TECHNISCHE FAKULTÄT

Department Informatik

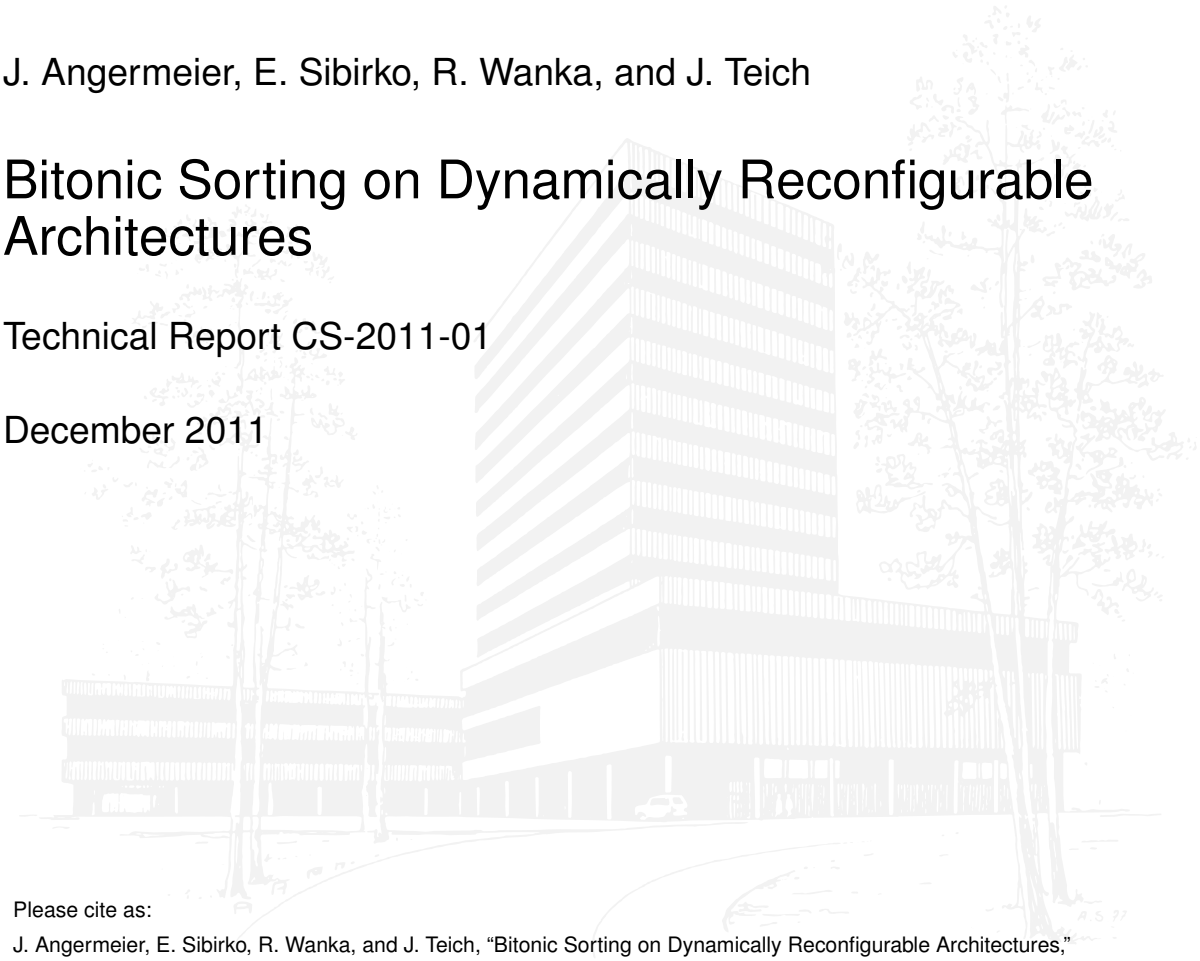
Technical Reports / ISSN 2191-5008

J. Angermeier, E. Sibirko, R. Wanka, and J. Teich

Bitonic Sorting on Dynamically Reconfigurable Architectures

Technical Report CS-2011-01

December 2011



Please cite as:
J. Angermeier, E. Sibirko, R. Wanka, and J. Teich, “Bitonic Sorting on Dynamically Reconfigurable Architectures,”
University of Erlangen, Dept. of Computer Science, Technical Reports, CS-2011-01, December 2011.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.informatik.uni-erlangen.de

