR is controlled by a memory interface generator.

Data transfer in the host PC is organized through direct memory access (DMA). To work with different devices, a driver (kernel module) was developed. The driver creates in the directory /dev a character device file that can be accessed through read and write functions, for example write(file, data_array, data_size). Up to 5 base address registers (BAR) can be allocated but we used just one.

The PC BIOS assigns a number (an address) to the selected BAR and a corresponding interrupt number that will be later used to indicate the completion of a data transfer. As soon as the driver is loaded, a special operation (probe) is activated and the availability of the device with the given identification number (ID) is verified (the ID is chosen during the customization of the AXI-PCIE). Then a sequence of additional steps is performed (see [59, pp.302-326] for necessary details). A number of file operations are executed in addition to the probe function (see Figure 11). In our particular case, access to the file is done through read/write operations. Figure 11 demonstrates the interaction of a user application with the driver (kernel module) and some additional operations that may be executed.

As soon as a user program calls the read function, the read(file, data_array, data_size) function gets the address in the user memory space and the number of bytes that need to be transferred. Initially, the data are copied to a buffer and then the physical address of the buffer is obtained. Now the data are ready to be transferred from APSoC/FPGA. Then the data are copied and the driver is waiting for an interrupt indicating that the data transmission is complete. The necessary operations for generating the interrupt are given in [56]. Additional details can be found in [59, pp.258-287].

The proposed networks (see Section 4) can be used as follows. The sorter receives blocks composed of N M-bit data items that are collected from sensors initially and stored in memories (such as external DDR and OCM). Interactions with memory are done through AXI HP/ACP ports (see Figure 9) or through the memory interface block (see Figure 10). The sorter (such as that shown in Figure 4) executes iterative operations over multiple parallel data and is controlled by a dedicated finite state machine (FSM) called Sorter Control Unit (see Figures 9 and 10). The ports are also controlled by a dedicated FSM

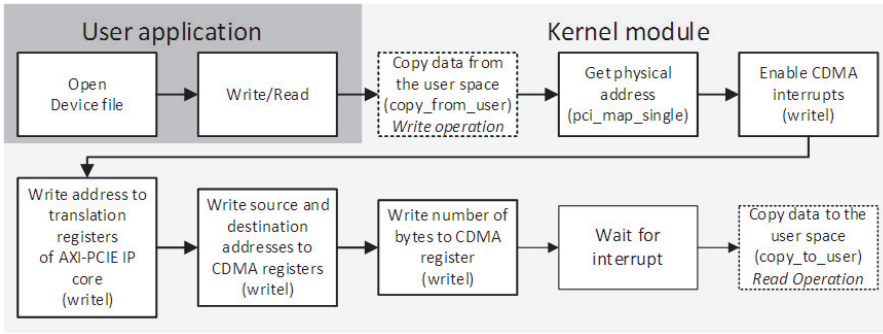

FIGURE 11. Operations with the device driver in the host PC and additional operations that may be executed

(see HP/ACP Control Unit in Figures 9 and 10). The results of sorting are copied back to memory and then transmitted to the host PC through the PCI-express bus. APSoC PS is responsible for data collection and organization that is done in accordance with the established requirements. For the case of FPGA, data collection and organization are done by specially developed dedicated circuits. Finally, either the PS or the dedicated circuits prepare data in memory so that these data can be processed in the PL/FPGA and the results of the processing (stored in memory) are ready to be transmitted to the host PC. The blocks CDMA with control units (PCI Control Unit and Interrupt Control Unit in Figures 9 and 10) are responsible for transmitting data.

6. **Experiments and Comparisons.** The system for data transfers between a host PC and an APSoC/FPGA has been designed, implemented, and tested. Experiments were done with two prototyping boards. The first is the Xilinx ZC706 evaluation board [7] containing the Zynq-7000 XC7Z045 APSoC device with PCI express endpoint connectivity "Gen1 4-lane (x4)". The PS is the dual-core ARM Cortex-A9 and the PL is a Kintex-7 FPGA from the Xilinx 7th series. The second board is VC707 [8] and it contains the Virtex-7 XC7VX485T FPGA from the Xilinx 7th series with PCI express endpoint connectivity "Gen2 8-lane (x8)". All designs were done for: 1) hardware in the PL of APSoC/FPGA synthesized from specifications in VHDL that describe circuits interacting with Xilinx IP cores (Xilinx Vivado Design Suite 2015.1/2015.2); 2) software in the PS of APSoC developed in C language (Xilinx Software Development Kit – SDK 2015.1); 3) user programs running under the Linux operating system in the host PC developed in C. Data were transferred from the ZC706/VC707 to the host PC through PCI-express. The host PC contains Intel core i7 3820 3.60GHz.

Figure 12 demonstrates organization of experiments with data sorters.

We assume that data are collected by the ZC706/VC707 board and stored in DDR memory (in the experiments, data are produced as described in point 1 below). Subsequently, different components (A, B, C, D) may be involved in data processing:

1) Data are randomly generated in the programmable logic (in the PL of APSoC or in the FPGA) and sorted using only networks in hardware (component A), indicated below as *Sorting blocks*;
2) Data are transferred from the ZC706/VC707 to the PC through PCI-express and sorted by software in the PC (component D), indicated below as *PC sort*;
3) Data are completely sorted in the APSoC (the set of data items is decomposed into blocks, blocks are sorted in the PL by the networks described above, the sorted blocks

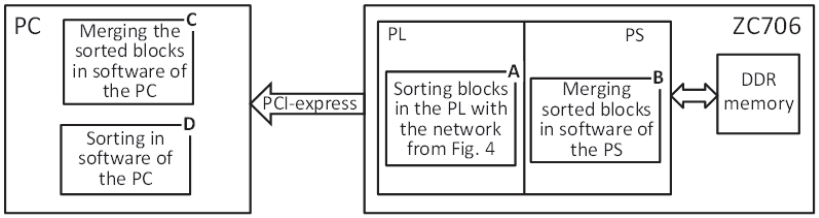

FIGURE 12. Organization of experiments with data sorters (the size of one block is 1024 32-bit data items)

are merged in the PS to produce the final result) and the sorted data are transferred to the PC through PCI-express (components A and B), indicated below as *Sorting + PS merge*;

4) Data are completely sorted in APSoC/FPGA and in the PC in such a way that: a) blocks of data are sorted in the PL of APSoC or in FPGA; b) the sorted blocks are transferred to the PC through PCI-express; and c) the blocks are merged by software in the PC (components A and C). This case is indicated below as *Sorting + PC merge*.

Sorting in hardware only (see point 1 above) permits the circuits that process the maximum possible number of data items and can be entirely implemented in the programmable logic without any support from software to be evaluated. In the next subsections we will present the following results: 6.1) evaluation of the circuits including threshold values that are potential limitations of the methods proposed; 6.2) comparisons with the best known alternatives.

6.1. **Evaluation of the proposed circuits.** Evaluation of the proposed circuits has been done through a set of experiments with the network from Figure 4, selecting four data sets sizes of 512, 1024, 2048, and 4096 items. The results are shown in Figure 13.

We counted only the percentage of look-up tables (LUTs), which are the primary PL/FPGA resources that are used for the network. The percentage of other resources is lower, for example, the percentage of flip-flops for the FPGA does not exceed 23% and

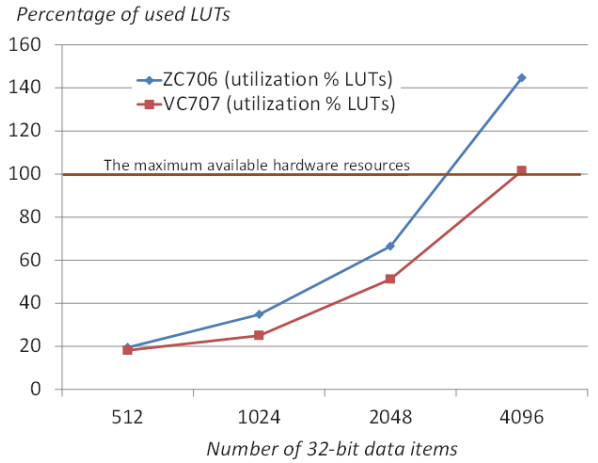

FIGURE 13. The results of sorting in hardware only using iterative networks from Figure 4

for the PL – 31% for all data set sizes (from 512 to 4096). From Figure 13 we can see that the available resources permit only iterative networks of up to 2048 32-bit data items to be implemented. Thus 2048 is the threshold for hardware only implementations based on the microchips indicated above. A preliminary evaluation shows that 8192 items is the maximum threshold value for hardware-only implementations of the circuit from Figure 4 in the most advanced FPGAs/APSoCs currently available on the market.

The circuit in Figure 5 was also implemented and tested. As we expected in Section 4, the occupied resources were reduced, and the time was increased compared to the network in Figure 4 by a factor of about M. For example, if the size of one block is 1024 ($N = 1024$) of 32-bit ($M = 32$) items, then the percentage of LUTs used for VC707 is 25% for the network in Figure 4, and 18% for the network in Figure 5. Thus, sorting 4096 items in the FPGA of VC707 is possible. However, we found that the remaining resources are not sufficient to implement the other blocks shown in Figure 14. Thus, the circuit in Figure 5 makes sense only for autonomous hardware networks.

In further experiments (see Figure 12) only the network from Figure 4 will be used. The size of data varies from 2 KB to 1024 KB ($M = 32$). The results obtained for the four measurements indicated above are reported in Figure 14 (the two curves *PC sort* and *PC sort + data transfer* show the same results without and with data transfers). The result for each type of experiment is an average of 64 runs.

The following conclusions can be drawn from Figure 14.

- The fastest results were obtained for the components A and C, i.e., pre-sort in the PL with a subsequent merge in the PC (see point 4 above). Note that the fastest (the lowest) curve in Figure 14 is built for sorting individual subsets only. Thus, the complete data set has not been sorted and the relevant results cannot be used for comparisons.
- The slowest result is shared between the remaining two cases (see points 2, 3 above).
- Note that for almost all data sizes, sorting and merging in APSoC is faster than sorting in PC software. Thus, cheaper (than PC) APSoCs are more advantageous and may be used efficiently for embedded applications.
- Sorting blocks in the PL network (see Figure 4) is significantly faster than subsequent merging. All communication and protocol overheads were taken into account.

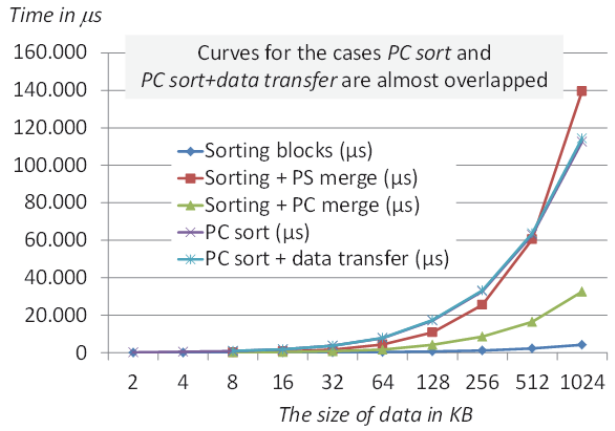

FIGURE 14. The results of experiments with the three-level system sorting data (the size of one block is 1024 32-bit data items)

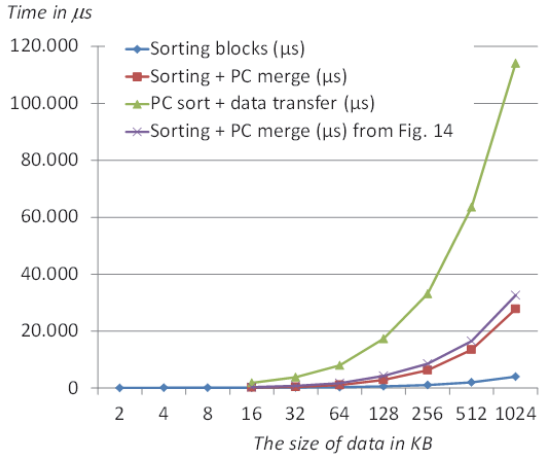

FIGURE 15. The results of experiments with the two-level system sorting data (the size of one block is 2048 32-bit data items)

Similar experiments were done with the VC707 prototyping board, but with the blocks of data containing 2048 32-bit data items (i.e., the blocks sorted in the hardware network are two times larger). The results are shown in Figure 15.

From analyzing these results we can conclude that:

- Using an FPGA from the Virtex-7 family, sorting in hardware networks is slightly faster, but the difference is negligible;
- Using larger blocks (2048 vs. 1024) allows sorting in point 4 (see the beginning of this section) to be faster by a factor ranging from 1.2 to 1.8. This is because the depth of software merges is reduced by one level.

The next experiments were done extracting the maximum and the minimum sorted subsets. We found that the acceleration is better than in Figures 14 and 15 for data sorting. This is because the number of data transferred through PCI express is significantly decreased and almost all operations are done in the APSoC/FPGA. We implemented and tested the circuit shown in Figure 7 in the PL of APSoC, which takes data from the DDR memory and extracts the maximum and minimum subsets with $L_{\max}/L_{\min}$ data items, where $L_{\max}/L_{\min}$ varies from 128 to 1024 (as before $M = 32$, L varies from 2 KB to 1024 KB). Table 1 presents the results for $L_{\max}/L_{\min} = 128$.

TABLE 1. The results of experiments extracting the maximum/minimum subsets

Data (KB)	Time (μs)	Data (KB)	Time (μs)
2	70	64	254
4	75	128	425
8	89	256	916
16	112	512	1543
32	157	1024	3535

For some practical applications the maximum and/or minimum subsets may be large and the available hardware resources become insufficient to implement the circuit from Figure 7 [54]. The problem can be solved by applying the following technique. Let $l_{\max}$ and $l_{\min}$ be constraints for the upper and lower parts of the sorting network in Figure 7,

i.e., circuits with larger values (than $l_{\max}$ and $l_{\min}$) cannot be implemented due to the lack of hardware resources or for 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 this case, the maximum/minimum subsets can be computed incrementally as follows [54].

1. In 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. This correction avoids the loss of repeated items in subsequent steps. Indeed, the minimum value from the maximum subset (the maximum value from the minimum subset) can appear in subsets that are generated in point 3 below, and they will be lost because of filtering (see point 3 below).
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 a general-purpose port.
3. The same data items (from memory), as in point 1 above, are initially filtered [55] so that only items that are less than B_u and greater than B_l are allowed to be processed, i.e., computing sorted subsets can be done only for the filtered data items. Thus, the second part of the maximum and minimum subsets will be computed and appended (in the PS) to the previously computed subsets (such as the subsets from point 1).

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.

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 becomes empty after the corrections in point 1 above, then an infinite loop will be created. In this case, we can use the previously described method based on software/hardware sorters (i.e., sorting in hardware and subsequent merging in software). Thus, data items are sorted before the desired number of the largest and/or the smallest items are taken.

Table 2 presents the results for larger numbers of data items in extracted subsets (from 128 to 1024) for $L = 256$ KB.

TABLE 2. The results of experiments with extracting subsets with different number of data items

Data	Time (μs)	Data	Time (μs)
$128 + 128$	916	$640 + 640$	4481
$256 + 256$	1808	$768 + 768$	5372
$384 + 384$	2698	$896 + 896$	6261
$512 + 512$	3589	$1024 + 1024$	7152

The developed software and hardware can also solve higher level tasks. As examples, we considered creating objects in software for further clustering and finding the frequency of occurrence of data items in hardware. The attributes of any individual object are generated randomly in software within a given range. Objects and attributes are associated with rows and columns of a matrix. Clearly, the Hamming weight of any row r 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 [2]; 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.

Finding the item that occurs most frequently can be done entirely in hardware. Suppose we have a set of L sorted data items which may include repeated items and we need the most frequently repeated item to be found. This problem is solved in the hardware circuit proposed in [2]. Thus, combining the proposed solutions with the circuit [2] enables the complete problem to be solved.

6.2. **Comparisons with the best known alternatives.** Comparisons with the best known alternatives can be done by analyzing the fastest known networks. For data sorting, the latency and the cost of the most widely discussed networks are shown in Table 3. The formulae for the table are taken from [1,23,25,26,32]. For example, if $N = 1024$ then the latency is equal to $D(1024) = 55$ for the fastest known even-odd merge and bitonic merge networks [28,29], which is smaller than the number of iterations for the proposed network. However, $C(1024)$ for the less resource consuming even-odd merge network is 24,063 C/S and for the proposed network $C(1024) = 1023$ C/S. Thus, the difference is a factor of about 24. It means that with the same hardware resources, the proposed networks can process blocks of data with significantly larger number N of data items. Indeed, the resources $C(1024) = 24,063$ of the known even-odd merge network are the same as for 24 proposed networks each of which sorts the same number of data items, i.e., 1024. This means that the proposed network occupies less than 5% of the resources of the known network and the number of sorted items is exactly the same.

TABLE 3. Cost $C(N)$ and latency $D(N)$ of the most widely discussed networks

Type of the network	$C(N)$	$D(N)$
Bubble and insertion sort	$N \times (N-1)/2$	$2 \times N - 3$
Even-odd transition	$N \times (N-1)/2$	N
Even-odd merge	$(p^2 - p + 4) \times 2^{p-2} - 1$, $N = 2^p$	$p \times (p+1)/2$, $N = 2^p$
Bitonic merge	$(p^2 + p) \times 2^{p-2}$, $N = 2^p$	$p \times (p+1)/2$, $N = 2^p$
The proposed network (see Figure 4)	$N - 1$	$\leq N$

The experiments done for the board [8] have shown that for the networks [28,29] $N \leq 128$, while for the proposed networks $N > 2048$. Thus, the proposed networks may handle about 16 times larger blocks. The blocks created in hardware are further merged in software, thus the number of levels in software will be increased in the known networks by a factor of $\lceil \log_2 16 \rceil = 4$ (comparing to the proposed network). The following experiments were done:

1. Blocks with two sizes (that are 128 and 2048 32-bit words) have been sorted in software using the known (for the size 128) and the proposed (for the size 2048) networks. The measured times are T_{128} and T_{2048}.
2. Since the known networks cannot be used for $N = 2048$, the same results have been obtained through a subsequent merge in software of blocks with $N = 128$ to get blocks with $N = 2048$. The measured time is $T_{128} + T_{\text{merge}}$.
3. Finally we measured the value $(T_{128} + T_{\text{merge}})/T_{2048}$. The fastest method was used i.e., pre-sort in the PL with subsequent merge in the PC (see Subsection 6.1). The result that was an average of 64 runs exceeds 5. Note that additional delays appeared also in data transmission through PCI-express of smaller blocks of data items.

