

A Rapid Sorting Unit based on Programmable Shifting Register Files

Lluís Ribas-Xirgo, David Castells-Rufas, Màrius Montón-Macián, Jordi Carrabina-Bordoll

Abstract—Hardware solutions for fast sorting are usually very area greedy. This paper presents a sorting hardware unit that uses low complexity, scalable shifting register files. We show that such a module takes much less area than other previously known fast sorters, resulting in an ease of implementation. Furthermore, we demonstrate that they are more efficient when integrated in bus-centric systems.

Index Terms—Hardware sorter, parallel sorter, sorter architectures, sorting, searching.

I. INTRODUCTION

IN many data-mining and/or searching engines (as those used in semantic searching applications), parallel architecture machines can cope with huge amounts of data but, in the end, processed data must be combined into a single list through a sorting unit that takes only the best candidates out of the whole.

In the searching problem, for instance, it is possible to use a massively parallel machine (see Fig. 1) to rapidly process incoming data from a number of independent sources, typically massive data storage systems. Each processing element in this kind of machine takes incoming data and determines their scores in terms of their matching with respect to a given search key. The result scores and datum references must be fed into a rapid sorting unit (RSU) able to select only the best scores; i.e. to sort the scores to take the topmost ones.

From the RSU point of view, the requirement is to read a large, unbounded number of score and datum reference pairs, while keeping a score-ordered list of n of such elements.

The reading speed is a critical aspect because it must be as many times as fast as the processing time of score calculating elements by their quantity.

Under this constriction, many works (e.g. [1]) propose solutions based on Batcher’s odd-even or bitonic [2] sorting networks. These sorting networks (see Fig. 2) have all $O(1)$ time complexity and, in fact, are able to output results at the clock rate.

However, they do use $O(n \log n)$ processing elements, with $O(n^2)$ registers in pipelined solutions [3], which are

This work has been supported by the Spanish *Ministerio de Ciencia y Tecnología* under grant MCYT-TIC2001-2508.

The authors are with the *Centre de Prototips i Solucions Hardware-Software* (CEPHIS) and the Microelectronics and Electronic Systems Department of the Universitat Autònoma de Barcelona (UAB), ETSE, Campus UAB, 08193 Bellaterra, Spain. (Phone: +34 935 811 078; fax: +34 935 813 033; e-mail: {Lluís.Ribas, David.Castells, Marius.Monton Jordi.Carrabina}@uab.es).

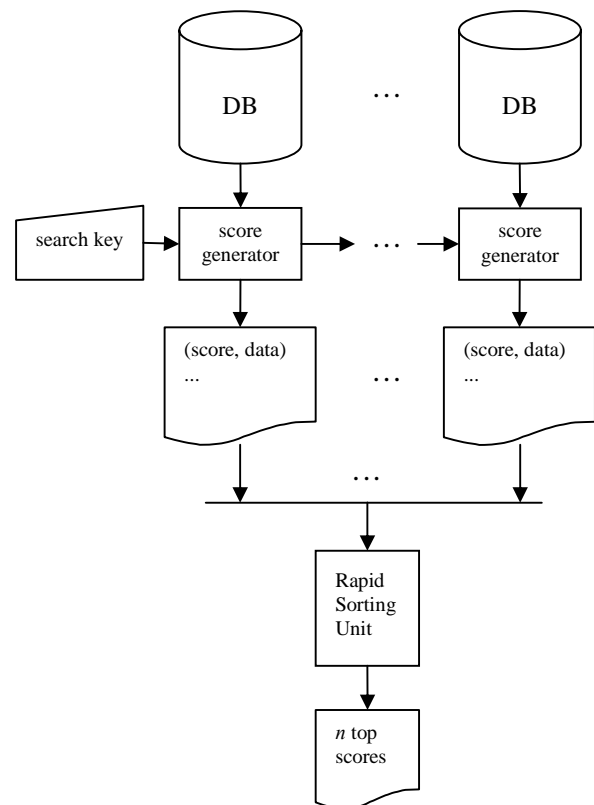


Fig. 1. Architecture of a searching engine.

the most common to avoid excessive combinational delay.