Bitonic Sorting on Dynamically Reconfigurable Architectures

J. Angermeier, E. Sibirko, R. Wanka, and J. Teich
Hardware/Software Co-Design, Department of Computer Science University of Erlangen-Nuremberg, Germany

Abstract

Sorting is one of the most investigated tasks computers are used for. Up to now, not much research has been put into increasing the flexibility and performance of sorting applications by applying *reconfigurable* computer systems. There are parallel sorting algorithms (*sorting circuits*) which are highly suitable for VLSI hardware realization and which outperform sequential sorting methods applied on traditional software processors by far. But usually they require a large area that increases with the number of keys to be sorted. This drawback concerns ASIC and statically reconfigurable systems.

In this paper, we present a way to adopt the well-known Bitonic sorting method to dynamically reconfigurable systems such that this drawback is overcome. We present a detailed description of the design and actual implementation, and we present experimental results of our approach to show its benefits in performance and the trade-offs of our approach.

I. Introduction and Motivation

Reconfigurable hardware devices combine performance and flexibility. They outperform software implementation by applying parallelization and offer the option to change the implementation after shipping and multiplex different applications after each other. Dynamic partial reconfiguration furthermore allows that only a part of the FPGA (Field Programmable Gate Array) will be reconfigured at runtime. This can be applied to modify and/or add new possibilities to the current configuration on the FPGA, saving by this the time required for the whole reconfiguration of the FPGA which was needed before.

Sorting (long) sequences of keys is one of the classical and most common tasks solved by computers today, e. g., in numerous database and computer graphics applications. The problem of sorting consists of receiving a sequence of n keys that must be rearranged in such a way that they form a non-decreasing sequence. The keys are usually integers or can be described by integers. Since sorting is one of the most often applied computer tasks, it does not surprise that there is a wealth of sorting algorithms, especially comparison-based methods. If in a single parallel step, up to $n/2$ comparisons are executed, a running time of $O(\log n)$ might be achievable. Unfortunately, there are only two such methods known, namely Cole's parallel merge sort [1] for EREW PRAMs and the AKS sorting circuit [2], [3]. But in both methods, the constant factors involved make them infeasible in practice.

An algorithm relevant for practical purposes due to the really small constant factor (see below) involved is Batcher's Bitonic sorting circuit [4]. Sorting circuits are build from comparator modules, or *comparators* for short, that connect horizontally drawn "wires" numbered from 1 to n where the n keys enter on the left side. A single comparator receives two keys and outputs the minimum key on the lower numbered wire and the maximum key on the other wire (e. g., see Fig. 1). The sequence of keys leaves the circuit on the "right."

These circuits are well suited to be realized by a hardware implementation. The Bitonic sorting algorithm performs $\frac{1}{2}\log n(\log n+1)$ parallel steps (also called *stages*) and consists of $\frac{1}{4}n\log n(\log n+1)$ comparators that all have to be implemented in hardware. The drawback of this approach is that only a small fraction of all comparators is active during a single step, i. e., in a certain way the implementation of sorting circuits in hardware leads to a considerable waste of area. Or, the other way around, this means that only short sequences can be sorted. Here, modern hardware with the ability of *dynamic partial reconfiguration* can be of great help: Just a piece of k parallel steps is implemented, and when the steps of the piece have been executed, the hardware can be reconfigured for the execution of the next piece of k parallel steps. So it is possible to sort longer sequences of keys with still all the benefits of parallel processing. In this paper, we demonstrate this approach by applying it to Bitonic sort.

The number of applications of dynamic partial reconfiguration has been increased considerably in the past few years because of the rapid development of the available size of the FPGAs, and by the demand for increased performance and more flexibility. Although sorting is considered as one of the most important computing problems, there are only few approaches to apply reconfigurable devices to speedup sorting (see below). Furthermore, especially the benefits of the techniques of *partial* dynamic reconfiguration for parallel sorting algorithms have not been investigated yet.

The paper is organized as follows: In Section II other approaches which apply reconfigurable devices and parallel sorting algorithms are presented. In Section III, we explain the general approach of two implementation variants of Bitonic sort which we realized in our work. In Sections IV and V, the partially reconfigurable architectures we used for our experiments and the corresponding design of the realization of the sorting application are presented. In Section VI, the implementation details are explained, then the experimental results are described in Section VII. Finally, in the last section a conclusion of the paper and an outlook on future research activities is given.