For subsequent merging required for larger data sets all the conditions for the proposed and known methods are the same. Thus, the proposed methods are always faster because merging in software begins with significantly larger pre-sorted blocks. Clearly, threshold

values for maximum sizes of sorted sets are the same as for general-purpose software running in a PC.

Comparison of the proposed methods for extracting the maximum and minimum sorted subsets with the results in [53] demonstrates that the proposed method permits significantly larger subsets to be constructed. Indeed, the maximum size of extracted subsets in [53] is only 8 items and the maximum size of initial set is only 256 items. The size of each item is 10 bits. This is because the methods [53] are based on even-odd merge and bitonic merge networks for which the complexity of the circuits, i.e., the value of $C(N)$, is limited. In our case, the maximum size of extracted subsets is 1024 (which exceeds the size of initial data sets in [53]) and the size of initial set is up to 1024 KB. The size of each item is 32 bits (versus 10 in [53]). The conclusion is the following: 1) the proposed methods enable data sets with significantly larger numbers of items to be processed; 2) the size of the extracted (minimum, maximum, or both) subsets may be increased in the proposed networks; 3) the performance (throughput) for processing large subsets in the proposed methods is better because complex tasks cannot be entirely solved in hardware using the methods [53] and the necessary software introduces large additional delays.

7. **Conclusions.** The paper is dedicated to distributed computing systems that involve higher level computers (such as a PC) interacting with programmable systems-on-chip through a PCI-express bus. We studied different levels of such systems, namely higher level software running in the host PC, data transfer through PCI-express, lower level software running in ARM of a programmable system-on-chip, and hardware accelerators implemented in the programmable logic. It is shown that sorting and searching are common operations in different types of data and information processing. We found the fastest way to implement such operations and suggested numerous supplementary operations that are common to different computing systems. A number of hardware accelerators were proposed, all of which were completely implemented and tested in commercial computers and microelectronic devices. The practical analysis consisted of numerous experiments using the most recent all programmable systems-on-chip combining a processing system with configurable logic. The experiments comprehensively demonstrated that the proposed multilevel solutions outperformed software running in a PC by a significant margin. It is also shown that the networks proposed can be used in numerous practical applications in control engineering and applied informatics.

Acknowledgments. The authors would like to thank Ivor Horton for his very useful comments. This research was supported by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, the Study IT in Estonia Programme, and Portuguese National Funds through FCT – Foundation for Science and Technology, in the context of the projects UID/CEC/00127/2013 and Incentivo/EEI/UI0127/2014.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching, Vol. III*, Addison-Wesley, 2011.

[2] V. Sklyarov and I. Skliarova, Digital hamming weight and distance analyzers for binary vectors and matrices, *International Journal of Innovative Computing, Information and Control*, vol.9, no.12, pp.4825-4849, 2013.

[3] D. Zmaranda, H. Silaghi, G. Gabor and C. Vancea, Issues on applying knowledge-based techniques in real-time control systems, *International Journal of Computers, Communications and Control*, vol.8, no.1, pp.166-175, 2013.

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

[5] W. Zhang, K. Thurow and R. Stoll, A knowledge-based telemonitoring platform for application in remote healthcare, *International Journal of Computers, Communications and Control*, vol.9, no.5, pp.644-654, 2014.
[6] Z. K. Baker and V. K. Prasanna, An architecture for efficient hardware data mining using reconfigurable computing systems, *Proc. of the 14th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, pp.67-75, 2006.
[7] Xilinx, Inc., *ZC706, All Programmable SoC Evaluation Kit (Vivado Design Suite 2014.3)*, http://www.xilinx.com/support/documentation/boards_and_kits/zc706/2014_3/ug961-zc706-GSG.pdf.
[8] Xilinx, Inc., *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*, http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf.
[9] R. Rajkumar, I. Lee, L. Sha and J. Stankovic, Cyber-physical systems: The next computing revolution, *Proc. of the 47th ACM/IEEE Design Automation Conference*, pp.731-736, 2010.
[10] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*, 2nd Edition, 2011.
[11] J. C. Jensen, E. A. Lee and S. A. Seshia, *An Introductory Lab in Embedded and Cyber-Physical Systems*, http://leeseshia.org/lab, 2014.
[12] K. Vipin, S. Shreejith, S. A. Fahmy and A. Easwaran, Mapping time-critical safety-critical cyber physical systems to hybrid FPGAs, *Proc. of the 2nd IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, pp.31-36, 2014.
[13] M. Panunzio and T. Vardanega, An architectural approach with separation of concerns to address extra-functional requirements in the development of embedded real-time software systems, *Journal of Systems Architecture*, vol.60, pp.770-781, 2014.
[14] A. Borangiu and D. Popescu, Digital signal processing for knowledge based sonotubometry of eustachian tube function, *Journal of Control Engineering and Applied Informatics*, vol.16, no.3, pp.56-64, 2014.
[15] Y. Benmoussa, J. Boukhobza, E. Senn, Y. Hadjadj-Aoul and D. Benazzouz, A methodology for performance/energy consumption characterization and modeling of video decoding on heterogeneous SoC and its applications, *Journal of Systems Architecture*, vol.61, pp.49-70, 2015.
[16] M. Santarini, Products, profits proliferate on ZynqSoC platforms, *XCell Journal*, vol.88, pp.8-15, 2014.
[17] M. Santarini, Xilinx 16nm UltraScale+ devices yield 2-5X performance/watt advantage, *XCell Journal*, vol.90, pp.8-15, 2015.
[18] Xilinx Inc., *Zynq-7000 All Programmable SoC Technical Reference Manual*, http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
[19] J. Silva, V. Sklyarov and I. Skliarova, Comparison of on-chip communications in Zynq-7000 all programmable systems-on-chip, *IEEE Embedded Systems Letters*, vol.7, no.1, pp.31-34, 2015.
[20] R. Budruk, D. Anderson and T. Shanley, *PCI-Express System Architecture*, Addison-Wesley, 2008.
[21] N. Edwards, *Theoretical vs. Actual Bandwidth: PCI Express and Thunderbolt*, http://www.tested.com/tech/457440-theoretical-vs-actual-bandwidth-pci-express-and-thunderbolt/, 2013.
[22] J. Lawley, *Understanding Performance of PCI Express Systems*, http://www.xilinx.com/support/documentation/white_papers/wp350.pdf, 2014.
[23] R. Mueller, J. Teubner and G. Alonso, Sorting networks on FPGAs, *The International Journal on Very Large Data Bases*, vol.21, no.1, pp.1-23, 2012.
[24] J. Ortiz and D. Andrews, A configurable high-throughput linear sorter system, *Proc. of IEEE Int. Symp. on Parallel & Distributed Processing*, pp.1-8, 2010.
[25] R. Mueller, *Data Stream Processing on Embedded Devices*, Ph.D. Thesis, ETH, Zurich, 2010.
[26] V. Sklyarov and I. Skliarova, High-performance implementation of regular and easily scalable sorting networks on an FPGA, *Microprocessors and Microsystems*, vol.38, no.5, pp.470-484, 2014.
[27] M. Zuluada, P. Milder and M. Puschel, Computer generation of streaming sorting networks, *Proc. of the 49th Design Automation Conference*, pp.1245-1253, 2012.
[28] K. E. Batcher, Sorting networks and their applications, *Proc. of AFIPS Spring Joint Computer Conference*, pp.307-314, 1968.
[29] S. W. Aj-Haj Baddar and K. E. Batcher, *Designing Sorting Networks: A New Paradigm*, Springer, 2011.
[30] S. Lacey and R. Box, A fast, easy sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines, *Byte*, vol.16, no.4, pp.315-320, 1991.

[31] R. D. Chamberlain and N. Ganesan, Sorting on architecturally diverse computer systems, *Proc. of the 3rd International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pp.39-46, 2009.

[32] P. Kipfer and R. Westermann, *GPU Gems*, Improved GPU Sorting, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.

[33] C. Grozea, Z. Bankovic and P. Laskov, FPGA vs. multi-core CPUs vs. GPUs, in *Facing the Multicore-Challenge*, R. Keller, D. Kramer and J. P. Weiss (eds.), Springer-Verlag, 2010.

[34] B. Cope, P. Y. K. Cheung, W. Luk and L. Howes, Performance comparison of graphics processors to reconfigurable logic: A case study, *IEEE Trans. Computers*, vol.59, no.4, pp.433-448, 2010.

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

[36] V. Sklyarov, I. Skliarova, J. Silva and A. Sudnitson, Design space exploration in multi-level computing systems, *Proc. of the 15th International Conference on Computer Systems and Technologies*, pp.40-47, 2014.

[37] S. Sun, *Analysis and Acceleration of Data Mining Algorithms on High Performance Reconfigurable Computing Platforms*, Ph.D. Thesis, Iowa State University, 2011.

[38] S. Sun and J. Zambreno, Design and analysis of a reconfigurable platform for frequent pattern mining, *IEEE Trans. Parallel and Distributed Systems*, vol.22, no.9, pp.1497-1505, 2011.

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

[40] M. F. Firdhous, Automating legal research through data mining, *International Journal of Advanced Computer Science and Applications*, vol.1, no.6, pp.9-16, 2010.

[41] E. Kovacs and I. Ignat, Clustering with prototype entity selection compared with K-means, *Journal of Control Engineering and Applied Informatics*, vol.9, no.1, pp.11-18, 2007.

[42] Intel, Corp., *Intel® SSE4 Programming Reference*, http://home.ustc.edu.cn/∼shengjie/REFERENCE/sse4_instruction_set.pdf, 2007.

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

[44] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger and W. Lehner, An application-specific instruction set for accelerating set-oriented database primitives, *Proc. of ACM SIGMOD International Conference on Management of Data*, pp.767-778, 2014.

[45] N. Abdelhamid, Multi-label rules for phishing classification, *Applied Computing and Informatics*, vol.11, pp.29-46, 2015.

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

[47] J. W. Tukey, *Exploratory Data Analysis*, Addison-Wesley, 1977.

[48] A. P. Batista and F. G. Jota, Effects of time delay statistical parameters on the most likely regions of stability in an NCS, *Journal of Control Engineering and Applied Informatics*, vol.16, no.1, pp.3-11, 2014.

[49] R. Sedgewick, Implementing quicksort programs, *Communications of the ACM*, vol.21, no.10, pp.847-857, 1978.

[50] Y. Yuan, Step-sizes for the gradient method, *Proc. of the 3rd International Congress of Chinese Mathematicians*, pp.785-796, 2008.

[51] A. Goodman, Perspective emerging topics and challenges for statistical analysis and data mining, *Statistical Analysis and Data Mining*, vol.4, pp.3-8, 2011.

[52] S. J. Piestrak, Efficient hamming weight comparators of binary vectors, *Electronic Letters*, vol.43, no.11, pp.611-612, 2007.

[53] A. Farmahini-Farahani, H. J. Duwe, M. J. Schulte and K. Compton, Modular design of high-throughput, low-latency sorting units, *IEEE Trans. Computers*, vol.62, no.7, pp.1389-1401, 2013.

[54] V. Sklyarov, I. Skliarova, A. Rjabov and A. Sudnitson, Zynq-based system for extracting sorted subsets from large data sets, *Journal of Microelectronics, Electronic Components and Materials*, vol.45, no.2, pp.142-152, 2015.

[55] V. Sklyarov, I. Skliarova, A. Rjabov and A. Sudnitson, Computing sorted subsets for data processing in communicating software/hardware control systems, *International Journal of Computers Communications & Control*, vol.11, no.1, pp.126-141, 2016.

[56] Xilinx, Inc., *AXI Central Direct Memory Access v4.1*, http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf, 2015.

[57] Xilinx, Inc., *LogiCORE IP AXI Bridge for PCI Express v1.06*, http://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_5/pg055-axi-bridgepcie.pdf, 2012.
[58] Xilinx, Inc., *PCI Express Endpoint-DMA Initiator Subsystem*, http://www.xilinx.com/support/documentation/application_notes/xapp1171-pcie-central-dma-subsystem.pdf, 2013.
[59] J. Corbet, A. Rubini and G. Kroah-Hartman, *Linux Device Drivers*, http://lwn.net/Kernel/LDD3/.

PUBLICATION VII

Sklyarov, V.; Skliarova, I.; **Rjabov, A**.; Sudnitson, A. (2016). Computing Sorted Subsets for Data Processing in Communicating Software/Hardware Control Systems. International Journal of Computers Communications & Control,11 (1), 126−141, 10.15837/ijccc.2016.1.

INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 11(1):126-141, February 2016.

Computing Sorted Subsets for Data Processing in Communicating Software/Hardware Control Systems

V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson

V. Sklyarov*, I. Skliarova
University of Aveiro, Dept. of Electronics, Telecommunications and Informatics/IEETA
Campus Universitário de Santiago, 3810-193, Aveiro, Portugal
*Corresponding author: skl@ua.pt

A. Rjabov, A. Sudnitson
Tallinn University of Technology, Dept. of Computer Engineering
Akadeemia tee 15A, 12618, Tallinn, Estonia

Abstract: Computing and filtering sorted subsets are frequently required in statistical data manipulation and control applications. The main objective is to extract subsets from large data sets in accordance with some criteria, for example, with the maximum and/or the minimum values in the entire set or within the predefined constraints. The paper suggests a new computation method enabling the indicated above problem to be solved in all programmable systems-on-chip from the Xilinx Zynq family that combine a dual-core Cortex-A9 processing unit and programmable logic linked by high-performance interfaces. The method involves highly parallel sorting networks and run-time filtering. The computations are done in communicating software, running in the processing unit, and hardware, implemented in the programmable logic. Practical applications of the proposed technique are also shown. The results of implementation and experiments clearly demonstrate significant speed-up of the developed software/hardware system comparing to alternative software implementations.
Keywords: computing sorted subsets, communicating hardware/software systems, filtering, sorting networks, control applications.

1 Introduction

Many electronic, environmental, medical, and biological control applications need to process data streams produced by sensors and measure external parameters within given upper and lower bounds (thresholds) [1]. Let us consider some examples. Applying the technique [2] in real-time applications requires knowledge acquisition from controlled systems (e.g. plant). For example, signals from sensors may be filtered and analyzed to prevent error conditions (see [2] for additional details). To provide more exact and reliable conclusion, combination of different values need to be extracted, ordered, and analyzed. Similar tasks appear in monitoring thermal radiation from volcanic products [3], filtering and integration of information from a variety of different sources in medical applications [4] and in other practical applications described in [5]. Since many control systems are real-time, performance is important and hardware accelerators may provide significant assistance for software. A similar data processing is applicable to data mining algorithms, such as [6].

Let us consider control systems that collect, filter and analyze data produced by some measurements. We will describe below such computations that permit:

- the maximum and/or minimum sorted subsets to be extracted (the maximum/minimum sorted subset of size L_{max}/L_{min} contains L_{max}/L_{min} data items with maximum/minimum values from a given set);
- the maximum and/or minimum sorted subsets to be found within the given upper B_u and lower B_l bounds.

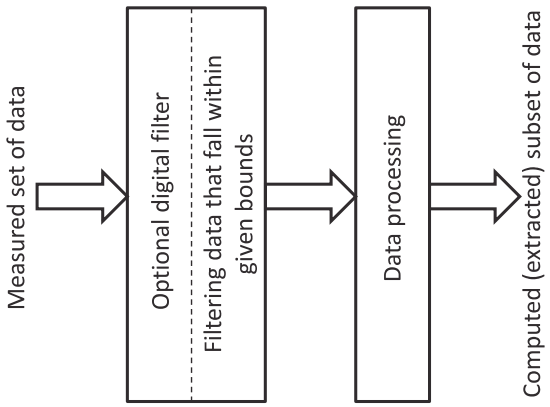

Figure 1: General architecture of data processing

The problem can be solved as it is shown in Fig. 1.

There are two blocks in Fig. 1. Measured data items are handled in such a way that the maximum and/or minimum subsets with L_{max} and/or L_{min} items are extracted by the data processing block. Input data may optionally be filtered allowing only items (such as D) that fall within pre-given constraints (e.g. $B_l \leq D \leq B_u$ or $B_l < D < B_u$) to be processed.

The paper suggests a method and high-performance implementation of architecture in Fig. 1 in all programmable systems-on-chip (APSoC) from the Xilinx Zynq-7000 family [7] that are recently developed field-configurable devices integrating the most advanced programmable logic (PL) and a widely used processing system (PS) based on the dual-core ARM® Cortex™ MP-Core™. The available interfaces between the PS and PL are supported by ready-to-use intellectual property (IP) cores. These, combined with numerous architectural and technological advances, have enabled APSoCs to open a new era in the development of highly optimized computational systems for a vast variety of practical applications, including high-performance computing, data, signal and image processing, control, and many others. The main target of APSoCs is integration in the developed systems of software and hardware components assuming that such integration enables characteristics (most often performance) of the system to be improved. The complexity of hardware only solutions is frequently limited by the available resources in the PL. Software/hardware solutions can be very complex and they are appropriate for control applications, such as that are described, for example, in [2,4]. The most close related work can be found in [5,8] where the importance of the considered problem is underlined, but the methods that allow the problem to be solved are different and the proposed below methods permit better results to be achieved.