Such solutions are very area-consuming and, to overcome the area occupancy problem, bitonic sorting networks (BSN) are usually “folded”.

In a folded BSN [4], data is re-circulated several times across the net before forming the final, sorted list. The advantages of such solutions are clear. For instance, folding a network into a half would nearly reduce the same rate the result circuitry, while doubling the latency. Note that such folding also implies that there should be included an accompanying interconnection network at the folded BSN input so to make the appropriate connections at every phase of the sorting procedure.

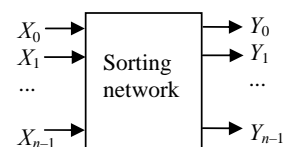


Fig. 2. Sorting network for “single-shot” sorting.

Furthermore, sorting networks do require all elements in the input list be ready at the same time to initiate the sorting, which leads to huge IO demands even for small n . (To sort a set of 32 32-bit numbers a sorting network requires 1024-bit inputs.) Therefore, they also require to be helped by some sort of serial-to-parallel input queues before they can start. In fact, this is a serious problem because it may limit the speed of the sorting task to the one of completely filling the input queues.

Finally, the approaches to the searching problem that use SN must sort $2n$ elements each time (see Fig. 3) to include both a new set of elements and the result set of the previous sorting as the new input list.

In this work we propose an alternative solution to the RSU of the searching problem by only using programmable register files as queues or priority queues are. In fact, there are a few previously published papers that also propose approaches based on linear arrays rather than in SN.

The advantages of linear arrays [5, 6] stem from their lower area complexity and lower IO demands, while their drawback is, basically, the higher latency times for a single sorting.

The rest of the paper is organized as follows. Next section describes the basic processing element of the proposed shifting register files and the architecture. The guidelines to composite several such modules are given in the section afterwards. Before the concluding section, it is presented a detailed approach to a sample problem with emphasis in the area and power saving solutions.

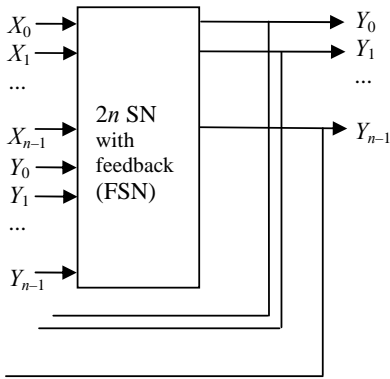


Fig. 3. FSN for gradual sorting.

II. THE SHIFTER SORTER MODULE

This section is devoted to explain the architecture and the basic processing element of the *shifter sorter* module [7]. Such a module is, in fact, some sort of programmable shifting register file that behaves as a priority queue. Thus elements with lower priorities (small scores) are taken out of the register file as other elements with higher scores are pushed in.

For the sake of simplicity, the rest of the section refers to data when speaking about scores. The datum references are not considered here.

A. Architecture of Shifter Sorters

The insertion sort algorithm is a well-known one that repeats for every unsorted datum the same “compare-and-

insert” procedure. It is easily converted into a parallel one by using n PEs which implement the same procedure for every new input. Because of using n PEs, a new datum is sorted at every step (clock cycle). Data with values lower than the new one are shifted to allow insertion, and the lowest value datum is lost.

The resulting register file (Fig. 4) is a simple linear structure that is easily expandable and requires no control at all. For each incoming datum D , PEs in the register file either take the successors’ data or hold their previous values. The one in the insertion point has a successor that holds its value. Connectivity of PEs is local with the exception of input data D .

