Preface

This year, the IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2008) is held as part of the IFIP World Computer Congress, held in Milan on September 7-10, 2008. The embedded systems world has a great deal of experience with parallel and distributed computing. Many embedded computing systems require the high performance that can be delivered by parallel computing. Parallel and distributed computing are often the only ways to deliver adequate real-time performance at low power levels.

This year's conference attracted 30 submissions, of which 21 were accepted. Prof. Jörg Henkel of the University of Karlsruhe graciously contributed a keynote address on embedded computing and reliability. We would like to thank all of the program committee members for their diligence.

Wayne Wolf, Bernd Kleinjohann, and Lisa Kleinjohann

Sorting Units for FPGA-Based Embedded Systems

Rui Marcelino, Horácio Neto, and João M. P. Cardoso

Abstract Sorting is an important operation for a number of embedded applications. As sorting large datasets may impose undesired performance degradation, acceleration units coupled to the embedded processor can be an interesting solution for speeding-up the computations. This paper presents and evaluates three hardware sorting units, bearing in mind embedded computing systems implemented with FPGAs. The proposed architectures take advantage of specific FPGA hardware resources to increase efficiency. Experimental results show the differences in resources and performances among the three proposed sorting units and also between the sorting units and pure software implementations for sorting. We show that a hybrid between an insertion sorting unit and a merge FIFO sorting unit provides a speed-up between 1.6 and 25 compared to a quicksort software implementation.

Key words: sorting, FPGAs, embedded systems, special-purpose architecture

1 Introduction

Search and sorting are becoming important operations for embedded computing. Even modest devices are being furnished with amounts of storage that were unthinkable only a couple of years ago. Handheld portable devices, such as PDAs and cell phones, have now the capacity to store large datasets and finding the contents the user wants is becoming critical. For example, an MP3 player with 160 GB can store about 40,000 songs!

Rui Marcelino
UALG/EST – Campus da Penha – Faro, Portugal
e-mail: rmarcel@ualg.pt

Horácio Neto · João M. P. Cardoso
UTL/IST/INESC-ID – Rua Alves Redol – Lisboa, Portugal
e-mail: hcn@inesc.pt, jmpc@acm.org

---

Also, new emerging applications, like sensor data logs, internet traffic, transactions logs, where the information occurs in the form of data streams [1], show how important are database and data stream management systems. The performance of queries in these systems is often dominated by the cost of the sorting algorithm [2]. Hence, sorting units able to improve performance may play an important role.

Our goal is to research efficient sorting units to couple to a general purpose processor (GPP) for FPGA-based embedded systems (see Figure 1). In this paper we present three sorting units and compare the execution time of those units to pure software solutions (e.g., quicksort). The three sorting units proposed explore parallel processing, streaming, and FPGA resources. To combine key properties of those sorting units we also present and evaluate a hybrid sorting unit.

This paper is organized as follows. In section 2 we review related work on sorting machines. Section 3 describes our proposed architectures. Section 4 shows experimental results. Finally, section 5 draws some conclusions.
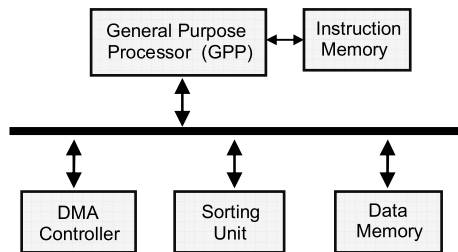
Fig. 1 Block diagram of the target system. The *sorting units* are instantiated as an OPB custom core and the data to be sorted are stored in BRAMs connected to the OPB bus.

2 Background and Related Work

Sorting has been exhaustively studied in the area of computer science and many sorting algorithms exist [3]. On GPPs, quicksort is the fastest of the common sorting algorithms for general case sorting [3]. Albeit the performance of quicksort, sorting remains a time spending operation.

A number of approaches have been studied to accelerate sorting operations on GPPs, namely the use of hyper-threaded technology to accelerate quicksort in the Intel compiler [4], and the use of graphics processors [2].