The remainder of the paper is organized in eight sections. Section 2 presents the proposed software/hardware architecture. Section 3 describes a novel method allowing the maximum and minimum sorted subsets for a given set of data items to be computed. Section 4 suggests a run-time filtering method. Section 5 is dedicated to on-chip communication mechanisms linking software and hardware components. Section 6 shows how large subsets (for which hardware resources are not sufficient) can be computed and discusses additional capabilities such as extracting only the maximum or only the minimum subsets. Section 7 demonstrates potential practical application (from the areas of control and data mining). Implementations in Zynq microchips and the results of thorough evaluation and comparison of software only and software/hardware solutions with explicit indication of the achieved acceleration are discussed in section 8. Section 9 concludes the paper.

2 Software/Hardware Architecture

Fig. 2 presents the proposed software/hardware architecture.

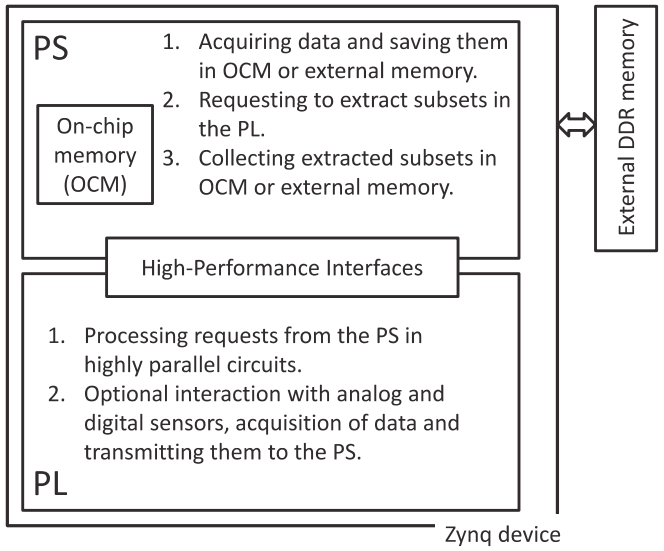

Figure 2: The proposed software/hardware architecture

The PS collects data, that may be acquired from different sources (such as from a host PC or from sensors connected to a Zynq device), and stores them in on-chip or external memory. The PL processes requests from the PS, that is, reads data from memories, and rapidly extracts the maximum and/or minimum subsets. Both parts, that are the PS and PL, may function in parallel and any request can be seen as a macroinstruction executed in the PL concurrently with other potential instructions in the PS.

It is shown in [9] that for transferring a small number of data items between the PS and the PL on-chip general-purpose ports (GPP) can be used more efficiently than other available interfaces. Thus, requests from the PS to the PL are formed through GPP where the PS is the master and the PL is a slave. It is also shown in [7, 9] that large volumes of data can be more efficiently transferred from/to memories to/from the PL through high performance (HP) interfaces: High-Performance Advanced eXtensible Interface (AXI HP) and AXI Accelerator Coherency Port (AXI ACP). In all our designs memories are slaves and either the PL or the processor in the PS is the master. To increase performance, data from memories may be requested to be cacheable.

3 Computing 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. Since N may have very large value (millions of items) it cannot be processed completely in hardware due to the unavailability of sufficient resources. It is shown in [10, 11] that even for relatively complex Field-Programmable Gate Arrays (FPGAs) the size

N is limited. For example, for even-odd merge and bitonic merge networks [12] N 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). In Zynq devices implementing circuits from [12] the maximum value of N does not exceed 128 32-bit items. Iterative even-odd transition networks from [11] permit significantly larger number of items (exceeding thousands of 32-bit items) to be processed and they will be used for computing sorted subsets in hardware. However, in practical cases the given sets anyhow cannot be entirely processed and computing the maximum and/or minimum sorted subsets needs to be done sequentially, nevertheless handling many items in parallel. Fig. 3 depicts the proposed architecture that enables the considered problem to be solved.

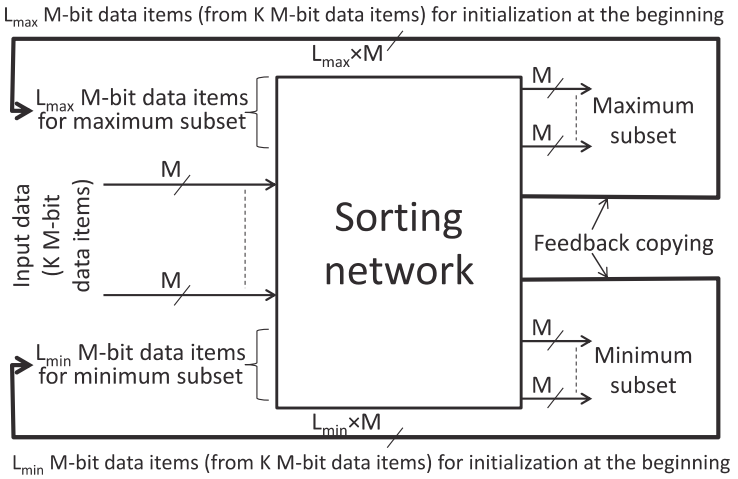

Figure 3: Computing the maximum and the minimum sorted subsets

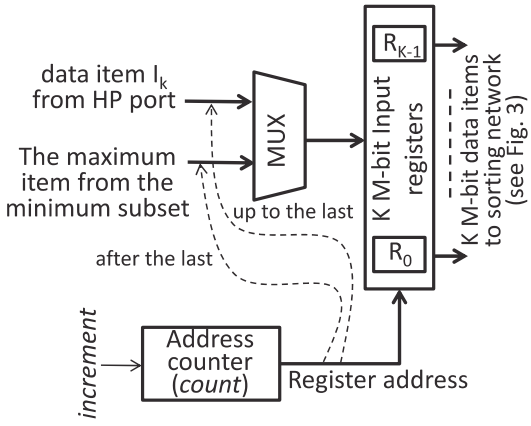

Figure 4: Processing the last (possibly incomplete) subset

Let us divide the given set S into Q = ⌈N/K⌉ subsets, all of which contain exactly K M-bit

items except the last one, which may have less than K M-bit items. Computing subsets is done incrementally in Q steps (we assume below that $K \leq N$).

At the first step, the first K M-bit data items are sorted in the network [11] which processes L_{max}+K+L_{min} data items but comparators linking the upper part (handling L_{max} M-bit data items) and the lower part (handling L_{min} M-bit data items) are deactivated (i.e. the links with the upper and bottom parts are broken). So, sorting is done only in the middle part handling K M-bit items. As soon as the sorting is completed, the maximum subset is copied to the upper part of the network and the minimum subset is copied to the lower part of the network (see Fig. 3).

From the second step, all the comparators are properly linked, i.e. the network from [11] handles L_{max}+K+L_{min} items, but the feedback copying (see the first step and Fig. 3) is disabled. Now for each new K M-bit items the maximum and the minimum sorted subsets are appropriately corrected, i.e. new items may be appended.

At the last step, the number of incoming items may be less than K. Fig. 4 explains how the maximum and minimum subsets are corrected for the last possibly incomplete subset of items. There is an additional MUX in Fig. 4, which supplies data items from a HP port (linking the PL with memory) until the received item is not the last. As soon as the last item is read from memory, the next items (until K) are taken as the maximum value from the minimum subset (see the lower subset in Fig. 3). Clearly, such an item cannot be moved again to the minimum subset and the last sorting step is executed similarly to the previous steps.

Let us look at the example shown in Fig. 5 for which: N = 21, K = 8, $L_{max} = L_{min} =$ 4, and S = 26,37,11,19,3,7,99,56,29,37,22,99,1,55,39,47,12,45,83,5,18. The set S is divided into the following three subsets: A = 26,37,11,19,3,7,99,56, B = 29,37,22,99,1,55,39,47, and C = 12,45,83,5,18.

Note that the last subset C contains only 5 elements and is incomplete. Symbol U in Fig. 5 indicates undefined value. The iterative sorting network is exactly the same as in [11]. Any comparator is shown in Knuth notation [13] and it converts two-item inputs in two-item outputs in such a way that the upper value is greater than or equal to the lower value. The maximum number of iterations for sorting is K/2 [14] and this number is almost always smaller because the method [11] terminates subsequent iterations as soon as all items are sorted. There are 3 steps in Fig. 5. At the first step, K (K=8) items are sorted and copied to the maximum and minimum subsets.

Two comparators are disabled in accordance with the explanations given above (breaking links of the middle section in the sorted network with the upper and the lower sections). At the second step, all the network comparators are enabled and L_{max}+K+L_{min} items are sorted by the iterative network with feedback register (FR). All necessary details can be found in [11]. It is easy to show that the maximum number of iterations is $\lceil(\max(L_{max},L_{min})+K)/2\rceil$ and much like the previous case this number is almost always smaller [11]. At the last (third) step, the incomplete subset C is extended to K items by copying the maximum value (11) from the minimum subset 11,7,3,1 to the positions of missing data (see Fig. 5). After sorting L_{max}+K+L_{min} items at the step 3 the final result is produced.

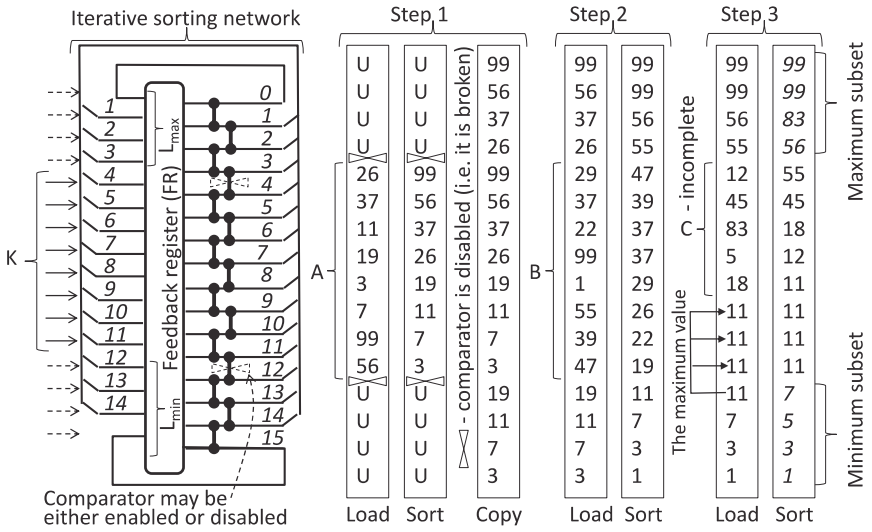

Figure 5: An example (computing subsets)

4 Filtering

Let B_u and B_l be predefined upper (B_u) and lower (B_l) bounds for the given set S. We would like to use the circuit in Fig. 3 only for such data items D that fall within the bounds B_u and B_l, i.e. $B_l \leq D \leq B_u$ (or, possibly, $B_l < D < B_u$). Fig. 6 depicts the proposed architecture that enables data items to be filtered at run-time (i.e. during the data exchange between the PS and PL). There is an additional block on the upper input of the MUX (see also Fig. 4), which takes a data item I_k from a HP port and executes the operation indicated on the right-hand part of Fig. 6. If the counter is incremented, then a new register is chosen to store I_k. Otherwise, the signal WE (write enable) is passive and a new item with a value that is out of the bounds B_u and B_l is not recorded in the registers.

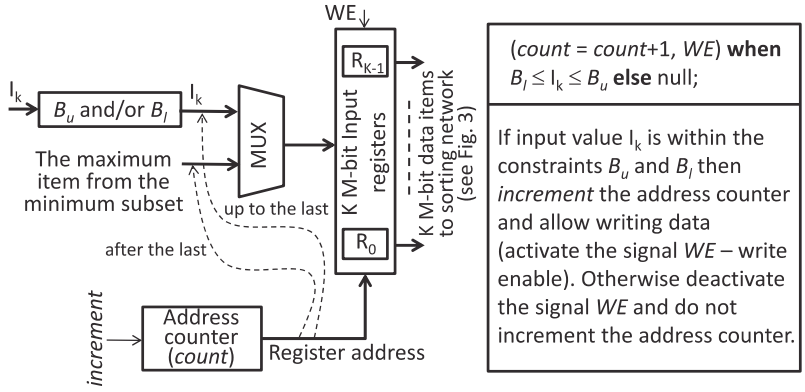

Figure 6: Digital filter

Let us look at the same example in Fig. 5 for which we choose $B_u = 90$ and $B_l = 10$ (see Fig. 7). At the first step incoming data items have preliminary been filtered, the values 99, 7, and 3 have been removed (because they are either greater than $B_u = 90$ or less than $B_l = 10$), and the subset A with 8 items is built from 11 first elements of the set S. At the second (last) step, the values 99, 1, and 5 have been removed, and the subset B = 55,39,47,12,45,83,18 is built from the remaining allowed elements of the set S. Since there are 7 items in B and K = 8, this subset is incomplete.

As can be seen from Fig. 7, two steps are sufficient to extract the maximum and the minimum subsets from the filtered set S. Similarly, filtering and computing sorted subsets can be done for very large data sets.

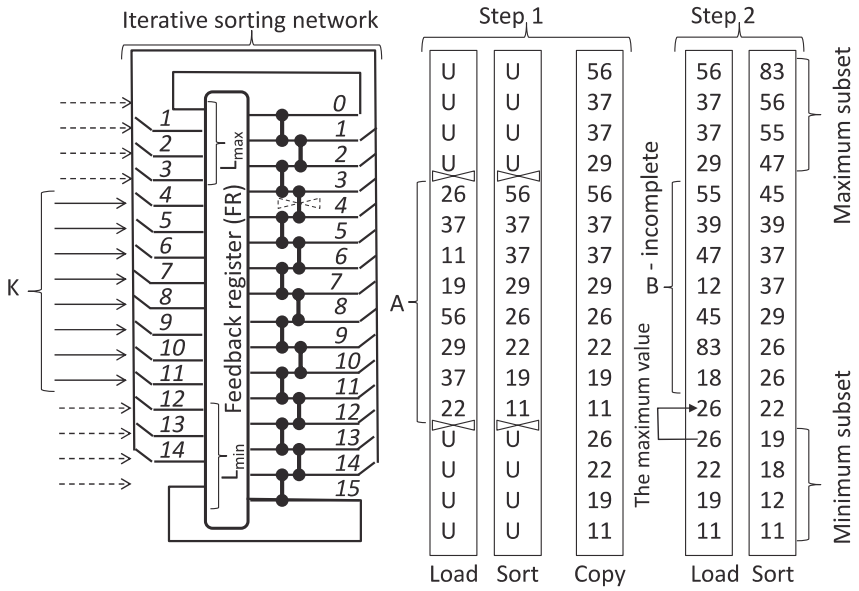

Figure 7: An example (filtering and computing subsets)

Clearly, the described above operations can be done in software. For example, C function qsort permits large data sets to be sorted. After that extracting the maximum and minimum subsets may easily be done. Filtering can be provided much like it is shown in Fig. 6 eliminating items that do not fall within the predefined constraints. However, for many practical applications performance of the described above operations is important. To evaluate software/hardware solutions three different components need to be taken into account (see Fig. 8): 1) software part; 2) hardware part; and 3) the circuits that provide for data exchange between software and hardware. Numerous experiments were done in [15] to compare such solutions with software only systems. One example in [15] enables sorting blocks of data composed of 320 32-bit items in the PL that are further merged in the PS (see Fig. 8). From 512,288 to 4,194,304 of 32-bit data items were randomly generated in the PS (i.e. the size of data varies from 2MB to 16MB) and then sorted in software with the aid of the function qsort and in the software/hardware system (see Fig. 8). The actual performance improvement was by a factor of about 2.5. It was shown in [15] that hardware circuits in the PL are significantly faster than software in the PS. In this paper we evaluate and compare software/hardware and software only solutions taking into account all the

involved communication overheads that were measured in [15]. We will mainly use AXI ACP [7] which provides one of the fastest interfaces for exchange of large data sets between the PS and PL [7, 9, 15]. The number of data items transferred from the PS/memory to the PL is the same as in [15]. However, the number of data items transferred from the PL to the PS/memory is significantly smaller enabling much better acceleration to be achieved.

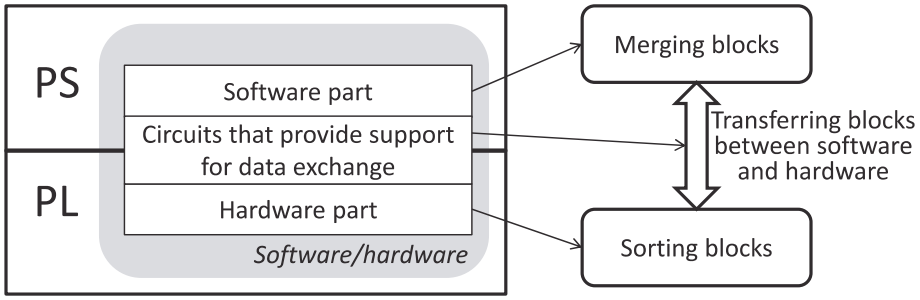

Figure 8: An example of a software/hardware system

5 Communication of Software and Hardware

Fig. 9 shows how communication is organized between software and hardware. It is done similarly to [8,15], but the proposed in this paper processing is different. The developed hardware in the PL is divided in two parts: application-specific (that is filtering and computing subsets) and communication-specific processing. The latter is studied in [15] and provides support for data exchange with storage of the PL that is either block RAM or registers built from flip-flops of configurable logic blocks.