II. Background and Related Work

A comprehensive introduction to sequential and parallel sorting can be found in [5]. In particular, Sec. 5.3.4 of [5] presents a thorough overview of sorting circuits.

The Bitonic sorting algorithm was developed by K. E. Batcher [4]. It is based on the classical sorting-by-merging algorithm that merges two sorted halves of a sequence to a completely sorted sequence. In contrast to sequential mergesort, Bitonic sort (as all sorting circuits) is *oblivious*, or non-data dependent, i. e., the same comparisons (with respect to the index positions) are performed regardless of the actual data, and can therefore be easily implemented as a sorting circuit. Bitonic sort executes $\frac{1}{2}\log n(\log n+1)$ parallel steps and is due to the $\frac{1}{2}$-factor, among the fastest sorting circuits for realistic input sizes. Therefore, it is the algorithm which we adopted to reconfigurable systems in this work and present in this paper. Its sorting circuit consists of $\frac{1}{4}n\log n(\log n+1)$ comparators. A theoretically optimal VLSI layout (with respect to Thomson's area-time model [6]) of this circuit has been known for a long time [7]. Unfortunately, this layout is not applicable in practice.

There are just a few reports on the implementation of parallel sorting algorithms on FPGAs. They have in common that they do not use partial reconfiguration. In [8], implementations of quicksort, heap sort, radix sort, Bitonic sort, and odd/even mergesort (a further sorting method by Batcher [4]) are compared on a reconfigurable platform. They are compared to each other with respect to memory bandwidth, clock speed, algorithm computational density and on the algorithms' ability to be pipelined.

In [9], a hardware realization of a recurrent scalable sorting circuit based on Bitonic sorting is presented. The Bitonic sorter has been implemented on an FPGA. Techniques to reduce the communication within the circuit and to minimize the costs in terms of hardware resources are explained. Additionally, an enhancement of the input registers in order to reuse of the same architecture for different input widths is shown in detail. Thus, the time complexity of the original Bitonic sorter could be achieved.

Newer work in this field can be found in [10], [11]. Zhang et al. present a parallel sorting algorithm based on the theory of artificial neural networks. A prototype of the proposed sorting algorithm is implemented in hardware using an FPGA in which a basic sorter can sort 10 keys in $10+1$ clock cycles. These basic sorter units may also be cascaded into a $10n$-keys sorter in order to sort longer sequences of numbers. While our paper focuses on increasing the performance in sorting, their approach focuses on dynamic adoption for sorting different types of data.

Marcelino et al. [11] present and evaluate three different hardware sorting implementations and compare them to the quicksort algorithm implemented in software. Their experimental results show the differences in resources and performance among the three proposed sorting units and also between the sorting units and plain software implementations. By combining an insertion sorting unit and a merge FIFO sorting unit, they achieve a speed-up between 1.6 and 25 compared to a quicksort software implementation. Furthermore, they realized implementations supporting up to 128 parallel inputs.

In contrast to these existing works, our approach newly applies the technique of partially dynamically reconfiguration. In this paper, it is shown that parallel sorting algorithm implementation on FPGAs can highly benefit when used in conjunction with partial reconfiguration. The huge area usage of parallel sorting circuits like the Bitonic sort can be reduced as partial reconfiguration allows to separate the complete sorting circuit in multiple pieces and

allows to load and execute them one after another. Thus, smaller FPGAs can be used and also parallel sorting circuits for a larger number of keys can be realized. This idea is completely new and has not been investigated before.

III. Algorithm

A. Bitonic Sort

The Bitonic sorting circuit is based on the sequential mergesort algorithm. It follows the divide-and-conquer approach. The basic procedure is exemplified in Figure 1. The unsorted sequence of keys enters the circuits on the left. A vertical connection represents a comparator which compares two keys and, if necessary, exchanges them. In Bitonic sort, $n/2$ comparators can be executed in parallel defining a parallel step. In the figure, the corresponding outputs of each comparator after each step are shown. Finally, the completely sorted sequence of keys is output on the right. The sorter consists of a cascaded sequence of Bitonic mergers. In Fig. 1, these mergers are marked with boxes. This design is developed for n being a power of 2. For arbitrary n, one can use the circuit for $2^{\lceil \log n \rceil}$ inputs and remove the unnecessary lower wires and the incident comparators. Bitonic sort belongs with its running time