Concerning application-specific architectures, two different approaches have been considered for accelerating sorting operations, one focusing on variations on the sorting networks [5], and the other exploring systolic linear arrays [9]. Although those approaches may achieve high-performance sorting, both rely on a large number of simultaneous load/stores to feed the sorting unit. This hampers their practical use with current technology.

Sorting networks are based on levels with arrays of 2-input swap-comparators. Martinez et al. [6] propose, for the Burrows Wheeler Transform operation, a hardware sorting network with two levels of pipelining, where the data is sorted in an iterative scheme. The sorting unit deals with 128 characters and results show a large FPGA area occupation and a maximum clock frequency of about 50 MHz.

Zhang and Zheng [7] present a parallel sorting algorithm using a fixed size sorting network. Their architecture is composed by three components: input queues, pipelined sorting network, and a termination detection circuit. Results for different queues size and numbers are shown.

Lin and Liu [8] propose a cascade of compare-swap cells to build the sorting circuit. The data to be sorted propagate through the sorting unit. They argue their approach is scalable and is suitable for VLSI implementations. However, they present an ASIC implementation in a 0.32 m CMOS technology, dealing only with 32 elements of 16 bits and achieving a 66 MHz maximum clock frequency.

Parahami and Kwai [9] propose a cell for systolic linear arrays where the control signals are pipelined with the data to be sorted. In their work, two parallel comparisons are performed in each cell. Bednara et al. [10] present a hybrid hardware/software implementation of a sorting algorithm that uses merge-sort for its sequential part and a Parhami and Kwai [9] type systolic array for the parallel part. In their approach, the sorting unit is implemented in FPGAs and is coupled to a microprocessor.

Recently, Ratnayke and Amer [11] propose an FPGA implementation variation of the counting sort algorithm. This algorithm is a histogram based sorter and explores the BRAM structures of the FPGAs for the modified counting sort algorithm. The sorting unit was implemented in a Virtex II-Pro FPGA and the results show that a significant number of FPGA resources is required to sort a large number of elements.

In this work, we exploit three different sorting units to couple to a host softcore processor, bearing in mind the trade-off between hardware resources and performance. The target system is tested using FPGA devices. Next sections describe those sorting units.

3 Sorting Units

The three approaches for hardware sorting units proposed herein are:

- Odd-Even Sorting Network Machine, based on sorting networks where we reduce the traditional area used for sorting network implementations by using an iterative scheme.
- Insertion Sorting Machine, based on a scalable and linear array.
- FIFO-based Merge Sorting Machine, based on the available and efficient FPGAs FIFO support using BRAMs.

Next, we describe in detail each one of the sorting units mentioned above.

## *3.1 Odd-Even Sorting Network Machines*

Hardware solutions using sorting networks, such as the one proposed in [7], require a large number of hardware resources to implement the complete network. To save hardware resources we propose a solution based on Batchers odd-even [5] sorting network with reuse of resources. We use an iterative sorting unit, where the sorting network is reduced to a single row. In this approach, the maximum computational time complexity is $O(n)$, being n the number of elements to sort.

The basic element of sorting networks is the comparator-swap block, shown in Figure 2(a), which performs the elementary sort between two elements. The block receives the two data elements to be sorted A, B and outputs the two sorted elements L and H, where L means "less than" or "equal", and H means "greater than". In addition a /CHANGE signal flags if a swap between the two input data elements has been done or not.

As the hardware implementation of the sorting network may require too many resources, especially when dealing with a large number of inputs, a split of the network in iterative sequential stages is performed. On this implementation, referred herein as "*sequential network*", the hardware is reused to implement all the required computing stages of the sorting network with a smaller number of physical stages. Note that this sorting network requires a simple control unit and is used for its simplicity, regularity and scalability.

Two schemes have been implemented using the odd-even transposition sorting algorithm. The first one, named sorting network with one pipeline level (SN-I), refers to a machine employing hardware reuse in every clock cycle. For this, a basic comparator-swap is used as shown in Figure 2(a), but without output registers. Registers are placed at the end of the stage to store the results every clock cycle (see Figure 2(b)). A switch network, implemented by an array of multiplexers, is included between the comparators and the output registers. The switch network is responsible for the data alignment, as show in Figure 3, then the output is fed to the input of the unit and this loop is continuously repeated until the data input items are sorted. Until all the elements become sorted, pairs of elements are switched every clock cycle. The sorting is finished when no swap is performed in two consecutives clock cycles of the machine, considering all comparator-swap blocks, or when it reaches the final number of iterations (n). This is detected by the control logic that reads the output global flag, /CHANGE, which is an AND of all the individual /CHANGE flags.