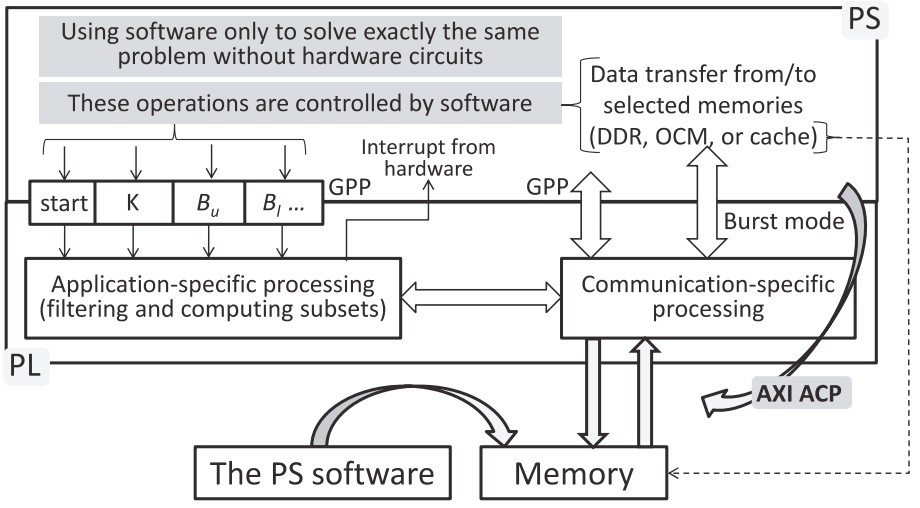

Figure 9: Communication between software and hardware components

Data are transmitted in blocks of 32/64-bit items (i.e. either M=32 or M=64) and the fastest burst mode is applied. Input data items I_k (k=0,1,...,K-1) are processed by the described above circuits (see Fig. 3, 4, 6). Note that GPP do not allow burst mode to be applied but are very appropriate for transferring small number of signals that may be used for control in the PL and for some additional details. In our system they are:

1) start requiring data processing in the PL to be initiated;

2) K (see Fig. 3);

3) B_u and B_l (see Fig. 6);

4) additional signals, namely source address, destination address and sizes of data to be transferred from the PS to the PL and vice versa.

Fig. 10 demonstrates a component diagram for reading the initial large volume data from the PS and for transferring the results (i.e. the computed maximum and minimum subsets) from the PL to the PS. We found that in such interactions between the PS and PL the best way is to use HP ports to read data from the PS (memories) and transfer them to the PL and to write data from the PL to the PS (memories). Since memory controllers belong to the PS we can talk about data transfers between the PS and PL. Exchange of data in both directions is done in burst mode supported by a burst reader and a burst writer described in [8, 15]. Both processing (T_h) and communication (T_c) times are measured and taken into account.

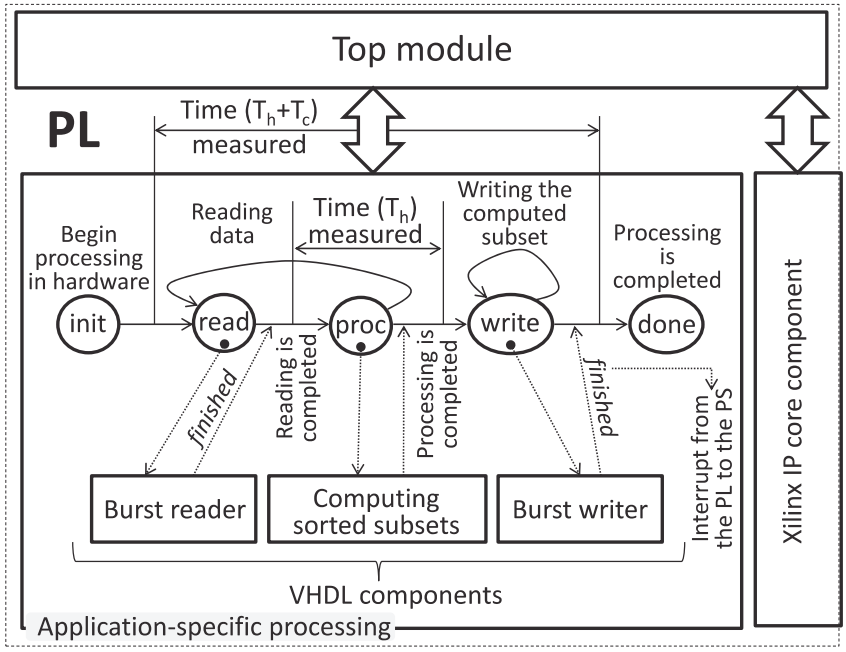

Figure 10: Operations in hardware components

6 Computing Large Subsets and Additional Capabilities

For some practical applications the maximum and/or minimum subsets may be large and the available hardware resources become insufficient to implement the circuits in Fig. 3.

The arising problem can be solved using the following technique. Let l_{max} and l_{min} be constraints for the upper and bottom parts of the sorting network in Fig. 3, i.e. circuits with larger values (than l_{max} and l_{min}) cannot be implemented due to the lack of hardware resources or for 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 incrementally [8] 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 (see Fig. 6) in such a way that only items that are less than B_u and greater than B_l are allowed to be processed, i.e. computing sorted subsets can be done only for the filtered data items. Thus, the second part of the maximum and minimum subsets will be computed and appended (in the PS) to the previously computed subsets (such as subsets from point 1). Note, that the method for processing incomplete subsets (see Fig. 4) may need to be applied for the last iteration.
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 [8]. This situation can easily be recognized. Indeed, if after corrections in point 1 above any new subset becomes empty then an infinite loop will be created. In such case we can use another method based on software/hardware sorters from [9]. In section 8 we will present the results of experiments for such sorters.

For some practical cases only the maximum or the minimum subsets need to be extracted. This task can be solved easier than in Fig. 3 with the aid of the circuit shown in Fig. 11 (for computing only the maximum subset).

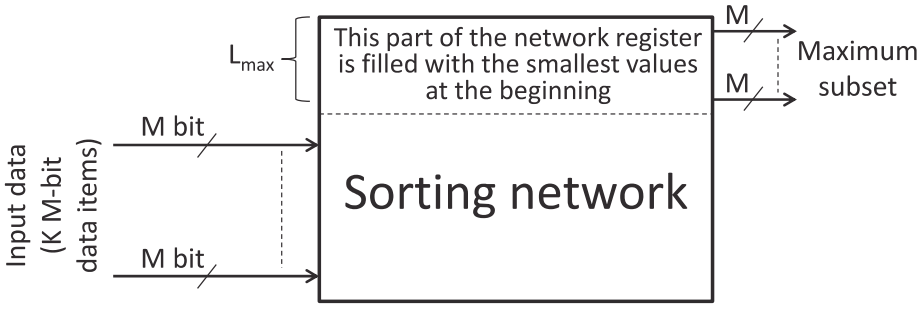

Figure 11: Computing the maximum subset for a given set

At initialization stage L_{max} M-bit words of the FR (see Fig. 5) are filled in with the smallest

possible value (such as zero or the minimal value for M-bit data items). After that the processing is executed as before (see section 3) and finally the maximum subset will be computed. For computing the minimum subsets the bottom part of Fig. 3 is filled in with the largest possible value (such as the maximum value for M-bit data items).

7 Practical Applications

Let us consider practical applications from the scope of control. We have already mentioned in section 1 that applying the technique [2] in real-time systems requires knowledge acquisition from controlled devices. The data may be compared with the previously collected data that are kept in databases for similar control scenarios. The results of comparison can be analyzed and used to modify the algorithms allowing control operations to be optimized, undesirable (or error prone) situations to be avoided, etc. Let us look at Fig. 12 where software collects important data from a controlled system, such as changes in temperature, deviation of positions, offsets, etc. The collected data are optionally filtered and their subsets (maximum, minimum, or both) are computed (see the bottom part of Fig. 12). Data from previous scenarios for analogous conditions are extracted from the database and they are also optionally filtered and similar subsets are computed (see the upper part of Fig. 12).

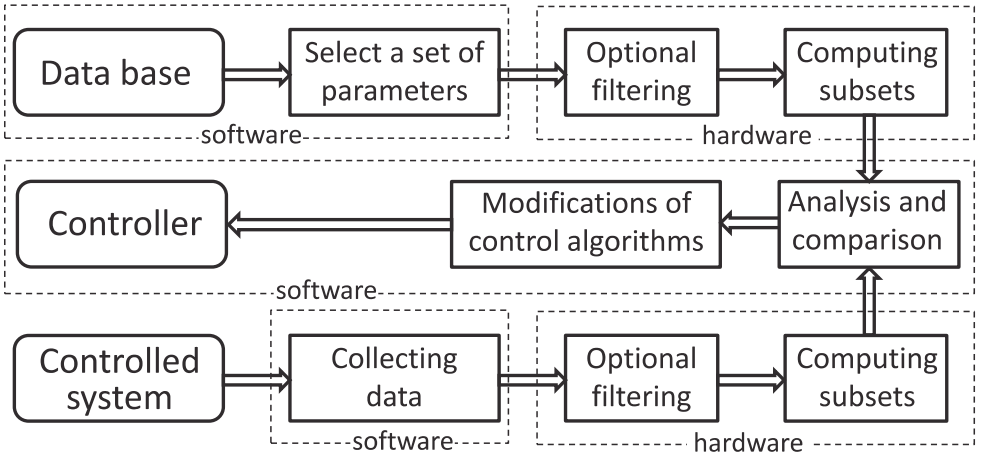

Figure 12: An example of control application

Data from the controlled system (see the bottom part of Fig. 12) and from the database (see the upper part of Fig. 12) are analyzed. For example, average maximum values are checked. The results of analysis may be used to modify control algorithms much like it is done in [9, 16]. For example, modules of controllers from [9] can be replaced to optimize execution of relevant operations.

Another group of potential applications is from the scope of statistical data manipulation such as data mining. To describe one of the problems from this area informally let us consider an example [6] 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 [6]. 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 [6]. Computing sets of the most frequent or the less frequent items in large data sets permits the relevant data mining algorithms to be simplified and accelerated. Sorting of subsets is involved in many known algorithms from this area e.g. [17–19] and the results of the paper may provide a valuable assistance.

8 Implementation, Experiments, and Comparisons

Much like [8] we have used a multi-level computing system [9]. Initial data are either generated randomly in software of the PS with the aid of C language rand or prepared in the host PC. In the last case data may be generated by some functions or copied from available benchmarks. Computing subsets in software/hardware systems is mainly done in Zynq APSoC xc7z020 housed on ZedBoard [20] with the aid of the described above software/hardware architectures (see Fig. 2-4, 6, 9, 10). Computing subsets in software only sorters is completely done in software of the PS calling C language qsort function which sorts data and after that the maximum and/or the minimum subsets are extracted. The results are verified in software running either in the PS or in the host PC. Functions for verification of the results are given in [9]. Verification time is not taken into account in the measurements below.

Synthesis and implementation of hardware modules were done in Xilinx Vivado 2015.2. Standalone software applications were created in C language and uploaded to the PS memory from the Xilinx Software Development Kit (SDK) using methods described in [9]. Interactions with APSoC are done through the SDK console window.

For all the experiments 64-bit AXI ACP port was used for transferring blocks between the PL and memories. The size of each block for burst mode is chosen to be 128 of 64-bit items. Two memories were tested: the OCM (for smaller number of data items) and external (on-board) DDR. The OCM is faster because it provides for 64-bit data transfers [7] but the size of this memory is limited to 256 KB. The available on ZedBoard 4 Gb DDR supports 32-bit data transfers.

The measurements were based on time units (returned by the function XTime_GetTime [21]) for $L_{max} = L_{min} = 128$, M=32, and K = 256 (see Fig. 3). The following operations have been executed: a) copying data to the selected memory in the PS; b) providing the necessary initialization for the function XTime_GetTime (i.e. the consumed time will be measured from this point); c) making the request, i.e. setting (through GPP) source address, destination address, the size of data to be copied, and start processing in the PL (optionally some other data, such as B_u and B_l for filtering, may be provided); d) copying data from the PS to PL and executing all the required operations in the PL; e) copying the computed subsets from the PL to PS; f) generating a hardware interrupt that is handled in the PS as a completion of the request (thus, the consumed time is measured at this point in the PS). Each unit returned by the function XTime_GetTime corresponds to 2 clock cycles of the PS [21]. 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 the software/hardware system comparing to the software only system. Note that Fig. 13, 14 give diagrams for the OCM. If DDR memory is used then communication overheads are slightly increased but acceleration in the software/hardware system comparing to the software only system is again significant.

Let us compare the results with [5, 8]. The number of data items in the proposed solutions is larger than in [5] and can easily be additionally increased. For similar data sets the achieved acceleration is better than in [8] thanks to additional optimization of the proposed circuits.

We also implemented and tested the proposed circuits in a more advanced prototyping board ZC706 [22] with Zynq microchip xc7z045. Data were taken from DDR memory and the maximum and minimum subsets were extracted with K data items where K varied from 256 to 1,024 (as before M = 32, N is equal to 256 KB). The consumed time varies from 1,850 μs for L_{min} = L_{max} = 256 to 7,200 μs for L_{min} = L_{max} = 1,024. Thus, the proposed solutions can be used for solving significantly more complicated problems that cannot be solved, in particular, with the aid of the methods [5]. If only the maximum or only the minimum subsets have to be computed the acceleration is slightly increased (although it is almost the same) and the occupied hardware resources are reduced.

The proposed filtering (see Fig. 6) does not consume any additional time because it is combined with data transfers. So, we can say that the time is included in communication overheads and the latter were taken into account in all measurements. It should be noted that filtering is not described in [5, 8].

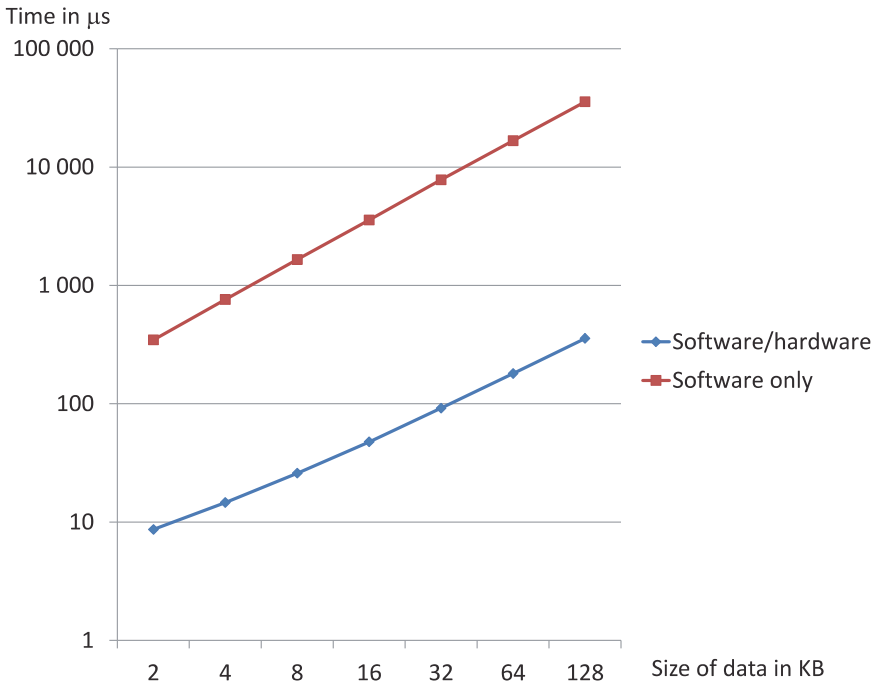

Figure 13: Computing time in software only and software/hardware systems

If the size of the requested subsets is increased in such a way that all data need to be read from memory several times then the results are the same as in [8] (see comments in [8] for additional details).

We found that parallel circuits that enable the maximum and minimum subsets to be ex-

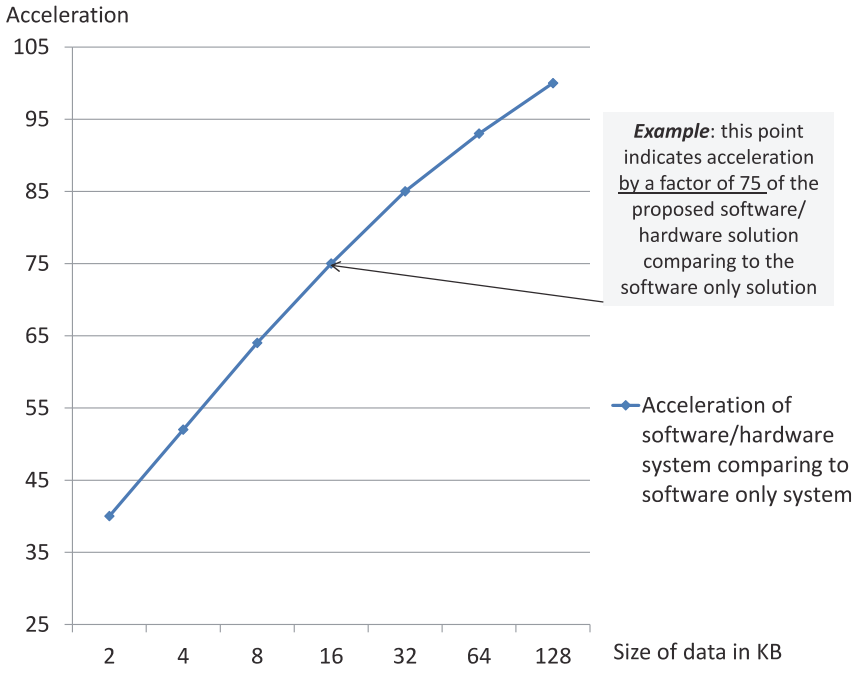

Figure 14: Acceleration of software/hardware system comparing to software only system

tracted in the ZedBoard [20] can be built up to K = 256. In this case additional hardware resources that enable data exchange between the PS (memories) and PL are available. Similar circuits for the ZC706 can be built up to K = 1,024. Note that if a block of data needs to be sorted in hardware then the number of processed data may be greater because in our case two blocks (each of which possesses K items) have to be handled in parallel and in case of data sorting [9] it is sufficient to handle just one block of data. Additional optimizations such as partial merging in hardware circuits permit the size K to be additionally increased. However, the processing time will also be increased.