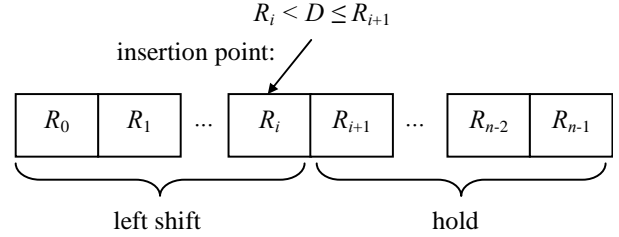


Fig. 4. Architecture of a register file for insertion sorting.

B. The Processing Element of Shifter Sorters

Each PE either writes D or successor’s R_{i+1} value into its register R_i , or holds its value. The logic function of a PE is, then, represented by the result load value into the corresponding register:

$$R_i^+ = p_i \cdot R_i + \overline{p_i} (p_{i+1} D + \overline{p_{i+1}} R_{i+1}) \quad (1)$$

where p_i corresponds to $R_i \geq D$; i.e. to the condition that determines if D should be placed at the left ($p_i = 1$) or at the right ($p_i = 0$). The latter case implies doing a left shift or loading D . Such a decision depends on p_{i+1} so, from the i -th PE, it can be viewed similarly; i.e., as performing a left shift. The last module takes $p_n=1$, as no datum can be placed at its right.

For practical reasons, computation of load data D_i is placed at corresponding modules and the result sent to their left, i.e.:

$$\begin{cases} R_i^+ = p_i \cdot R_i + \overline{p_i} D_{i+1} \\ D_{i+1} = (p_{i+1} D + \overline{p_{i+1}} R_{i+1}) \end{cases} \quad (2)$$

As a result, each PE (Fig. 5) contains a b -bit wide number comparator, one register of the same size, and a 2-1 multiplexer for words of b -bits. The logic to determine where the new data should be placed (on the left or right part of the i -th position) can be performed with the control lines of the MUX.

III. ARCHITECTURES BASED UPON SHIFTER SORTERS

As previously indicated, linear array sorters take less area than sorting networks at the price of processing only one element at a time. Besides, they cannot be pipelined as networks can. However, these linear structures are simple

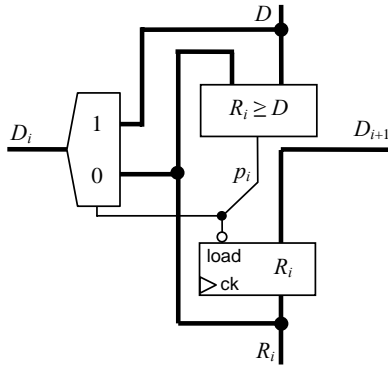


Fig. 5. Processing element of the insert sort register file.

and adapt very well to the searching problem; i.e. to obtaining the topmost n scores out of a large amount of data.

Assuming that k new scores are available at every clock cycle, the sorting subsystem must be able to process them in a single clock cycle. While this is not a problem for network sorters, it requires linear array sorters to be very fast. Ideally, k times faster than the subsystem that processes data and computes corresponding scores.

Obviously, it is not realistic to suppose that linear array sorters might operate at such relatively high speeds. Therefore, it is required to use several sorters in parallel. Assuming that operating frequency is the same for the producer and the consumer (i.e. the sorter), processing k data per clock tick requires k linear sorters.

Provided that $k \ll \log n$, parallel combination of shifter sorters take less area than their sorting networks counterparts.

A. Linear Composition

The simplest form of combining the shifter sorters is the linear one. Basically, it consists of an array of k sorters that takes k data per clock tick. When incoming data has run out, outputs (shifted-out elements) of sorters are taken as inputs of preceding sorters, except for the last one, S_{k-1} , which has infinity as input to cause the whole to shift. (See Fig. 6.)