The second approach implemented is a sorting network with two pipeline levels (SN-II). The sorting scheme used is the same, the odd-even transposition sorting network, but the data alignment is performed by the use of two comparator levels (see Figure 4). Now the comparator-swap blocks have the outputs registered, creating a 2-stage pipelined machine. These two-stages are reused every two clock cycles. As before, the sorting finishes when the control logic detects that no swap was performed on all comparator-swap blocks in two consecutives clock cycles or when it reaches the final number of iteration (n).
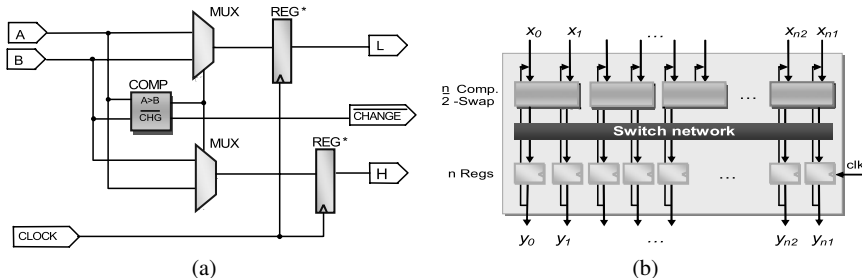
Fig. 2 (a) Comparator-Swap block, the output registers are not used in the SN-I machine implementation (b) SN-I, one pipeline level sorting unit
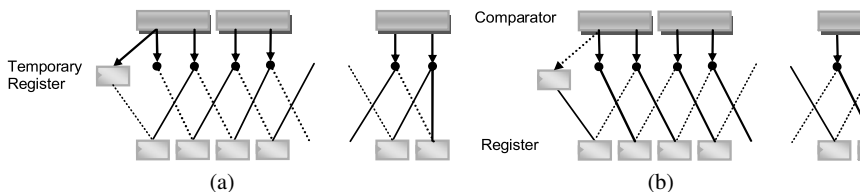

Fig. 3 Switch network for data alignment on SN-I, an extra temporary register have been used: (a) Odd; (b) Even
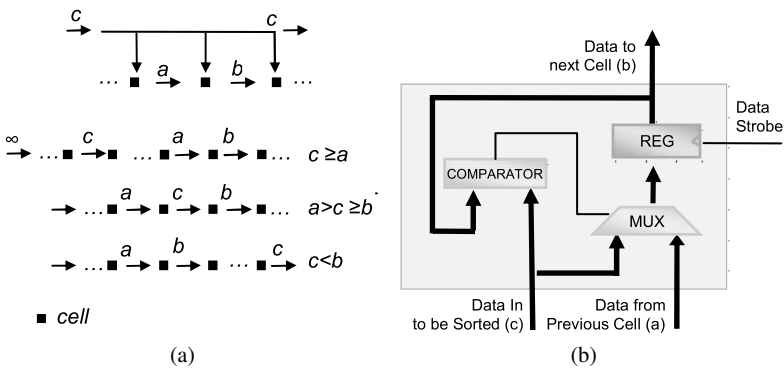
3.2 Insertion Sorting Machine

The insertion sorting machine is represented by the dependence graph shown in Figure 5(a), where each node represents a comparison/insert cell. The number of cells equals the number of elements to be sorted. A new element to be sorted/inserted is broadcasted to all nodes and comparisons are performed in order to find the right node for inserting this new element. Depending on the sort direction, ascending or descending, the most right node reflects the minimum or the maximum element. The data are read from the machine through the right cell in a sequential way (one by one), or in a parallel way. In this machine, the sorting operation is overlapped with the input data operations.