9 Conclusion

The paper suggests methods for computing the maximum and minimum subsets that are extracted from large data sets in communicating software/hardware systems, namely in devices from the Xilinx Zynq family, which combine a high-performance processing system with advanced programmable logic. The extracted subsets may be filtered and this feature is useful for control applications. The proposed solutions are highly parallel permitting capabilities of programmable logic to be used very efficiently. All the proposed methods were implemented in commercial microchips, tested, evaluated, and compared with alternatives. The results of experiments have shown significant speed-up of the proposed software/hardware systems comparing to software only systems and to competitive hardware/software implementations. In particular, the size of

subsets was increased and additional tasks important for control applications were discussed and solved. Practical applications of the proposed technique for control applications and statistical data manipulation were also given.

Acknowledgment

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.

Bibliography

[1] Sklyarov, V.; Skliarova, I. (2013); Digital Hamming Weight and Distance Analyzers for Binary Vectors and Matrices, *Int. Journal of Innovative Computing, Information and Control*, 9(12): 4825-4849.

[2] Zmaranda, D.; Silaghi, H.; Gabor, G.; Vancea, C. (2012); Issues on applying knowledge-based techniques in real-time control systems, *International Journal of Computers Communications & Control*, 8(1): 166-175.

[3] 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*, 74(10): 2251-2271.

[4] Zhang, W.; Thurow, K.; Stoll, R. (2014); A knowledge-based telemonitoring platform for application in remote healthcare, *International Journal of Computers Communications & Control*, 9(5): 644-654.

[5] Farmahini-Farahani, A.; Duwe, H. J.; Schulte, M. J.; Compton, K. (2013); Modular design of high-throughput, low-latency sorting units, *Computers, IEEE Transactions on*, 62(7): 1389-1402.

[6] Baker, Z. K.; Prasanna, V. K. (2006); An architecture for efficient hardware data mining using reconfigurable computing systems. In Field-Programmable Custom Computing Machines, *Proc. 14th Annual IEEE Symp. on Field-Programmable Custom Computing Machines - FCCM*, Napa, USA, 67-75.

[7] Xilinx, Inc.; Zynq-7000 All Programmable SoC Technical Reference Manual, available at http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[8] Sklyarov, V.; Skliarova, I.; Rjabov, A.; Sudnitson, A. (2015); Zynq-based System for Extracting Sorted Subsets from Large Data Sets, *Journal of Microelectronics, Electronic Components and Materials*, 45(2): 142-152.

[9] Sklyarov, V.; Skliarova, I.; Silva, J.; Rjabov, A.; Sudnitson, A.; Cardoso, C. (2014); *Hardware/software co-design for programmable systems-on-chip*, TUT Press.

[10] Mueller, R.; Teubner, J.; Alonso, G.; (2012); Sorting networks on FPGAs, *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(1), 1-23.

[11] Sklyarov, V.; Skliarova, I. (2014); High-performance implementation of regular and easily scalable sorting networks on an FPGA, *Microprocessors and Microsystems*, 38(5): 470-484.

[12] Baddar, S. W. A. H.; Batcher, K. E. (2012); *Designing sorting networks: A new paradigm*, Springer Science & Business Media.

[13] Knuth, D. E. (2011); *The art of computer programming: sorting and searching (Vol. 3)*, Addison-Wesley.

[14] Kipfer, P.; & Westermann, R. (2005); Improved GPU sorting, *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*, edited by M. Pharr, 733-746, available at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.

[15] Silva, J.; Sklyarov, V.; Skliarova, I. (2015). Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip, *Embedded Systems Letters, IEEE*, 7(1): 31-34.

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

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

[18] Wu, X.; Kumar, V.; Quinlan, J. R. et al. (2008); Top 10 algorithms in data mining, *Knowledge and Information Systems*, 14(1): 1-37.

[19] Firdhous, M. (2012); Automating legal research through data mining, *Journal of Advanced Computer Science and Applications*, 1(6): 1-8.

[20] Avnet, Inc. (2014); *ZedBoard (ZynqTM Evaluation and Development) Hardware User's Guide, Version 2.2*, available at: http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.

[21] Xilinx, Inc.; *OS and Libraries Document Collection UG647*, available at: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/oslib_rm.pdf.

[22] Xilinx, Inc.; *ZC706, All Programmable SoC Evaluation Kit, UG961*, Available at: http://www.xilinx.com/support/documentation/boards_and_kits/zc706/2014_3/ug961-zc706-GSG.pdf.

PUBLICATION VIII

Rjabov, A.; Sklyarov, V.; Skliarova, I.; Sudnitson, A. (2017). RAM-based mergers for data sort and frequent item computation. Conforence on Information and Communication Technology, Electronics and Microelectronics (MIPRO2017), Opatija, Croatia, May 22-26, 2017, IEEE.

RAM-based mergers for data sort and frequent item computation

Artjom Rjabov*, Valery Sklyarov**, Iouliia Skliarova**, Alexander Sudnitson*
* Department of Computer Engineering, Tallinn University of Technology, Tallinn, Estonia
** Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, Aveiro, Portugal
artjom.rjabov@ttu.ee

Abstract - Data sorting and frequent item computation are important tasks in data processing. The paper suggests an architecture for parallel data sorting with simultaneous counting of every item frequency. The architecture is designed for streaming data and incorporates data sorting in hardware, merging of preliminary sorted blocks with compressing of repeated items with calculating of repetitions in hardware, and merging large subsets received from the hardware in general-purpose software. Hardware merge components of this architecture count and compress repeated items in sorted subsets in order to reduce merging time and prepare the data for frequent item computation. The results of experiments clearly demonstrate advantages of the proposed architectures.

Keywords—High-performance computing systems, Information processing; Sorting networks; Parallel sorting; Partial sorting; reconfigurable computing.

I. INTRODUCTION

Sorting is a procedure that is needed in numerous computing systems. Parallel algorithms for data sorting have been studied in computer science for decades. There are many different parallel sorting algorithms. The most notable of them are Parallel QuickSort, Parallel Radix Sort, Sample Sort, Histogram Sort [1] and a family of algorithmic methods known as sorting networks [2]. To better satisfy performance requirements, fast hardware accelerators have been researched in depth. The sorting networks presents a great interest for hardware acceleration because of their massive parallelism. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagate through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line. Sorting is a very resource expensive and time consuming operation. There are different approaches to overcome the resource limitation. Utilizing iterative networks with reusable comparators permits to process significantly larger data sets, but still to some extent. The combination of iterative network-based sorters with subsequent merging permits to process larger data sets than the sorting network allows. The merge operation can be implemented completely in software or partially in hardware for relatively small sorted data subsets. The merging can be implemented as tree-like structure of merge units. This approach allows us to reduce sorting time even further by detecting the repeated elements with subsequent recording of how many times the data was repeated and deleting the repeated entries in the sorted list. This approach allows us to reduce sorting time for data sets with repeated elements and prepare the data for frequent item computation.

Data sorting and frequent item computation is required in searching, statistical data manipulation and data mining (e.g. [3, 4]). To describe one of the problems from data mining informally let us consider an example [3] 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 [3]. 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 [3]. Fast extracting the most frequent or the less frequent items from large sets permits data mining algorithms to be simplified and accelerated.

The paper suggests a method and high-performance hardware implementation of data sorting algorithm based on parallel sorting network with subsequent merge. The functionality of the merge units is expanded by adding the operation of compressing the data by counting of the repeated data. The system is designed for working over streaming data. It utilizes Advanced eXtensible Interface (AXI) interfaces and is suggested as a PCI express (PCIe) peripheral.

The remainder of the paper contains 6 sections. Section II analyzes the related work. Section III describes highly parallel networks for sorting and explores hardware co-design. Section IV presents proposed system for data merge and item counting. Section V describes experimental setup and hardware accelerator architecture. Section VI presents the results of experiments and comparisons. The conclusion is given in Section VII.

II. RELATED WORK

Different approaches of hardware sorting units were studied by Marcelino et al. in [5]. They implemented a hardware/software hybrid sorter with a sorting unit based on insertion sorting algorithm and unbalanced merging

unit. They also utilized Bathcer's Even-Odd sorting network for software implementation and experimented with different combinations of software (QuickSort, Even-Odd network) and hardware (Insertion sorting, unbalanced merge). They also discussed possibilities of using pipelined sorting networks and balanced merging units. Chen and Prasanna in [6] proposed a hardware/software hybrid solution for accelerating database operations using Field-Programmable Gate Array (FPGA) and Central Processing Unit (CPU). Their sorting algorithm is based on merge-sort algorithm where first few sorting stages are implemented in FPGA as folded bitonic sorting networks. The rest of the algorithm is implemented in CPU.

Hardware acceleration of frequent item computation was explored by Teubner et al. in [7]. They suggested to use FPGAs and proposed three different methods. The first method is a straightfoward implementation of Space-Saving algorithm with min-heap data structure in Block Random-Access Memory (BRAM) for data storage. For the second method instead of BRAMs with min-heap structure they used two search trees implemented in lookup tables in order to get rid of min-heap sorting. The pipelined circuit of their third solution choses the best results in terms of performance and scalability. They achieved throughput four times higher than the best published result.

In our previous works we also explored different approaches of hardware/software systems for high-performance data processing and sorting [8-12]. In [9] we proposed a sorting-network-based hardware sorters with subsequent merge in software as well as different approaches of partial sorting for minimal and maximal subsets extraction which is used in frequent item computation. The latter problem also was explored in [10, 11]. In [12] we proposed a multi-level architecture for minimal/maximal subset extraction which utilizes a general purpose processor of a host PC and the programmable logic and processing system of Zynq device.

III. Pipelined Iterative Periodic Sorting Network

As a basis for our sorting circuit we use a periodic pipelined Odd-Even Transition Sorting Network (also known as Odd-Even Transposition Sorting Network or OETS). A periodic network is a type of network which

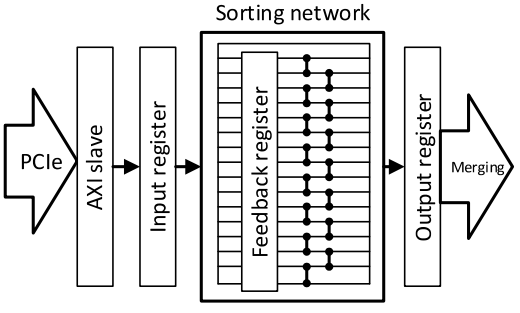

Fig. 1. The circuit for sorting data blocks

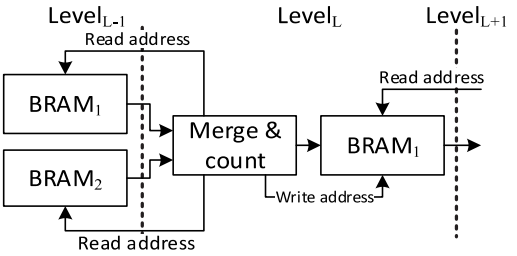

Fig. 2. Interaction between levels of merge operation

consists of identical sequences of comparators. Traditional implementation of OETS is less efficient than Batcher's networks, but it is more reliable and its implementation is simpler. Salloum and Wang proved that OETS has good fault-tolerant properties [13].

Like in our previous works, we use pipelined approach with reusable comparators presented in [14]. K M-bit data items that have to be sorted are loaded (from block RAM) to the feedback register. 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 [14] according to which if there are no swaps of data on the right-most line of comparators then sorting is completed. Note that the network [14] possesses significantly smaller combinational delays than networks from [2]. Besides, in the proposed architecture iterations are done at the same time as subsequent data are being received from the inputs. Such parallelism enables delays to be optimally adjusted allowing the total performance to be improved.

The sorter used in the proposed architecture is depicted in Fig. 1. It is based on the iterative sorting network described above and designed as an AXI bus peripheral to receive data from PCI express bus. The AXI slave control unit writes the data to the input register and initiates the sorting operation when the register is full. At the same time it starts writing the next data subset to the input register while the sorting network performs data sorting. After the completion of the sorting the system moves the data to the output register and the merging system copies it into its embedded RAM blocks. All three operations (receiving the data, sorting, copying from the output register) work in parallel in order to achieve maximal possible performance.

IV. Pipelined Merging and Item Counting

The merging part of the circuit is based on embedded block-RAM. Xilinx Virtex 7 FPGA provide RAM blocks with 36kbit of memory [15], where the data word size can be adjusted for the needs of the system. Every word in RAM blocks of our system contain a pair of data item and its count. The sizes of both the item and the count are also adjustable and should be chosen considering the nature of the input data.

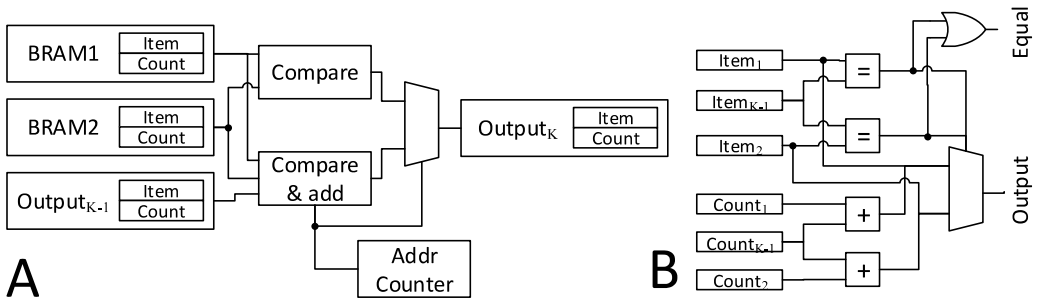

Fig. 3. "Merge and count" architecture: A) General architecture of the merger B) Compare and add operation

Fig. 2. depicts the elements of the merging system and interactions between them. The "merge and count" block of level L receives the input data from two embedded block-RAMs of the previous level(L-1). The circuit compares all the items in two sorted subsets of N data items and merges them into one sorted subset. The maximal size of the final data set is 2×N items. This worst case scenario can occur if no repeated items were found in both input subsets. Every merging element of the system contain two address counters for selecting the data from block RAMs of the previous level and one counter for writing the merged and counted data to the output RAM block.

Although the maximal number of clock cycles for merging N-item blocks is 2×N, our system with compression and item counting require less clock cycles for the data sets with repeated items. Every subsequent level of merging require less clock cycles than the previous one, because the compression and counting was partially done in the previous level.

Merge fragment from Fig. 2. is depicted on Fig. 3(a). The compression and the counting of the items is done in "compare and add" block shown in Fig. 3(b). The system stores the data item which was written after the previous comparison and compares it with both inputs. If the item part of the item/count pair previously written to the RAM block is not equal to both of them, then the merger writes the item/count pair with larger item value to the output RAM block and increments both write address counter and read address counter for the input with the largest value. Otherwise, the merger does not increment the write address of the block and writes the new count number to the count part of the item/count pair. The new count number is the sum of the count parts of the previously written data item and the count of one of the inputs, which has an item part equal to the previously written one. During the first level of merging every pair have '1' as its count value. All zeros in the count part mean that the total number of repetitions exceeded the capabilities of the RAM block.

The RAM blocks of every item of the merging system are capable of storing all data from the inputs, but if the sorted set supplied to the merger contains repeated items, the system does not fill the RAM blocks completely. The merger reads the value from the write address register of the mergers from the previous level. It informs the merger about how many item pairs were actually written during the previous merge operation.

The proposed architecture is designed to be constructed with any number of merging/counting layers and the size of the initial sorter, but the final implementation always depends on the target device.

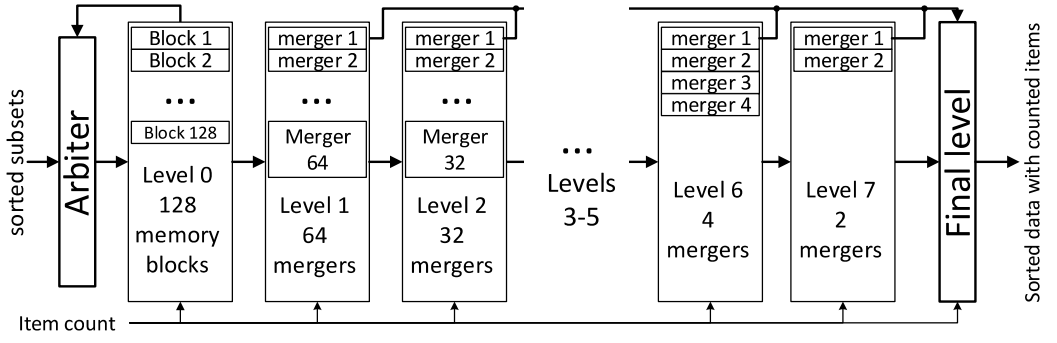

Fig. 4. Merging system

Fig. 4. depicts the merging system implemented for Virtex-7 FPGA device with 8 level of merging/counting. It has one initial level of RAM blocks which contain data subsets sorted with the sorting network described in section III. The sorter in this implementation receives 32-bit items grouped by 4 and writes sorted subsets of 512 elements into the initial level of RAM blocks. The writing to the first RAM blocks of the merging system is also implemented in groups of 4 items in order to keep up with the bus speed. The merging operation is not performed during this step.

The first element of the first level of the merge operation activates when the first two sorted subsets become available on the initial BRAM level. Every merging and counting element of every level starts its operation immediately when two mergers/counters connected to it from the previous level finish merging and counting.

Although the number of merging/counting levels and therefore the size of the final data set is limited by the resource availability of the target device, the architecture was designed to process streaming any volumes of data. The merging levels can be disabled if not all of them are required for the data processing. It can be done by supplying item count value (see Fig. 4). The final level is responsible for the preparation of the data from the last enabled level for the bus transaction and merges the subsets from the previous level if all levels are enabled. The merging/counting system also can merge several data sets in a pipeline for subsequent processing in GPC. When the merge level finishes its operations, the previous level becomes available to process new item pairs since the next level has acquired all necessary data.