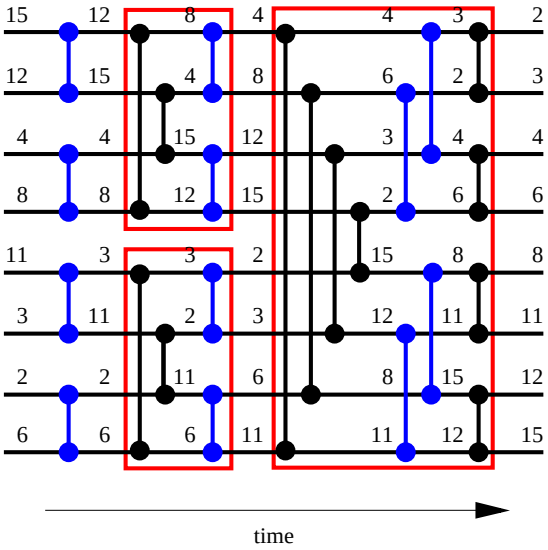


Fig. 1. Bitonic Sort for $n = 8$ keys. Vertical connections denote comparator, the minimum key is sent upwards, the maximum key is sent downwards. Boxed areas denote Bitonic mergers, whose outputs are already sorted. Here, we have 6 stages and 24 comparators.

of $\frac{1}{2} \cdot \log n \cdot (\log n + 1)$ parallel steps to the fastest practical sorting algorithms. But the original algorithm also has its disadvantages it shares with most circuit-based sorting methods. (i) The input sequence must always pass the full sorting circuit in order to guarantee that the result is eventually sorted. That means, there is almost no simple way to recognize an already sorted sequence at the beginning or in the middle of the computation and thereby save the rest of the sorting process. It is even improbable that in the middle of the computation the current sequence is sorted! (ii) It necessarily needs large hardware area, in particular due to the wiring [6]. The more keys are to be sorted by the Bitonic sorting circuit, the more space for comparator stages and wiring it needs, and thus it consumes a lot of space on a chip. (iii) In addition, the space is used inefficiently, as at all times just one single stage is active while the remaining comparator stages remain idle and wait for either the inputs or have their share of work already completed.

The last mentioned disadvantage can be compensated by applying pipelining techniques when several sequences are to be sorted. Thus, the throughput can be increased, and many comparator units are engaged in each time step. In this paper, we focus on (ii). We propose to apply partially dynamic reconfiguration to overcome the large area usage.

In our implementation, we differentiate between two implementation variants, the static version and the dynamic one. Their characteristics are explained in the next section.

B. Static Variant

In the static Bitonic algorithm implementation, the Bitonic sorting circuit is implemented as a whole on the reconfigurable device. The complete circuit is synthesized and loaded into the FPGA in a single step. Thus, the corresponding device must offer enough reconfigurable resources, e. g., comparator units, for the corresponding number of inputs. As the number of comparators increases considerably with the number of inputs, either very huge FPGAs must be engaged or only sorting short input sequences can be supported. Nevertheless, the execution in hardware can speedup the sorting task enormously on an adequate device.

C. Dynamic Variant

In the dynamic variant of our implementation of the Bitonic sorting algorithm, we adopt it to dynamically partially reconfigurable systems. We partition the sorting circuit as illustrated with dotted lines in Fig. 2 into multiple pieces such that each piece includes a set of parallel steps which fit altogether on just a portion of our reconfigurable device. Thus, we separated the complete sorting circuit into multiple smaller pieces, also called partial modules.

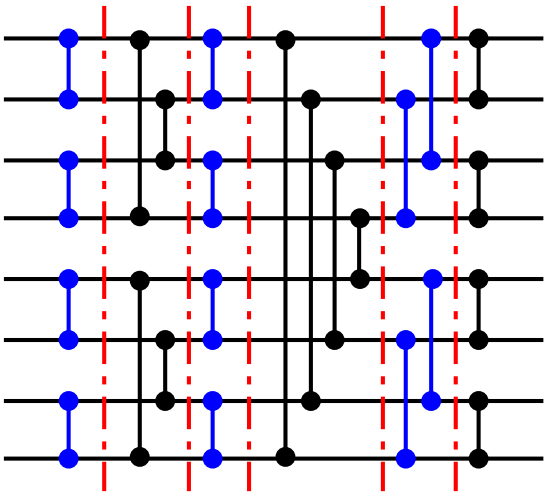


Fig. 2. Exemplary partition of the Bitonic sorting circuit into 6 pieces (here, of one stage each) in order to separate the whole circuit into multiple partial modules.