Considering the ascending sorting mode, where the element with the lowest value will be at the right element of the sorting array, as represented in Figure 5(a). In one cell we have an element a from the previous cell, and an element b in the current cell register. For ascending mode of sort the comparator performs the following condition: $b \leq a$. The new element to be inserted c is compared with the data held in all the cell registers. In the general case, where the element a has not the largest possible value, we have one of three possibilities for each cell of the array:
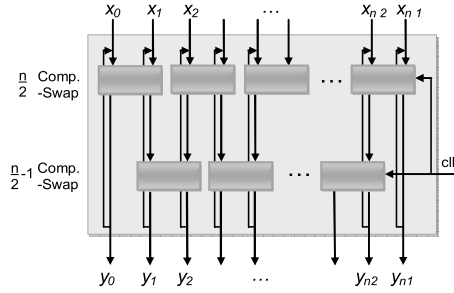
Fig. 4 SN-II, two pipeline levels sorting unit. In this machine the comparator-swap has the output registers.

- $c \geq a$: c is inserted in this cell and the a element and all the elements on its right are right shifted.
- $a > c \geq b$: c is inserted immediately after a and the b element and all the elements on its right are right shifted.
- $c < b$: no change since the insertion point is somewhere on the right of this cell.

Fig. 5 Insertion sort: (a) Dependence graph for insertion sorting in ascending mode with one cell per node (∞ denotes the largest possible value); (b) Basic Cell Comparator- Register.

The basic element of the sorting unit is implemented by the cell showed in Figure 5(b). The cell is composed by a comparator, a multiplexer, a register to hold data, and control logic. The array is composed of a number of these cells, corresponding to the number of elements to be sorted (see Figure 6). Two tags work in a pipeline fashion interconnecting the cells. One tag represents the active cells and works like a carry flag (CY) that is propagated through the cells, as the elements are inserted in the sorting unit. The other tag (LE) reflects the comparison result between the new element to sort and the element presented in the register of each cell. If the new element is greater than the element presented in the register this tag is reset, other

way is set. The two tags drive the control logic located in the cell, in order to define the exact cell where this new element is inserted.
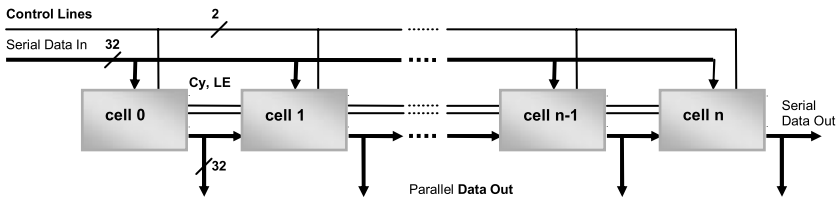

Fig. 6 Insert Sorting circuit block diagram for n-elements.
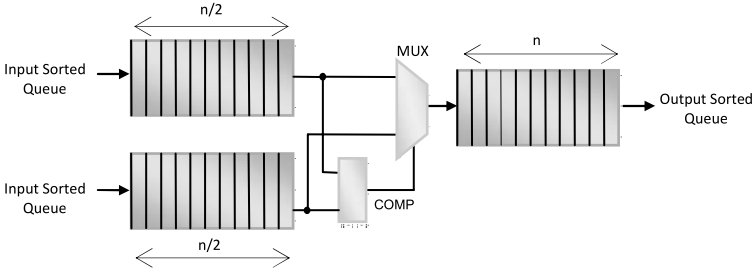
3.3 FIFO-based Merge Sorting Machine

Our FIFO-based merge sorting unit uses the merge scheme shown in Figure 7. The sorting structure consists of three FIFO queues: two input FIFOs and one output FIFO. The input FIFOs have depth $n/2$ and the output FIFO has depth n. This unit assumes that the data in the two input FIFOs have been sorted before.

A truly FIFO-based implementation needs to start by sorting two data elements, each one in a different input FIFO and then repeatedly performs sorting of two sets of $k/2$ elements to achieve k sorted elements until it reaches the last iteration where n elements are sorted based on the two sets of $n/2$ elements previously sorted. This approach might be, however, inefficient and thus a different strategy can be used to feed the FIFO-based merge sorting unit with the two sorted sets of $n/2$ elements each. For example, we can use a Sorting Network Unit or an Insertion Sorting Unit to sort those two sets of $n/2$ elements.