The control of such linear composition of shifter sorters

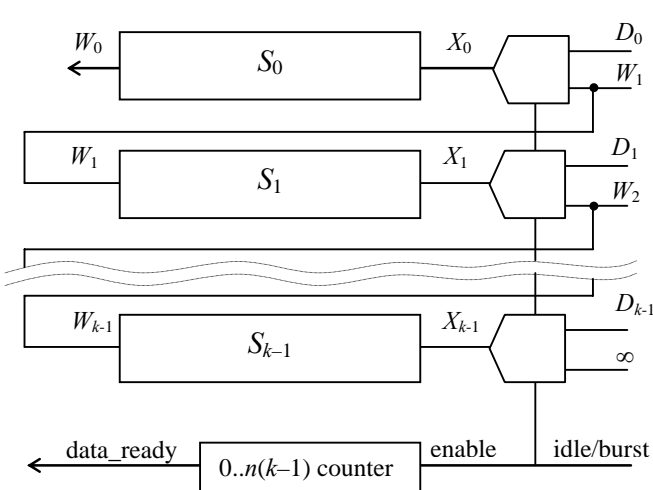


Fig. 6. Linear composition of k shift register files.

consists of a $n(k-1)$ counter that is enabled to count when input data is over and might emit a completion signal when counting is finished. The result is stored in S_0 .

This architecture, however, requires that the data producer is idle, at least, $n(k-1)$ clock cycles before bursting a new stream of data references and scores to be processed.

B. Tree Composition

In order to diminish such latency, a binary tree composition of sorters can be done. This composition results from simultaneous collapsing of groups of 2 linearly linked shifter sorters, thus dividing by 2 the number of register files with true data.

Note that, after $\log k$ iterations, all shifters are “collapsed” into one. A register file collapse is the result of entering n data equal to infinity into one register to cause it to shift out its real data. Consequently, the binary composition of shifter sorters only requires $n \log k$ clock cycles before being ready to accept a new stream of data.

Control circuitry, in this case, is as simple as in the linear architecture. It takes about the same number of multiplexers, and a $n \log k$ counter.

Control lines of MUX are derived from the $\log k$ most significant bits of the counter.

Inputs to sorters X_i are selected by those MUX and inputs to these (Y_j with $j=0 \sim \log k$) are derived from:

$$X_i = (C = 0)D_i + (C > 0) \left[W_{i+2^l} \text{first}(i, l) + \overline{\text{first}(i, l)} \right] \quad (3)$$

where l is the l -th bit of binary number i , $\text{first}(i, l) = (i \bmod 2^{l+1} > 0)$ is used to determine if there is some one in a less significant position than l , and C is the value of the control lines.

The former expression enables automatic generation of such structures and produces a tree like combination of shifter sorters with S_0 being the root and the odd index ones as the leaves.

Fig. 7 shows a tree composition of $k=4$ shift register files. Differences from linear composition architectures occur in the idle/flush MUX inputs: half of them correspond to

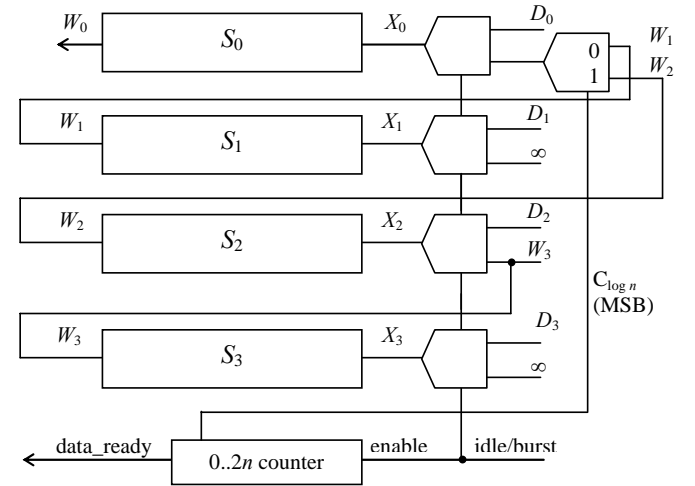


Fig. 7. Tree composition of 4 shift register files.