V. Hardware Architecture and Experimental Setup

The system was designed as a hardware accelerator for a host PC which communicates through PCI-express interface in Direct Memory Access (DMA) mode. Fig. 5. depicts this architecture.

Software in the host PC runs the 32-bit Linux operating system (kernel 3.16) and executes programs (written in C language) that take results from PCI-express (from the FPGA) for further processing. We assume that the data collected in the FPGA are preprocessed in the programmable logic by applying various highly parallel networks (see Section III), and the results are transferred to the host PC through the PCI-express bus. To support data exchange through PCI-express, a dedicated driver was developed. The programmable logic uses the Intellectual Property (IP) core of the central direct memory access (Xilinx CDMA) [16] module to copy data through AXI PCI express bridge (Xilinx AXI-PCIE) [17]. Data transfer in the host PC is organized through direct memory access (DMA). To work with different devices, a driver (kernel module) was developed. The driver creates in the directory /dev a character device file that can be accessed through read and write functions, for example write(file, data array, data size). The PC BIOS assigns a number (an address) to the selected base address register (BAR) and a corresponding interrupt number that will be later used to indicate the completion of a data transfer. As soon as the driver is loaded, a special operation (probe) is activated and the availability of the device with the given identification number (ID) is verified (the ID is chosen during the customization of the AXI-PCIE). Then a sequence of additional steps is performed (see [18 pp. 302-326] for necessary details). A number of file operations are executed in addition to the probe function. In our particular case, access to the file is done through read/write operations.

VI. Experimental Results and Comparison

The system for data transfers between a host PC and an FPGA has been designed, implemented, and tested. Experiments were done in the VC707 prototyping board [19] that contains Virtex-7 XC7VX485T FPGA from the Xilinx 7th series with PCI express endpoint connectivity "Gen1 8-lane (x8)". All circuits were synthesized from the specification in VHDL and implemented in the Xilinx Vivado 2016.2 design suite. Software programs in the host PC run under Linux operating system and they were developed in C language. Data were transferred from the host PC to the VC707 and back through PCI express. The host PC is based on Intel core i7 3820 3.60 GHz.

For the experiments two sorting and merging systems were built. The first system is capable of sorting 32-bit data items and does not perform sorting compression and data counting. The second system is based on the first system, but includes compressing and merging functionality. The second system is adjustable for different data item sizes. Both systems are capable of sorting 2^{16} of 32-bit data items (256KB) maximum and require identical number of RAM blocks.

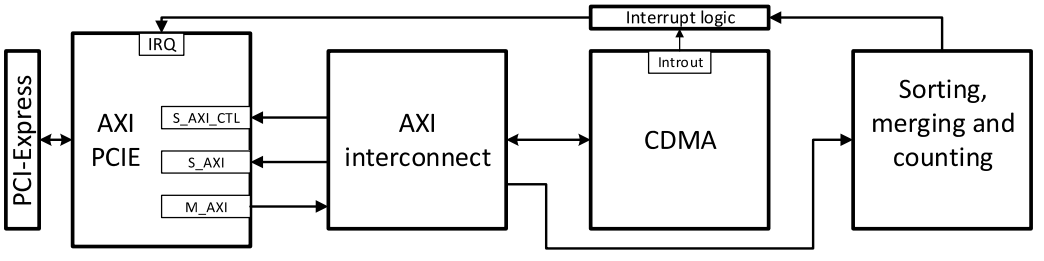

Fig. 5. Basic hardware architecture.

We conducted the experiments for sorting and merging without any item counting only for of 2^{16} 32-bit, since the merge operation is the most time consuming part of the system and it depends only on the number of the inputs and not the size of the item. Merging with item counting was performed for 2^{16} of 32-, 16- and 8-bit items. The 36-bit size of the word for 32-bit items in the BRAM was chosen. It means that the item count part of the word is 4-bit and capable of counting up to 15 repetitions, which is enough for experiments with randomly generated data. The system was configured to work with 32-bit words with 16-bit size of both the item and the count parts for counting and merging of 16- and 8-bit data.

The experiments were conducted with randomly generated numbers. The merging with counting 32-bit items didn't show any noticeable speedup over simple merging, since 2^{16} of randomly generated numbers do not have significant number of repetitions. The merging with counting of the same number of 16-bit data items is 1,45 times faster than the simple merge and merge of 8-bit items is 27,28 times faster.

We experimented with different volumes of 8-, 16- and 32-bit data items and compared them with software sorting. The host PC was used for merging the data sets larger than volume of data that can be processed with the FPGA. In addition to data sorting and merging, PCI express throughput and operating system overhead were also taken into account.

Fig. 6. shows the experimental results of sorting different volumes of randomly generated 32-bit data items in the proposed system compared to the software sorting implemented with C language `qsort` function. The results clearly demonstrate that the proposed solution is faster. Our experiments did not show any noticeable difference in sorting sets with different size of data items in software with `qsort` and therefore the results only for 32-bit data items are presented.

Fig. 7. depicts comparison of sorting data sets in the proposed system with 8-, 16-, and 32-bit item sizes.

The experiments show that the sorting throughput for the proposed systems is significantly better than in the host PC. Also it is clearly seen that merge operation with compression of repeated items is faster for data sets with high item repetition. Also the proposed system performs not only data sorting, but provides the number of repetitions of every data item in the set along with completely sorted data. This information can be used for extracting the most frequently encountered items and other subsequent data processing. The subset extraction of the most frequent numbers can be done in hardware by inserting after the last level of merging partial sorters for subset extraction presented in [10].

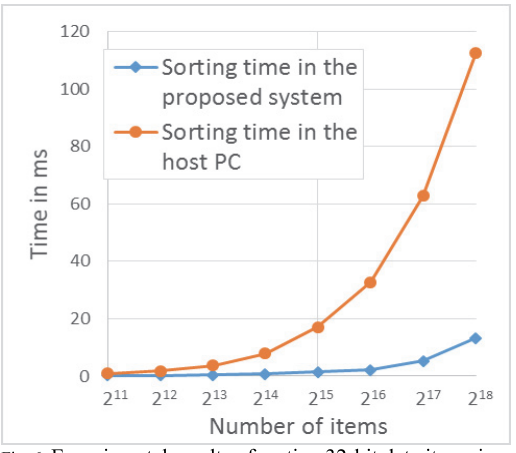

Fig. 6. Experimental results of sorting 32-bit data items in the proposed system and in the host PC.

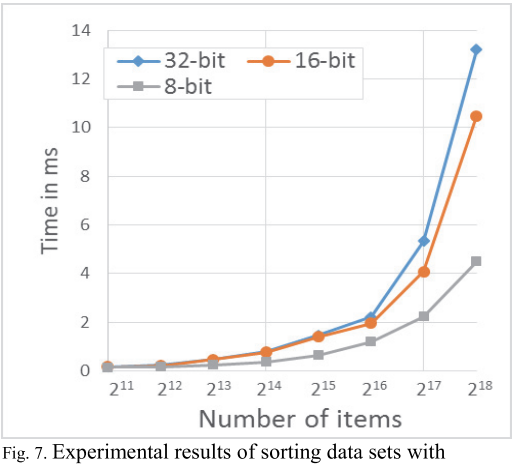

Fig. 7. Experimental results of sorting data sets with different item sizes.

VII. Conlusion

The paper suggests hardware-based methods of data sorting with simultaneous counting the repetition of all data items. The merging system performs counting and compression of the sorted subsets, which speeds up sorting of data sets with large number of repeated items and provides the number of repetitions for all items along with completely sorted data set. The proposed solutions are highly parallel permitting capabilities of programmable logic to be used very efficiently. All the proposed methods were implemented in commercial microchips, tested, evaluated, and compared. The results of experiments have shown significant advantages of the proposed architecture.

Acknowledgment

This research was supported by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, the Study IT in Estonia Programme, and Estonian Association of Information Technology and Telecommunications.

REFERENCES

[1] E.V. Kalé, E. Solomonik, "Sorting," in Encyclopedia of Parallel Computing, Springer Science+Business Media, 2011, pp. 1855-1862.

[2] S.W. Aj-Haj Baddar, K.E. Batcher, Designing Sorting Networks. A New Paradigm., Springer, 2011.

[3] Z.K. Baker, V.K. Prasanna, "An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems," in Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, USA, 2006.

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

[5] Marcelino, R.; Neto, H.C.; Cardoso, J.M.P., "A comparison of three representative hardware sorting units," in *35th Annual Conference of Industrial Electronics*, 2009.

[6] Ren Chen, Viktor Prasanna, "Accelerating Equi-Join on a CPU-FPGA," Ming Hsieh Department of Electrical Engineering – Systems, University of Southern California, Los Angeles, California, 2016.

[7] J. Teubner, R. Muller, and G. Alonso, "Frequent item computation on a chip," IEEE Trans. Knowl. Data Eng, vol. 238, pp. 1169-1181, 2011.

[8] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson , (2013), Implementation of Parallel Operations over Streams in Extensible Processing Platforms. Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on (pp. 852-855).

[9] Sklyarov, V.; Rjabov, A.; Skliarova, I.; Sudnitson, A. (2016). High-performance Information Processing in Distributed Computing Systems. International Journal of Innovative Computing, Information and Control, 12 (1), 139–160.

[10] V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, (2015). Zynq-based System for Extracting Sorted Subsets from Large Data Sets. Informacije MIDEM, 45(2), 142-152.

[11] V. Sklyarov, A. Rjabov, I. Skliarova, A. Sudnitson, (2016). High-performance Information Processing in Distributed Computing Systems. International Journal of Innovative Computing, Information and Control, 12 (1), 139–160.

[12] A. Rjabov, V. Sklyarov, I. Skliarova, A. Sudnitson, (2015). Processing Sorted Subsets in a Multi-level Reconfigurable Computing System. Elektronika ir Elektrotechnika, 21(2), 30-33.

[13] S.N. Salloum, D.H. Wang,, "Fault tolerance analysis of odd-even transposition sorting networks with single pass and multiple passes," in IEEE Pacific Rim Conference on Communications, Computers and signal Processing, 2003.

[14] V. Sklyarov, I. Skliarova,. (2014); High-performance implementation of regular and easily scalable sorting networks on an FPGA, Microprocessors and Microsystems, 38(5): 470-484.

[15] Xilinx, Inc., 7 Series FPGAs Memory Resourcesv1.12, https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf, 2016

[16] Xilinx, Inc., AXI Central Direct Memory Access v4.1, http://www.xilinx.com/support/documentation/ipdocumentation/axicdma/v41/pg034-axi-cdma.pdf, 2015.

[17] Xilinx, Inc., LogiCORE IP AXI Bridge for PCI Express v1.06, http://www.xilinx.com/support/documentation/ipdocumentation/axipcie/v25/pg055-axi-bridgepcie.pdf, 2012.

[18] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, http://lwn.net/Kernel

[19] Xilinx, Inc., VC707 evaluation board for the Virtex-7 FPGA user guide. http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf, 2016 (accessed 08.06.2016).

PUBLICATION IX

Sklyarov, V.; Skliarova, I.; **Rjabov, A**.; Sudnitson, A. (2017). Fast Iterative Circuits and RAM-based Mergers to Accelerate Data Sort in Software/Hardware Systems, Proceedings of the Estonian Academy of Sciences, 66 (3), 323-335.

Proceedings of the Estonian Academy of Sciences,
2017, **66**, 3, 323–335
https://doi.org/10.3176/proc.2017.3.07
Available online at www.eap.ee/proceedings

COMPUTER
SCIENCE

Fast iterative circuits and RAM-based mergers to accelerate data sort in software/hardware systems

Valery Sklyarov[a], Iouliia Skliarova[a], Artjom Rjabov[b]*, and Alexander Sudnitson[b]

[a] Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal; {skl, iouliia}@pb.ua.pt
[b] Department of Computer Systems, School of Information Technologies, Tallinn University of Technology, Akadeemia tee 15A, 12618 Tallinn, Estonia; aleksander.sudnitson@ttu.ee

Received 3 February 2017, accepted 14 March 2017, available online 6 July 2017

Abstract. The paper suggests and describes two architectures for parallel data sort. The first architecture is applicable to large data sets and it combines three stages of data processing: data sorting in hardware (in a Field-Programmable Gate Arrays – FPGA), merging preliminary sorted blocks in hardware (in the FPGA), and merging large subsets received from the FPGA in general-purpose software. Data exchange between the FPGA and a general-purpose computer is organized through a fast Peripheral Component Interconnect (PCI) express bus. The second architecture is applicable to small data sets and it enables sorting to be done at the time of data acquisition, i.e. as soon as the last data item is received, the sorted items can be transferred immediately. The results of experiments clearly demonstrate the advantages of the proposed architectures that permit the reduction of the required hardware resources and increasing throughput compared to the results reported in publications and software functions targeted to data sorting.

Key words: parallel data processing, merging, iterative networks, communication-time processing, Field-Programmable Gate Array (FPGA), Peripheral Component Interconnect (PCI) express bus.

1. INTRODUCTION

Sorting is a procedure that is needed in numerous computing systems [1,2]. For many practical applications, sorting throughput is very important. To better satisfy performance requirements, fast accelerators based on Field-Programmable Gate Arrays (FPGAs) (e.g. [3–11]), Central Processing Units (CPUs) (e.g. [7,12–16]), and multi-core CPUs (e.g. [17,18]) have been researched in depth. Two of the most frequently explored parallel sorters are based on sorting [1–3,19] and linear [4] networks. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagate through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line. Three types of such networks have been studied: pure combinational (e.g. [3,9]), pipelined (e.g. [2,3,9]), and combined (partially combinational and partially sequential) [2,5,20]. The linear networks, which are often referred to as linear sorters [4], take a sorted list and insert new incoming items in the proper positions. The method is the same as the insertion sort [1] that compares a new item with all items in parallel, then inserts the

* Corresponding author, artjom.rjabov@ttu.ee

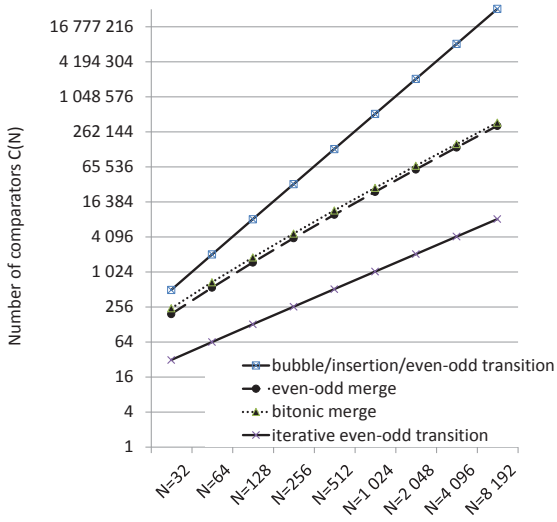

Fig. 1. The number of comparators for different values N of data items.

new item at the appropriate position, and shifts the existing elements in the entire multi-item vector. The main problem with this method is that it is applicable only to small data sets (see, for example, the designs discussed in [4], which accommodate only tens of items).

The majority of sorting networks implemented in hardware use Batcher even-odd and bitonic mergers [21]. Other types are rarer (see, for example, the comb sort [22] in [8], the bubble and insertion sort in [3,9], and the even-odd transition (transposition) sort in [12]). Research efforts are concentrated mainly on the following three directions: (1) networks with a minimal depth or number of comparators (e.g. [3,13]); (2) co-design, rationally splitting the problem between software and hardware (e.g. [3,9]), and (3) the regularity of the circuits and interconnections (e.g. [2,5]).

We target our results towards FPGAs because these devices are regarded more and more as a universal platform that enables computational algorithms to be significantly accelerated. The FPGAs still operate on a lower clock frequency than non-configurable Application-Specific Integrated Circuits (ASICs) and Application-Specific Standard Products (ASSPs) and broad parallelism is evidently required to compete with potential alternatives. Thus, sorting and linear networks can be seen as very adequate models. Unfortunately, they have many limitations. Suppose N data items, each of size M bits, need to be sorted. The results of [3,13] show that the most widely used sorting networks [19,21] cannot be built for $N > 128$ $(M = 32)$, even in a relatively advanced FPGA because the hardware resources are not sufficient. Iterative networks from [2] enable the number of comparators $C(N)$ to be notably decreased but even after that we cannot sort more than 4096 items in the most advanced FPGAs, such as that from the Virtex-7 family of Xilinx. When N is increased, the complexity of the networks (the number of comparators/swappers $C(N4)$) grows rapidly [1–3,9,19] (see Fig. 1).