The merging process is performed by presenting the data of the two previous sorted input FIFOs to the inputs of a comparator and a multiplexer. The comparator output defines which element is "greater than" and signals the multiplexer control line in order to select the appropriate element to be written to the output FIFO. A new data element is sorted every clock cycle and the process repeats until all the data are processed. The computational time complexity of this approach is O(n), being n the number of elements to sort.

Although not exploited in this work, it is possible to build sorting units of this kind using more than 2-input FIFOs and more than two levels of FIFOs. Topologies based on trees of FIFOs can be used and might be suitable when it is possible to sort concurrently the data elements in the input FIFOs.

Fig. 7 Block diagram for the FIFO-based merge sorting machine. Two input FIFO and one output FIFO are used.

4 Experimental Results

The sorting units and their control units have been specified in parameterized behavioral RTL-VHDL code. A Xilinx Spartan 3 FPGA (xc3s400-5fg456) has been used to characterize the FPGA implementation of those units. For this particular study we use sorting units working with 32-bit data elements.

The Embedded Development Kit (EDK) and WebISE, release 8.2i, from Xilinx were used for system development and configuration, logic synthesis and placement and routing. For prototyping and test, we use a system with a Xilinx 32-bit MicroBlaze *softcore* processor. For the particular cases presented herein, the sorting units are instantiated as an OPB custom core and the data to be sorted are stored in BRAMs connected to the OPB bus. For data transfer between the BRAMs and the sorting units a DMA controller is included. The MicroBlaze was configured with default parameters, i.e., without the optional datapath units, and without caches. The stack memory was adjusted for the quicksort algorithm requirements. The software implementations were compiled using the C compiler included in the EDK (mb-gcc), with the -O2 option selected.

Our analysis is performed in two steps, one regarding the FPGA resources and the other the execution time of the sorting units being evaluated. All the results obtained by the proposed sorting units are compared with the software algorithm quicksort. In these experiments both the sorting units and the *softcore* processor were running at the same clock frequency (50 MHz for the validations done using the FPGA board). Note, however, that higher speed-ups could be obtained if we consider maximum frequencies for each unit as the maximum clock frequency of MicroBlaze in the FPGA used is around 100 MHz.

Table 1 summarizes the FPGA resources used for the sorting units and the maximum clock frequencies achieved. For the comparisons, we use units with size $n = 128$. The results indicate that the Sorting Network with two levels (SN-II) needs 20% more FPGA resources than the Sorting Network with one level (SN-I). The amount of FPGA resources required for the Insertion Sorting unit is similar to SNI. As can be seen, the FIFO-based merge sorting unit uses mainly BRAMs and much less FPGA resources than the other sorting units.

For execution time analysis, we use sets with 16K 32-bit unsigned integers (N). Those sets were randomly generated (uniform distribution). The data communication between the memory and the sorting unit is performed by a DMA controller.
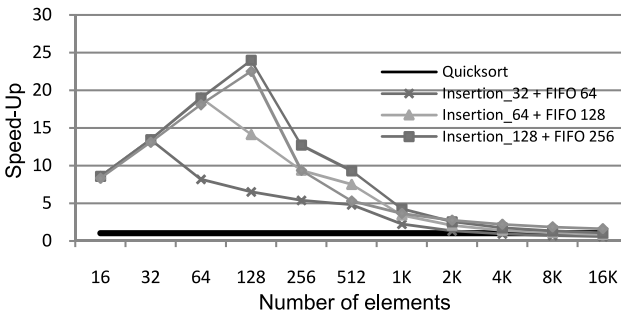
The FIFO-based Merge Sorting Unit requires that two blocks of data with $n/2$ elements be previously sorted and stored in the input FIFOs. In this case, the sorting unit will then give the n elements sorted. To sort those $n/2$ elements we tested the use of a Sorting Network and an Insertion Sorting Unit. The sorting units are able to directly sort a certain pre-defined number of elements (n). Sorting data N size over n needs a merge-sort scheme. For that, we use a software implementation of a merge-sort (identified as software-merge), where each block of data to be sorted (with size n) is sorted by the hardware sorting unit.