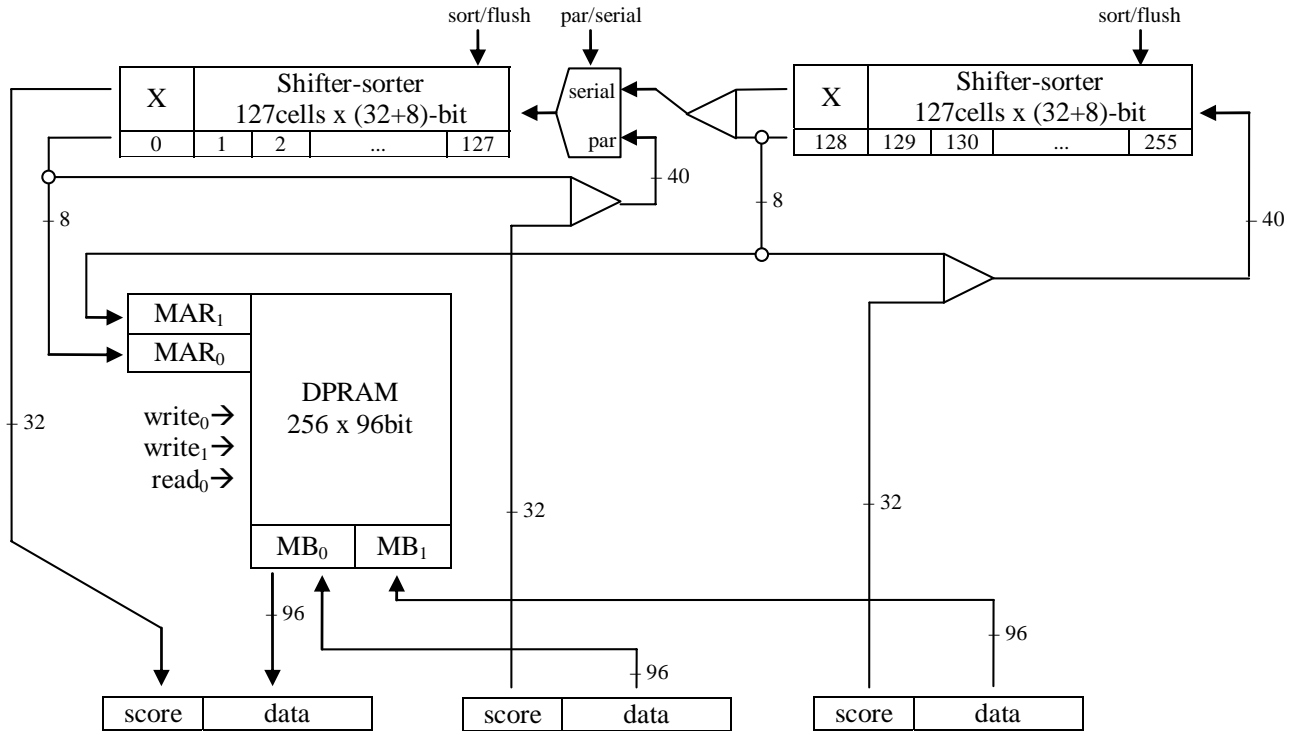


Fig. 8. Complete RSU based on a linear composition of shifter sorters.

“infinity” and the other half are attacked by MUXes that determine which register file output is to be selected in accordance to the progression of the flushing.

IV. FULL RSU EXAMPLE

This section is devoted to the discussion of a complete example of a RSU in the searching problem. For this, we used 128-bit wide input data, with 32-bit wide scores and 96-bit wide datum-references. Such a format certainly enables the system to deal with huge amount of data. We also determined to obtain the 128 or 256 best scores.

Taking into account such numbers, most SN cannot fit in a single FPGA chip solution due to both area and IO requirements, even the folded ones. Contrarily, we shall see that there are a number of solutions when implementing the system with shifter sorters.

In order to simplify PEs and diminish register usage, it is much preferable to use an accompanying RAM to store the datum-references, while PE registers basically store corresponding scores (and associated internal memory addresses).