It is easy to conclude from Fig. 1 that sorting networks can be implemented in an FPGA only for a small number N of items while practical applications require millions of such items to be processed. One possible way is to sort relatively small subsets of larger sets in an FPGA and then to merge the subsets in software of a higher-level system (see Fig. 2). The initial set of data that is to be sorted is divided into Z subsets of N items. Each subset is sorted in an FPGA using the referenced networks. Merging is executed as shown in Fig. 2, in a host system/processor that interacts with the FPGA. Each horizontal level of merging permits the size of

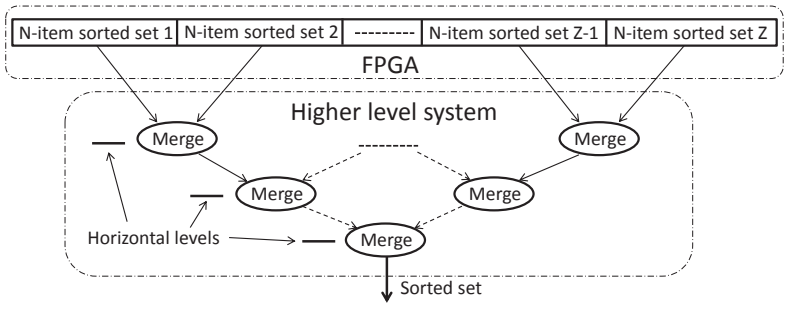

Fig. 2. The merging of sorted subsets in a software of a higher-level system.

blocks to be doubled. Thus, if $N = 2^{10} = 1024$ and $K = 2^{20} = 1\,048\,576$ items are to be sorted, then 10 levels of mergers are required (see Fig. 2). Clearly, the larger are the blocks sorted in FPGAs, the less merging is needed. Thus, we have to sort in hardware as many data items as possible with such throughput that is similar to the throughput of sorting networks. Besides, the networks [19,21] involve significant propagation delays through long combinational paths. Such delays are caused not only by comparators, but also by multiplexers that have to be inserted and by interconnections. Hence, clock signals with high frequency cannot be applied. Pipelining permits the clock frequency for circuits to be increased because delays between registers in a pipeline are reduced. A number of such solutions are described in [3,13]. However, once again, the complexity of the circuits becomes the main limitation. The analysis presented in [2] enables us to conclude the following: (1) the known even-odd merge and bitonic merge circuits [19,21] are the fastest and enable the best throughput to be achieved. However, they are very resource-consuming and can only be used effectively in existing FPGAs for sorting very small data sets; (2) pipelined solutions permit faster circuits than in point (1) to be designed. However, assuming that pipelining can be based on flip-flops in the used slices (so that additional slices are not required), resource consumption is at least the same as in point (1), therefore, in practice, only very small data sets can be sorted; (3) the existing even-odd merge and bitonic merge circuits are not very regular (compared to the even-odd transition network, for example) and, thus, the routing overhead may be significant in FPGAs.

There is also another problem that might arise. As a rule, initial data and final results are stored in conventional memories and each data item is kept at the relevant address of the memory. Suppose we would like to sort a set of data items. Let us look at Fig. 3 where the initial (unsorted) set is saved in the memory and the resulting (sorted) set is also saved in the memory. Parallel operations need to be applied to parallel subsets of data items, thus, in the beginning, initial data need to be unrolled (see Fig. 3) and the sorted items need to be stored in the memory one by one (see Fig. 3). Hence pre- and post-processing operations are involved and they (1) sequentially read unsorted data items and save them in a long-size input register and (2) copy the sorted data items from the long-size output register to conventional memories. These operations undoubtedly involve significant additional time. To reduce or even avoid such time, we have to be able to combine reading/writing data items and their sorting. We will call such type of data sorters communication-time data sorters.

This paper proposes a set of methods and device architectures with the following novel contributions:

1. The less resource-consuming iterative networks from [2] should be combined in hardware with pipelined Random Access Memory (RAM)-based data mergers, which permits
   (a) increasing the number of data items sorted in hardware significantly (more than one hundred times compared to [2]) without performance degradation,
   (b) performing data sorting in parallel with merging in hardware;

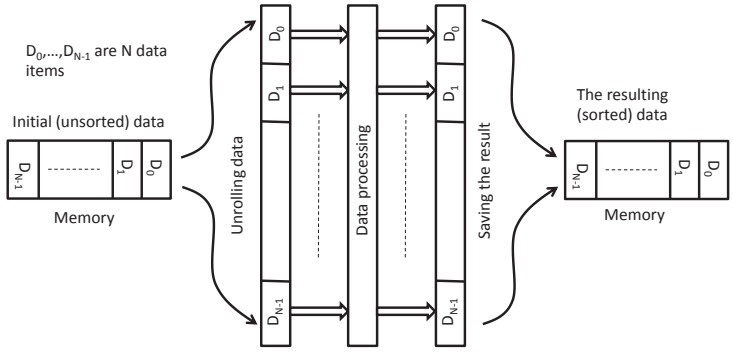

Fig. 3. Pre- and post-processing.

2. Communication-time data sorters that enable data acquisition and sorting to be executed in parallel in such a way that data sorting is completed as soon as the last data item has been received;
3. Three-level data sorters, two of which (network-based sorters and RAM-based mergers) are implemented in an FPGA and the last one – in a higher-level computing system that is in our case a general-purpose computer interacting with the FPGA through the Peripheral Component Interconnect (PCI) express bus.

2. SYSTEM ARCHITECTURE

Figure 4 depicts the considered system architecture. There are two basic subsystems that are a general-purpose computer (GPC) and an FPGA interacting through the PCI express bus. Let us assume that the FPGA can sort L blocks and each block contains up to N data items, i.e. such a number of items that can be sorted in the network [2]. The FPGA receives L blocks (containing up to N data items) from the GPC, sorts each block (see the rectangle A in Fig. 4), merges the sorted blocks (see the rectangle B in Fig. 4), and sends $L \times N$ sorted data items back to the GPC. The size M of each item is chosen to be 32 bits and it might be increased easily (FPGA circuits are easily scalable). Four 32-bit data items are packed in 128-bit words for data exchange through the PCI express bus.

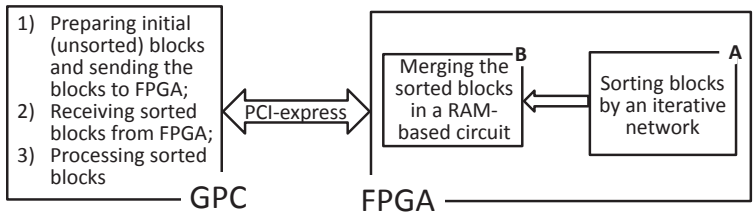

Fig. 4. General architecture of the considered system.

The FPGA implements circuits for the two levels referenced above, i.e. for an iterative sorter (see the rectangle A in Fig. 4) and for a merger (see the rectangle B in Fig. 4). In the beginning, we will use the network from [2] extended with some additional registers allowing data acquisition, sorting, and subsequent merging to be partially combined. Such an architecture implemented in the FPGA will be discussed in the following two subsections. The next section suggests some improvements of the network to design communication-time data sorters.

2.1. Iterative network for sorting data

Figure 5 depicts the used iterative network. The core of the network is the circuit proposed in [2]. There are also two additional registers R_i and R_o. The register R_i sequentially receives N data items from the GPC through the PCI express bus. It was explained above that such N items compose one block that can be entirely sorted in the network [2]. In practice, four items are packed and thus, parallel writing to the register R_i of four 32-bit items is actually done. As soon as the first block is received, all data items from this block are sorted in the iterative network from [2], and the maximum number of clock cycles is $N/2$ [2]. At the same time, data items from the next block are received from the GPC through the PCI express bus. As soon as data items from the first block are sorted, they are copied in parallel to the output register R_o. After that the second block is copied to the register R and sorted (see Fig. 5) and the third block is being received from the GPC through the PCI express bus. At the same time, the first sorted block is copied to the embedded block-RAM for subsequent merging. Hence, the first sorted block will be copied to RAM after the acquisition of two blocks from the PCI express bus. Then data acquisition from the GPC, data sorting, and copying data to the merger will be done in parallel. We can see from Fig. 5 that there are just two sequential levels of comparators/swappers in the iterative data sorter [2]. Thus, the delay is very small and we can apply high synchronization frequency. The results of [2] clearly demonstrate that such circuits are very efficient. Additional improvements are done to adjust the speed of data acquisition and sorting. Indeed, one block of N data items is received in $N/4$ clock cycles and the sorting time is up to $N/2$ clock cycles, i.e. it is almost two times longer.

Figure 6 demonstrates how to adjust the speed. There are now two iterative data sorters running in parallel. The first sorter processes data from the first half of the register R_i and the second sorter processes data from the second half of the register R_i. In the beginning, two blocks with $2 \times N$ items are copied to R_i and it involves $2 \times N/4 = N/2$ clock cycles. Then two blocks are sorted in parallel, which also involves up to $N/2$ clock cycles.

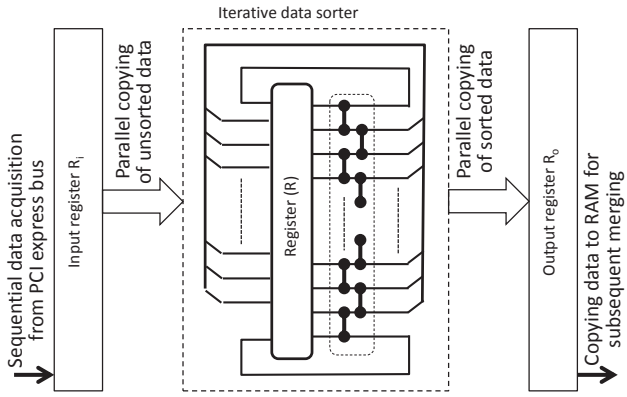

Fig. 5. The circuit for sorting blocks.

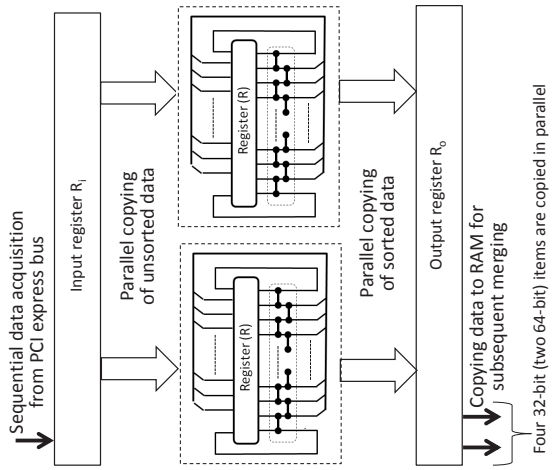

Fig. 6. Adjusting the number of clock cycles required in different blocks.

Finally, two sorted blocks are copied to two dual-port embedded block-RAMs. The respective write port is configured for data width 64. Thus, pairs of data items are copied in each clock cycle and it involves totally also $N/2$ clock cycles for both blocks. Therefore, everything is completely adjusted.

2.2. Pipelined merging

Merging is done on the basis of embedded block-RAM. Figure 7 shows one level of merging. Input data comes from two embedded block-RAMs, which is merged, and copied to a new embedded block-RAM. There are two address counters for each input RAM. In the beginning they are set to 0. Two data items are read and compared. If the item is selected from the first RAM, the address counter of the first RAM is incremented, otherwise the address counter of the second RAM is incremented. Two N-item blocks are merged in $2 \times N$ clock cycles. Different types of parallel merging have been verified and compared. We found that the best result (i.e. the fastest and the less resource-consuming) is produced in a simple RAM-based circuit depicted in Fig. 8.

There are G levels to merge L sorted blocks and $2^{G-1} < L \leq 2^G$. The first level is composed of L embedded block-RAMs. The second level is composed of $L/2$ embedded block-RAMs, and the last level is composed of one embedded block-RAM. The size of each RAM for the first level is N 32-bit words for reading and $N/2$ 64-bit words for writing. The size of each subsequent level is doubled. Initially, L embedded block-RAMs of the first level are filled in with sorted blocks. Then these blocks are merged at the second level. Afterwards the blocks of the second level are merged at the third level and at the same time the block-RAMs of the first level are being

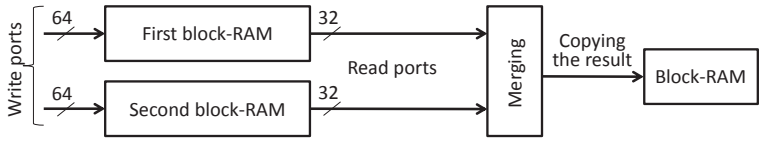

Fig. 7. Simple merging of two sorted blocks.

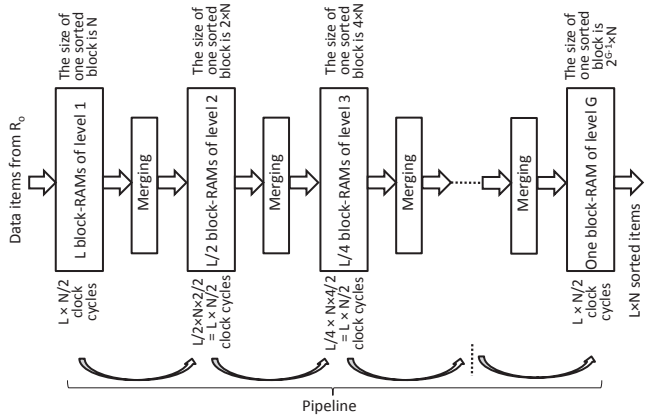

Fig. 8. Pipelined merging with embedded block-RAM.

filled in with a new subset of L sorted blocks. Thus, many subsets of L blocks will be processed in parallel and this is a special type of pipeline organized based on embedded block-RAMs (see Fig. 8).

The architecture in Fig. 8 permits many sets with L blocks (each block contains N M-bit data items) to be sorted in the pipeline in the way shown in Fig. 9. Equal numbers enclosed in circles indicate the steps executed in parallel. It was shown in the previous section (2.1) that the first time the level 1 block-RAM will be filled in with sorted data from the first block is after $3 \times N/2$ clock cycles. After that it is updated with the new block in $N/2$ clock cycles. So, an additional delay appears just from the beginning and it is avoided in the subsequent steps. As soon as data are copied to the first-level RAM, merging is started and the sorted data are copied from the first-level to the second-level RAM. This process involves $L \times N/2$ clock cycles. During this period of time the first-level RAM is used for merging and new data items cannot be copied to this RAM. In fact, it is possible to merge and to sort data at the same time. However, we found that such merger requires a complex arbitration which significantly increases hardware resources leading to reducing the size N of blocks. Finally, such more

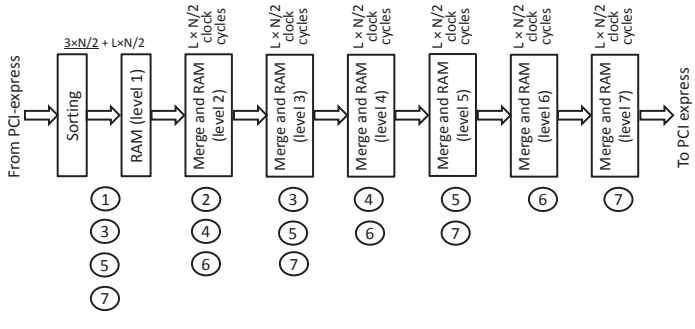

Fig. 9. Parallel operations in the proposed architecture.

complicated circuits do not give any advantage. This means that the resulting throughput cannot be increased. As soon as merging is completed, all data are copied to the second-level RAM and the first-level RAM may be refilled with new L sorted blocks.

Figure 9 explicitly indicates parallel operations. For example, merging at levels 3, 5, 7 is executed in parallel with data sorting. This method can be applied to the sorting of very large sets of data (tens and hundreds of millions of data items). In this case, the GPC (see Fig. 4) divides a very large set into subsets composed of $L \times N$ data items. The subsets are sorted in the pipelined structure shown in Fig. 9 and then merged in the software of the GPC. The experimental section below demonstrates that the data sorter implemented in Virtex-7 FPGA allows sorting data in hardware for $L = 128$ and $N = 512$. Thus, $512 \times 128 = 65\,536$ 32-bit data items (or 256 KB) are sorted and then 256 KB blocks can be merged in software. It will be shown in the experimental section that sorting in hardware (including data exchange with the GPC) is faster than similar sorting in software. Merging larger blocks permits the time of sorting in software to be considerably reduced.

3. COMMUNICATION-TIME DATA SORTERS

The actual performance of the designed circuits is often limited by the interfacing circuits that supply the initial data and return the results. Indeed, even for the most recent and most advanced on-chip interaction methods, such as those used in the Advanced eXtensible Interface (AXI), the communication overheads do not allow the theoretical throughput to be achieved in practical designs. The method and architecture described above permit only a small delay for data transmission in the beginning. When we sort large sets of data such delay is indeed negligible compared to the total delay. So, the proposed technique is very effective. In many practical cases we would like to sort small sets, such as those composed of N data items. For such a case the delay between the last received item and the final result of sorting becomes up to $N/2$ clock cycles and this may not be acceptable for many practical applications. We consider in this section such a method that enables the sorted results to be sequentially copied immediately after the last data item is received.

We describe below a parallel circuit that enables sorting to be entirely done within the time required for data transfers to and from the circuit; no additional time is required. Further, the design is very economical. The communication-time circuit, which is based on the network for discovering minimum and maximum values from [23], is shown in Fig. 10. It is composed of N M-bit registers R_0,..., R_{N-1}, and $N-1$ comparators/swappers.

At the initialization step, all the registers R_0,..., R_{N-1} are set to the minimum possible value for data items. For the sake of simplicity, this value is assumed to be 0. Any other value may also be chosen. Data items are received sequentially from interfacing circuits through the multiplexer Mux. The value x is set to 0, so all input

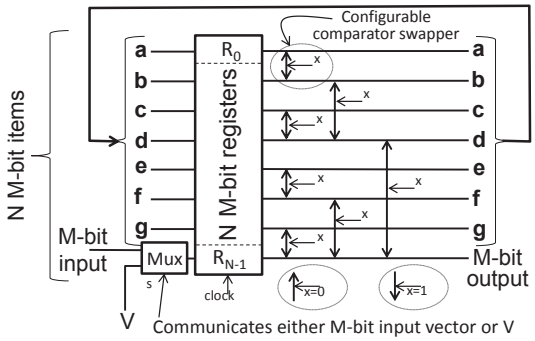

Fig. 10. Communication-time data accumulator/sorter.

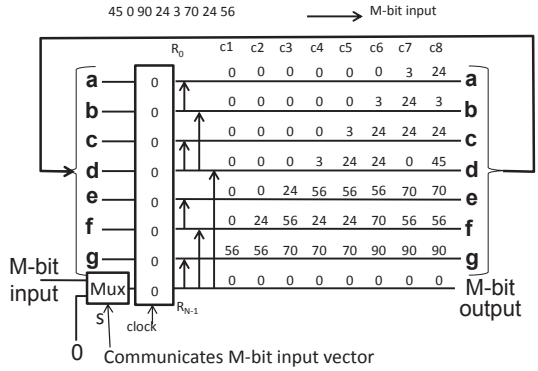

Fig. 11. An example of communication-time accumulation of input data items.

items will be moved up and accommodated somehow in the registers. Indeed, since the bottom line (marked as M-bit output) always contains the smallest value [23], any incoming item is either the smallest, or will be moved up. Figure 11 demonstrates how N M-bit items are accommodated, using an example with $N = 8$ items arriving in the following sequence: 1) 56; 2) 24; 3) 70; 4) 3; 5) 24; 6) 90; 7) 0; 8) 45.