Each of these modules can be loaded and executed one after another on a reconfigurable device. Furthermore, also multiple modules can be loaded and executed on the reconfigurable device. The advantage of this approach is that less reconfigurable area is needed by the comparison units. This additional space might be used to run different units of the application in parallel or also other applications simultaneously on the FPGA. But the main advantage is that longer sequences can now be sorted and processed in parallel by multiplexing the pieces one after another, until all steps of the circuit are executed.

IV. Design

In Figure 3, an overview of the design of the static algorithm is given. The inputs are prepared by the control CPU and sent via a hardware-software communication module to the main FPGA of the platform. Furthermore, the control CPU is also in charge of the initial static reconfiguration of the main FPGA. The main part of this application is done on the main FPGA. For the static variant, this hardware application consists of two main modules. The first is a *Pre- and Postprocessing* module, the second realizes the actual sorting circuit. This module controls the *hardware-software communication* module, receives the input, and sends the output back to the control CPU. Furthermore, its main task is to de-serialize and serialize the sequentially transferred data. The second hardware module realizes the Bitonic sorting circuit. Its size directly depends on the number of input keys to be sorted.

In comparison to the static version, the design of the dynamic adaptation of Bitonic sort has some important differences. An overview of this design is given in Figure 4. The control CPU is not only in charge of transferring the input and output data and loading the initial configuration into the main FPGA, it also controls the stepwise

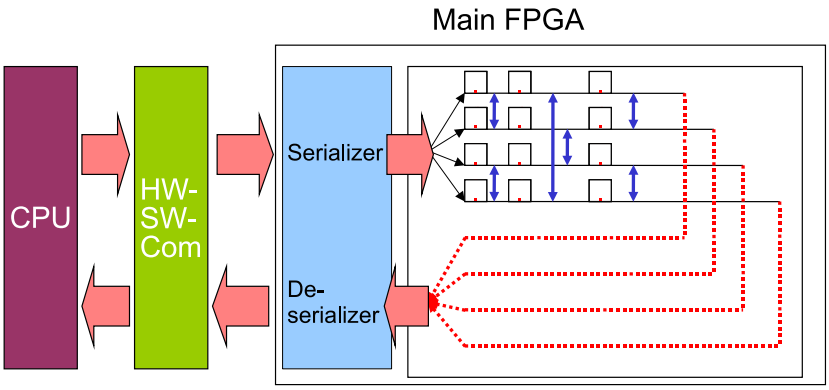


Fig. 3. Design overview of the static implementation of the Bitonic algorithm: The control CPU is in charge of preparing the inputs and outputs. On the hardware side, a serializing/de-serializing unit, and the full Bitonic sorting circuit.

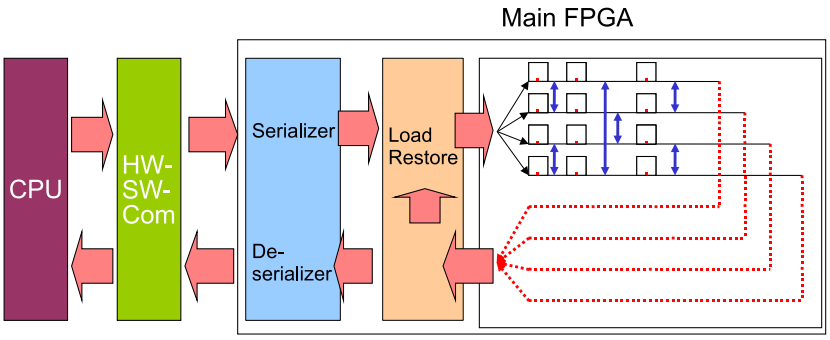


Fig. 4. Design overview of the dynamic Bitonic algorithm approach: The control CPU is in charge of the reconfiguration process and the transfer of the input and output values. On the FPGA, the de-serialization and serialization of the input and output data is done, respectively. A temporary memory unit is used to save and restore the results after executing the partial modules. Furthermore, the main FPGA is loaded with the actual partial sorting module.

reconfiguration of the partial sorting modules. On the hardware side, an additional module is needed for storing and restoring the temporary results in a temporary memory when reconfiguration between the execution of successive pieces takes place. Thus, the control CPU initially loads all static modules and the first partial sorting module, then transfers the input data. After executing the first piece of sorting steps, the results are saved in a temporary memory. Then, the control CPU triggers the reconfiguration of the next partial sorting module according to the Bitonic sorting circuit. Then, the temporarily saved sequence is restored from the temporary memory and fed into the newly reconfigured part of the Bitonic sorter. This procedure is repeated until all partial modules of the Bitonic sorter have been run through and we are guaranteed that the resulting sequence is completely sorted. Finally, the resulting sequence is transferred back to the control CPU.