Much more important than this, the use of dual-port RAM modules enables solutions with shifter sorters to fit in single FPGA chips.

PEs are slightly modified to include a signal to force them to shift their contents, so the result shifter sorter can either insert a new score at its corresponding position or gradually flushing its contents.

By doing this, no infinity values are longer required and,

consequently, most latches are not set to 1 unnecessarily. This mechanism reduces latches’ outputs swings, thus diminishing power consumption.

Fig. 8 illustrates the final schematic of a RSU with 2 shifter sorters in series (linear composition). The behavior is described below.

During the sorting phase, the scores are entered into the corresponding shifter sorters and the associated datum-references are written in the addresses contained in an extra register that buffers the values that are shifted out of the register files. (All internal memory addresses in the registers are conveniently initialized to different memory positions, as illustrated in the figure.)

During the flushing phase, so to obtain the sorted result of the search, the MUX is set to “serial”. The first 128 shifts are done with only the rightmost sorter module set to “flush”, and the next 128 shifts are done with both shifter sorters in mode “flush”. In this last stage, the read₀ signal is set so to obtain the 127 topmost scores (and datum-references) through the output ports.

In order to avoid energy consuming resets, zero scores are input at shifter sorters when operating in mode “flush”. Hence, there is no need to reset device’s latches but at the very beginning of the operation.

The RSU shown in Fig. 8 has been synthesized into a Xilinx 2v6000 with a speed degree of 6. The result circuit used 30K BELS (basic elements), i.e. about 35% of the device area for logic, and 2% of the total available RAM. Clock operating frequency has been slightly over 150MHz.

In order to cope with even more input bandwidth, we had

also simulated a model with four shifter sorters that were linearly composed and also in the tree-like fashion. Of course, if the presented RSU yield fits the needs of a given application, a cheaper device can be used.

V. CONCLUSION

RSU are a key component in parallel machines for searching purposes. In this paper we have introduced the shifter sorter module, which is based on priority queues. We have also shown that these modules are much more efficient than sorting networks with respect to the searching problem, to the area occupancy, and to design simplicity. Besides, we have also suggested architectural solutions to combine several shifter sorters. Finally, we have implemented a working version of the RSU to demonstrate the validity of our approach.

REFERENCES

- [1] K. Claessen, M. Sheeran, and S. Singh, "The Design and Verification of a Sorter Core," *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*. Springer-Verlag Lecture Notes in Computer Science, vol 2144, pp. 355–369, Sep. 2001.
- [2] K. E. Batcher, "Sorting Methods and their Applications," *AFIPS Spring Joint Comp. Conf.*, vol. 32, pp. 307–314, 1968.
- [3] R. Kannan, "A pipelined single-bit controlled sorting network with $O(N \log^2 N)$ bit complexity," *16th. Annual Joint Conf. of the IEEE Comp. and Communications Soc., INFOCOM '97*.
- [4] J.-D. Lee, and K. E. Batcher, "Minimizing communication in the bitonic sorter," *IEEE Trans. on Parallel and Distributed Systems*, vol. 11, no. 5, May 2000.
- [5] B. Parhami, and D.-M. Kwai, "Data-driven control scheme for linear arrays. Application to a stable insertion sorter," *IEEE Trans. on Parallel and Distributed Systems*, pp. 23–28, 1999.
- [6] Chi-Sheng Lin, and Bin-Da Liu, "Design of a pipelined and expandable sorting architecture with simple control scheme," *IEEE International Symposium on Circuits and Systems, ISCAS 2002*, vol. 4, pp. IV-217–IV-220, May 2002.
- [7] Ll. Ribas-Xirgo, D. Castells-Rufas, J. Carrabina-Bordoll, "A Linear Sorter Core based on a Programmable Register File," *XX Conf. on Design of Circuits and Integrated Systems, DCIS 2004*, pp. 635–640, November 2004.