Table 1 Maximum clock frequencies and FPGAs resources obtained after Place and Route for the sorting units.

Sorting Unit	LUTs	FFs	Slices	BRAM	Frequency (MHz)
SN I ($N = 128$)	14,629	3,976	7,438	0	80
SN II ($N = 128$)	18,764	8,345	8,906	0	160
Insertion Sorting Machine ($N = 128$)	12,954	4,296	6,486	0	198
FIFO-based Merge Sorting ($N = 128$)[1]	516	444	384	3	104

[1] This machine uses the same resources and achieves the same maximum clock frequency for sizes below or equal $n = 512$.

Figure 8 shows the speed-ups of the hybrid proposed solution (Insertion + FIFObased merge sorting unit) over software quicksort. The hybrid units used here are of size 32, 64, and 128. As can be seen, the speed-up is high and increases with the size of the sorting units. For machines with size 256 with 128 pre-sorted queue, which is the maximum size of sorting units we have experimented with the FPGA used, a speed-up of about 25 has been achieved.


Fig. 8 Speed-ups for different FIFO-based merge sorting units over software quicksort.

As previously referred, when the number of data elements to be sorted surpasses the number of elements sorted by each execution of the sorting unit, a software-merge algorithm is used. In this later case, a degradation in the speed-ups is present (the inflexion points in the chart shown in Figure 8). Note that the software- merge adds a computational time complexity $O(n \cdot \log n)$, being n the number of elements to sort.

Figure 9 gives estimations for sorting a set of 16K elements with three sorting units. We exploit the case of having support for simultaneous load/store operations to communicate data to the sorting units. For the estimations, we use two completely parallel Sorting Networks (SN-II), able to directly sort 16 and 32 elements. The second machine is a 1024-element Insertion Sorting Unit. The third machine is a FIFO-based Merge Sorting Unit able to output 512 sorted elements using two sets of 256 elements sorted by an Insertion Sorting Unit. The results take into account typical DMA load/store latencies, acquired from experimental measurements. For calculating the execution time when sorting 16K elements, the overhead of a *software-merge* has been included.

These results indicate that the Sorting Network SN-II with size 16 (*SN_II_16*) achieves worse results than software quicksort, even with 16 simultaneous load/store operations. The SN-II with size 32 (*SN_II_32*) surpasses quicksort when considering more than 2 simultaneous load/store operations. The Insertion Sort Unit with size 1024 (*Insertion_1024*) achieves for all the cases better performance than quicksort, but since the data is fed to the sorting unit sequentially no gain is obtained by performing simultaneous load/store operations. The highest speed-ups are obtained by the Insertion 256 + FIFO-based merge sorting unit with size 512 (*Insertion_256 + FIFO_512*). In this case, the speed-up increases between 1 to 2 simultaneous load/store operations, as is explained by the fact that this particular unit uses 2 input FIFOs.
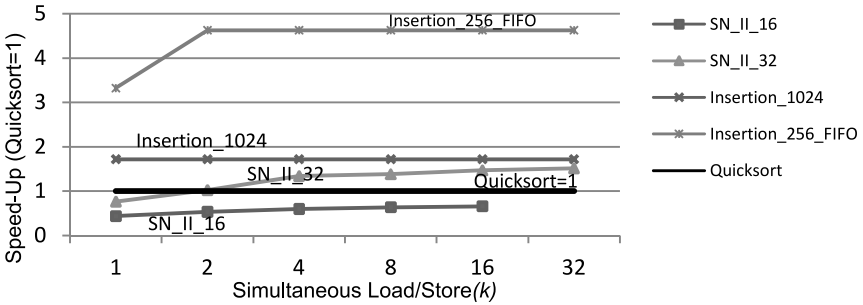
Fig. 9 Speedups for sorting 16K, 32-bit elements, with different sorting units exploring the number of simultaneous load/store operations

For the estimation we use the following equations (1) and (2), adapted from [5]:

$$T_{(n)} = \frac{n}{k}\left(t_{load} + t_{store}\right) + t_{sort\ unit(n)} \tag{1}$$

where $T_{(n)}$ is the total time to sort n elements, considering that n is the maximum number of elements to sort directly on the sorting unit, k represents the simultaneous load/store operations, t_{load} the time to load data from the memory, t_{store} the time to store data in the memory, and $t_{sort\ unit(n)}$ the time required by the sorting unit to sort n elements, considering the data are been loaded

$$T_{software\ merge(N)} = \left[\left(p^2 - p + 4 \right) 2^p - 1 \right] T_{(n)} \tag{2}$$

where $T_{software\ merge(N)}$ is the total time to sort N, elements using software merge, and $p = log(N)$. For larges sorts typically the number N is much greater than n.