Data may be received from a host system (such as ARM [24]) and accommodated in the registers R_0,..., R_{N-1} during communication time in N clock cycles indicated in Fig. 11 by symbols $c1, ..., c8 (N = 8)$. As soon as N sorted data are received, the sorted result can be transferred immediately to the host system as shown in Fig. 12.

Now the multiplexer Mux communicates the maximum possible data value (m) to the register R_{N-1} and x is 0. Since $x = 0$, the maximum value will always be moved up at each clock cycle [23] enabling real-time transmission of the sorted items (through the M-bit output) in ascending order. To transmit the sorted items in descending order, it is necessary to set x to 1 and to replace the maximum possible value for data items (m) that is supplied to the multiplexer M with the minimum possible value.

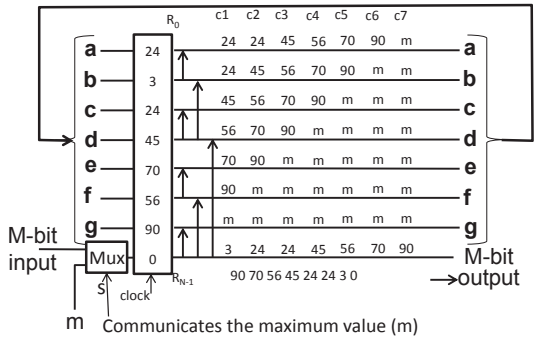

Fig. 12. An example of transmitting sorted data items.

The experiments have demonstrated that the circuit shown in Fig. 10 for $N = 512$, $M = 32$ can be built even for relatively small FPGAs, such as those available in the Nexys-4 prototyping board of Digilent. For advanced FPGAs, such as those from the Xilinx Virtex-7 family, the communication-time data sorter may be built for $N > 4096$. The results of experiments and comparisons will be given in the next section. Note once again that the communication-time circuits described above are advantageous for small autonomous sorters, which need the result to be produced immediately after the last item is received. In particular, they do not give any advantage for the methods and architectures described in Section 2. Therefore, the methods described in Section 2 are beneficial for sorting large data sets and the methods considered here are beneficial for sorting small data sets.

4. EXPERIMENTS AND COMPARISONS

The system for data transfers between a host PC and an FPGA has been designed, implemented, and tested. Experiments were done in the VC707 prototyping board [25] that contains Virtex-7 XC7VX485T FPGA from the Xilinx 7th series with PCI express endpoint connectivity "Gen1 8-lane (x8)". All circuits were synthesized from the specification in VHDL and implemented in the Xilinx Vivado 2016.1 design suite. Software programs in the host PC run under the Linux operating system and they were developed in C language. The data were transferred from from the host PC to VC707 and back through the PCI express. The host PC is based on Intel core i7 3820 3.60 GHz.

The experiments were done in accordance with Fig. 4. The maximum size of data that are entirely sorted in the FPGA is 256 KB. For a larger size of data additional merging is done in the host PC. The results are presented in Fig. 13. It is clearly seen that the sorting throughput for the proposed systems is significantly better than in the host PC. For example, 1024 KB data can be sorted in the proposed system in 16 ms and in the host PC in 110 ms. The comparison of the time of sorting reported in the referenced papers and the results of Fig. 13 clearly shows that the proposed solutions are faster. Figure 14 demonstrates the organization of the experiments for communication-time data sorters (see Section 3).

Now autonomous circuits applicable to small data sets are synthesized, implemented, and tested. We have used a relatively low-cost Digilent Nexys-4 prototyping board with Xilinx Artix-7 FPGA xc7a100 [26]. N initial unsorted 32-bit data items ($M = 32$) are generated randomly and supplied to the communication-time data accumulator/sorter through the M-bit input (see Fig. 10). The clock frequency for data transfers was chosen to be 100 MHz (that is the default frequency of the on-board oscillator). An initial unsorted set of data is supplied and the sorted set is transmitted back entirely within $2 \times N$ clock cycles, which is just the time for data communication.

Table 1 displays the hardware resources that were used, as obtained from the Vivado post-implementation reports (including supplementary circuits, such as random number generation (RND)). Clearly, circuits for

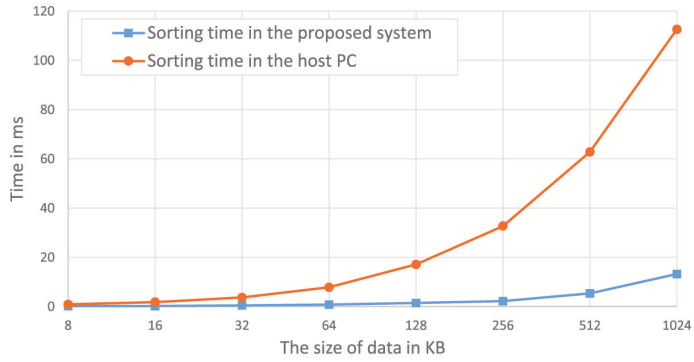

Fig. 13. An example of transmitting sorted data items.

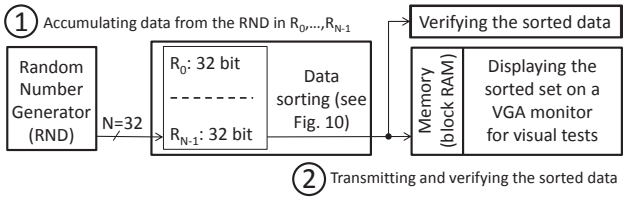

Fig. 14. Experimental setup.

Table 1. The hardware resources used for the Nexys-4 prototyping board

	$N = 64$	$N = 128$	$N = 256$	$N = 512$
Lookup tables	3760 (6%)	7448 (12%)	20699 (33%)	35645 (56%)
Flip-flops	2213 (2%)	4276 (3%)	8405 (7%)	16643 (13%)

significantly larger values of N than in the known even-odd merge and bitonic networks [19,21] have been built. The design proposed is also faster. Indeed, solving similar problems to those in Table 1 in networks [19,21] requires data to be copied to a long register that provides network inputs. The size S of this register, even for the smallest number of $N = 64$ in Table 1, is equal to $N \times N = 2048$ and if $N = 512$, then $S = 16384$ bits. Commercial FPGAs do not have such a large number of external pins and data items need to be copied sequentially and multiplexed to different sections of the register. Similarly, the sorted items must be segmented and transmitted back sequentially through the relevant interfacing circuits. If we consider on-chip communications (such as those available for Zynq all programmable systems-on-chip – APSoC [25]), we can see that the maximum number of high-performance AXI interfaces is 5 and the maximum number of bits transferred through each interface is 64. Thus, multiplexing is also necessary, which involves additional delays and resources. In the proposed design, the circuit itself receives and transmits data in parallel with sorting and no additional resources are required. The number of combinational levels in the proposed circuit is equal to $\lceil \log 2N \rceil$ and it is less than for the networks [19,21] where it is equal to $\lceil \log_2 N \rceil \times (\lceil \log_2 N \rceil - 1)$.

5. CONCLUSION

The paper proposes two architectures that are applicable to sorting large and small data sets. The distinctive feature of the first architecture is parallelization at several stages with the adjusted time. The first stage is data sorting, which is done in such a way that data acquisition, sorting, and transferring the sorted data are carried out at the same time. The second stage is a pipelined RAM-based merger that enables merging at different levels to be done in parallel and it can also be combined with the first stage. Such a type of processing is efficient for sorting large sets (tens and hundreds of millions of data items). The distinctive feature of the second architecture is communication-time processing, which permits sequential transfer of the results of sorting immediately after the last data items have been received. Such a type of processing is often needed for autonomous sorter operations over a relatively small number of data items (from hundreds to thousands of items). Thus, the proposed architectures complement each other. The experiments were done with an advanced prototyping system (allowing data processing in a general-purpose computer and in a recent FPGA from the Virtex-7 family of Xilinx) and with autonomous circuits implemented in a low-cost FPGA from the Artix-7 family of Xilinx. The results of experiments demonstrate significant acceleration compared to general-purpose software and the results reported in publications.

ACKNOWLEDGEMENTS

This research was supported by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, the Study IT in Estonia Programme, Estonian Association of Information Technology, and Telecommunications and Portuguese National Funds through FCT – Foundation for Science and Technology, in the context of the project UID/CEC/00127/2013. The publication costs of this article were covered by the Estonian Academy of Sciences.

REFERENCES

1. Knuth, D. E. *The Art of Computer Programming. Sorting and Searching*, *Vol. III*. Addison-Wesley, 2011.
2. Sklyarov, V. and Skliarova, I. High-performance implementation of regular and easily scalable sorting networks on an FPGA. *Microprocess. Microsyst.*, 2014, **38**(5), 470–484.
3. Mueller, R., Teubner, J., and Alonso, G. Sorting networks on FPGAs. *Int. J. Very Large Data Bases*, 2012, **21**(1), 1–23.
4. Ortiz, J. and Andrews, D. A configurable high-throughput linear sorter system. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2010, 1–8.
5. Zuluaga, M., Milder, P., and Puschel, M. Computer generation of streaming sorting networks. In *Proceedings of the 49th Design Automation Conference*. ACM, New York, 2012, 1245–1253.
6. Singh, S. and Greaves, D. J. Kiwi: synthesis of FPGA circuits from parallel programs. In *Proceedings of the 16th IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2008, 3–12.
7. Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the 2008 Symposium on Application Specific Processors*. IEEE, 2008, 101–107.
8. Chamberlain, R. D. and Ganesan, N. Sorting on architecturally diverse computer systems. In *Proceedings of the 3rd International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. ACM, New York, 2009, 39–46.
9. Mueller, R. *Data Stream Processing on Embedded Devices*. Ph.D. thesis, ETH, Zurich, 2010.
10. Koch, D. and Torresen, J. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, 2011, 45–54.
11. Sklyarov, V., Skliarova, I., Mihhailov, D., and Sudnitson, A. Implementation in FPGA of address-based data sorting. In *Proceedings of the 21st International Conference on Field-Programmable Logic and Applications*. IEEE, 2011, 405–410.
12. Kipfer, P. and Westermann, R. GPU Gems 2, Improved GPU Sorting. `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html`, 2005 (accessed 08.06.2016).
13. Gapannini, G., Silvestri, F., and Baraglia, R. Sorting on GPU for large scale datasets: a thorough comparison. *Inf. Process. Manage*, 2012, **48**(5), 903–917.
14. Ye, X., Fan, D., Lin, W., Yuan, N., and Ienne, P. High performance comparison-based sorting algorithm on many-core GPUs. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2010, 1–10.
15. Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, 1–10.
16. Cederman, D. and Tsigas, P. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th Annual European Symposium on Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2008, 246–258.
17. Grozea, C., Bankovic, Z., and Laskov, P. FPGA vs. multi-core CPUs vs. GPUs: hands-on experience with a sorting application. In *Facing the Multicore-Challenge* (Keller, R., Kramer, D., and Weiss, J. P., eds). Springer-Verlag, Berlin, Heidelberg, 2010, 105–117.
18. Edahiro, M. Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE, 2009, 230–233.
19. Aj-Haj Baddar, S. W. and Batcher, K. E. *Designing Sorting Networks. A New Paradigm*. Springer, 2011.
20. Marcelino, R., Neto, H. C., and Cardoso, J. M. P. A comparison of three representative hardware sorting units. In *Proceedings of the 35th Annual IEEE Conference on Industrial Electronics*. IEEE, 2009, 2805–2810.
21. Batcher, K. E. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*. ACM, New York, 1968, 307–314.
22. Lacey, S. and Box, R. A fast, easy sort: a novel enhancement makes a bubble sort into one of the fastest sorting routines. *Byte*, 1991, **16**(4), 315–320.
23. Sklyarov, V. and Skliarova, I. Fast regular circuits for network-based parallel data processing. *Adv. Electr. Comput. Eng.*, 2013, **13**(4), 47–50.
24. Xilinx, Inc. *Zynq-7000 all programmable SoC. Technical Reference Manual*. `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2016 (accessed 01.02.2017).

25. Xilinx, Inc. *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*. https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf, 2016 (accessed 01.02.2017).
26. Digilent, Inc. *Nexys4 DDR FPGA Board Reference Manual*. https://reference.digilentinc.com/_media/nexys4-ddr:nexys4ddr_rm.pdf, 2016 (accessed 08.06.2016).

Kiired iteratiivsed ahelad ja RAM-i baasil ühendajad, kiirendamaks andmete sortimist riist- ning tarkvara süsteemides

Valery Sklyarov, Iouliia Skliarova, Artjom Rjabov ja Alexander Sudnitson

On välja pakutud ja kirjeldatud kaks arhitektuuri paralleelsete andmete sortimiseks. Esimene on mõeldud suuremahuliste andmekogude jaoks, ühendades kolm andmete töötlemise astet: andmete sortimine riistvaras (FPGA-s), eelsorditud andmete ühendamine riistvaras (FPGA-s) ja seejärel nende suurte alamhulkade üldotstarbeline ühendamine tarkvara abil. Andmete vahetamine FPGA ja üldotstarbelise arvuti vahel toimub läbi PCI ekspress-siini. Teine arhitektuur on rakendatav väiksemate andmekogude puhul, võimaldades sortimist andmete samaaegse vastuvõtuga, st kui viimane andmete osa on käes, võib sorditud osad kohe edasi saata. Võrreldes erinevate varem avaldatud tulemustega, kus on kasutatud tarkvaralisi lahendusi, näitavad katsetulemused pakutud arhitektuuride eeliseid, mis lubavad vähendada vajaminevaid riistvararessursse ja suurendada tootlikkust.

CURRICULUM VITAE

Personal data

Name: Artjom Rjabov

Date of birth: 23.08.1988

Place of birth: Orenburg, Russia

Citizenship: Estonian

Contact data

Address: Järveotsa tee 7-23, Tallinn, Estonia

Phone: +37258160184

E-mail: artjom.rjabov@gmail.com

Education

2013 – 2017 Tallinn University of Technology PhD

2010 – 2013 M.Sc. in Computer Engineering, Tallinn University of Technology

2007 – 2010 B.Sc. in Computer Engineering, Tallinn University of Technology

Professional employment

2015 – … Tallinn University of Technology, Faculty of Information Technology, Department of Computer Engineering; Early Stage Researcher

2012 Web Developer, WeDo OÜ

2010 – 2011 SQA Partners; QA Engineer

Awards

2016 Ustus Augur grant, Estonian Association of Information Technology and Telecommunications (ITL)

2015 – 2016 "IT Academy" scholarship for PhD students (Information Technology Foundation for Education)

2013 – 2014 "Tiger University" scholarship for ICT PhD students (Information Technology Foundation for Education)

ELULOOKIRJELDUS

Isikuandmed

Nimi: Artjom Rjabov

Sünniaeg: 23.08.1988

Sünnikoht: Orenburg, Venemaa

Kodakondsus: Eesti

Kontaktandmed

Aadress: Järveotsa tee 7-23, Tallinn, Estonia

Telefon: +37258160184

E-mail: artjom.rjabov@gmail.com

Hariduskäik

2013 – 2017 Tallinna Tehnikaülikool, info- ja kommunikatsioonitehnoloogia õppekava. Doktoriõpe

2010 – 2013 Tallinna Tehnikaülikool, arvuti ja süsteemitahnika. Magistriõpe

2007 – 2010 Tallinna Tehnikaülikool, arvuti ja süsteemitahnika. Bakalaurusõpe

Teenistuskäik

2015 – … Tallinn University of Technology, Faculty of Information Technology, Department of Computer Engineering; Early Stage Researcher

2012 WeDo OÜ, Programmeerija

2010 – 2011 SQA Partners; QA inseneer

Teaduspreemiad

2016 Ustus Aguri nimiline stipendium (Eesti Infotehnoloogia ja Telekommunikatsiooni Liit)

2015 – 2016 IT Akadeemia stipendium doktorantidele (Hariduse Infotehnoloogia Sihtasutus)

2013 – 2014 Tiigriülikooli stipendium IKT doktorantidele (Hariduse Infotehnoloogia Sihtasutus)

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.

111. **Syed Saif Abrar**. Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC. 2016.

112. **Arvo Kaldmäe**. Advanced Design of Nonlinear Discrete-time and Delayed Systems. 2016.

113. **Mairo Leier**. Scalable Open Platform for Reliable Medical Sensorics. 2016.

114. **Georgios Giannoukos**. Mathematical and Physical Modelling of Dynamic Electrical Impedance. 2016.

115. **Aivo Anier**. Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems. 2016.

116. **Denis Firsov**. Certification of Context-Free Grammar Algorithms. 2016.

117. **Sergei Astatpov**. Distributed Signal Processing for Situation Assessment in Cyber-Physical Systems. 2016.

118. **Erkki Moorits**. Embedded Software Solutions for Development of Marine Navigation Light Systems. 2016.

119. **Andres Ojamaa**. Software Technology for Cyber Security Simulations. 2016.

120. **Gert Toming**. Fluid Body Interaction of Biomimetic Underwater Robots. 2016.

121. **Kadri Umbleja**. Competence Based Learning – Framework, Implementation, Analysis and Management of Learning Process. 2017.

122. **Andres Hunt**. Application-Oriented Performance Characterization of the Ionic Polymer Transducers (IPTs). 2017.

123. **Niccolò Veltri**. A Type-Theoretical Study of Nontermination. 2017.

124. **Tauseef Ahmed**. Radio Spectrum and Power Optimization Cognitive Techniques for Wireless Body Area Networks. 2017.