5 Conclusions

We describe in this paper three different approaches for hardware sorting units. The sorting units proposed have been coupled to a microprocessor in an FPGAbased embedded system. The sorting units explore different architectures: sorting networks with one or two levels, an insertion sorting array, and a particular sorting unit based on FIFOs. We evaluated these units by coupling them to the peripheral on-chip bus in a system based on a softcore microprocessor (Xilinx MicroBlaze) and implemented in an FPGA. The results show the execution times achieved and the resources needed by each sorting unit. From our preliminary study, the best unit, when a small number of load/store operations can be simultaneously performed (1 or 2), is a hybrid between an insertion sorting and an FIFO-based merge sorting. This sorting unit provides speed-ups between 1.6 and 15 compared to a quicksort pure software solution running in the microprocessor of the system. Even when the number of simultaneously load/store operations is higher (3 or more), the FIFO-based merge sorting unit is from the three units tested in this paper the fastest.

Acknowledgments

References

1. Golab L., Özsu M.T.: Issues in data stream management, ACM SIGMOD Record, v.32 n.2, p.5–14, June, San Diego, California (2003)
2. Govindaraju, N., Raghuvanshi, N., Henson, M., Tuft, D., Manocha, D.: GPUTera- Sort: high performance graphics co-processor sorting for large database management, in Proceedings of the 2006 ACM SIGMOD international conference on Management of data, June 26-29, Chicago, IL, USA (2006)

3. Knuth, D.E.:The Art of Computer Programming, Vol. 3 - Sorting and Searching. Addison-Wesley (1973)
4. Rajiv, R.D.P.:Accelerating Quicksort on the Intel® Pentium® 4 Processor with Hyper-Threading Technology, http://softwarecommunity.intel.com/articles/eng/2422.htm, October (2007)
5. Batcher, K.:Sorting Networks and Their Applications. Proc. AFIPS Spring Joint Computer Conf. Vol. 32, pp. 307–314, Atlantic City, NJ, USA, 30 April - 2 May (1968)
6. Martínez J., Cumplido, R.R., Feregrino, C.:An FPGA-based parallel sorting architecture for the Burrows Wheeler transform, Proceedings International Conference on Reconfigurable Computing and FPGAs, 28-30 Sept., Puebla City, Mexico (2005)
7. Zhang, Y., Zheng, S.Q.: An Efficient Parallel VLSI Sorting Architecture, VLSI Design, vol. 11, no. 2, pp. 137–147, (2000)
8. Lin, C.S., Liu, B.D.:Design of a Pipelined and Expandable sorting Architecture with Simple Control Scheme. IEEE International Symposium on Circuits and Systems, Volume: 4, pp. 217–220, 26-29 May. Scottsdale, Arizona, USA (2002)
9. Parhami, B., Kwai, D.M.: Data-driven control scheme for linear arrays. Application to a stable insertion sorter, IEEE Trans. On Parallel and Distributed Systems, January 1999, Vol. 10, No. 1, pp. 23–28, (1999)
10. Bednara, M., Beyer, O., Teich, J., Wanka, R.: Tradeoff Analysis And Architecture Design Of Hybrid Hardware/Software Sorter, Application-Specific Systems, Architectures, and Processors. Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, Proceedings, pp. 299, 10-12 July, Boston, MA, USA (2000)
11. Ratnayake, K., Amer, A.: An FPGA Architecture of Stable-Sorting on a Large Data Volume : Application to Video Signals, 41st Annual Conference on Information Sciences and Systems, pp. 431–436, 14-16 March, Baltimore, USA (2007)