V. Dynamically Partially Reconfigurable Architecture

Now, a short introduction to the used partially reconfigurable platform is given in order to present the methodology.

For the implementation of the algorithm, a dynamically partially reconfigurable platform called *ESM* is used [12], [13]. The platform is centered around an FPGA serving as the main reconfigurable engine, and an FPGA realizing a crossbar switch. They are distributed to two physical boards called *BabyBoard* and *MotherBoard*, and are implemented using a Xilinx Virtex-II 6000 and a Spartan-II 600 FPGA, respectively. The slot-based architecture of the ESM consists of the Virtex-II FPGA on the BabyBoard, the so-called *Main FPGA*, local SRAM memories, configuration memory, and a reconfiguration manager. Additionally, there is a control CPU, a PowerPC MPC875, which schedules the configuration of the *Main FPGA*.

The advantage of the ESM platform is its unique slot-based architecture arranged in 1-D vertical slots, which allows to configure individual hardware modules independently of their peripheral needs at run-time. A separate crossbar switch is in charge of routing dynamically data from the periphery, e. g., a video input signal, to the current

position of the responsible hardware module. The crossbar is implemented off-chip on the Motherboard in order to have as many resources free on the FPGA as possible for partially reconfigurable modules.

VI. Implementation

Our sorting application consists on the one side of a software part, which is executed on the control CPU. On the other side, it consists of a (reconfiguration using) hardware part. Both components are connected via a hardware-software communication infrastructure. The hardware part consists of two components, namely the sorting circuit itself and *Pre- and Post-processing* units. The hardware description of the sorting circuit component is created with help of a new tool called *HDL Generator for Sorting Circuits*. Both the static and the dynamic Bitonic sorter are realized with partially reconfigurable modules, which may be exchanged during runtime. Note that the sorting circuit of the static Bitonic sorter consists only of a single "partial" module, while the dynamic Bitonic sorter (usually) consists of multiple partial modules. The *Pre- and Post-processing* units include the so-called *Load and Restore* unit, which is also realized as a partial module and only loaded when executing the dynamic Bitonic sorter. Furthermore, the *Pre- and Post-processing units* include the so-called *Deserializer* unit and *Serializer* unit, which are necessary for both Bitonic sorter variants and therefore put into the static part of the reconfigurable system. The static and the partial parts on the reconfigurable device as well as the partial components are connected via a reconfigurable communication infrastructure. In the following, all all these components of the implementation are explained in more detail.

The sorting circuits of the Bitonic sorter shows some regularity, but is rather complex to describe. Additionally, the dynamic variant of our implementation requires to cut the sorting circuit into multiple parts, which must fit on the FPGA and can be executed after each other. Therefore, we created a new tool called *HDL Generator for Sorting Circuits*. It allows to generate the hardware descriptions for the sorting circuit of the static and the dynamic variant of the Bitonic sorter. It can create the comparator circuit of the Bitonic sorter in order to sort an arbitrary number of integers n. Furthermore, it can also generate the individual descriptions of an appropriately partitioned sorting circuit. The program was written in an extendable manner, such that it may also be used by others and support for additional sorting circuits may be added. The mandatory input parameters consist in the name of the sorting circuit to generate, the number of integers to be sorted, and the amount of available area for one reconfigurable module. The last quantity is specified in CLBs. According to later value, the generator tool puts one more set of parallel sorting steps in the current partial module, if there is still enough space or will include it in the VHDL description of the next partial module. Based on the input parameters the new tool will output a VHDL description of the static and partial parts of the demanded sorting circuit, and additionally a so-called map file. In the later one, a specification is given of which partial modules correspond to which parallel sorting stages in the sorting circuit. Furthermore, the map file also reveals information about which partial modules implement equivalent parallel sorting steps, and thus maybe omitted in the final system in order to minimize configuration bit file storage costs.

On the control CPU, a Linux user mode program is in charge of the reconfiguration of the initial static and also the partial sorting modules. Furthermore, it allows to specify the data set to be sorted, sends it to the reconfigurable device, and receives and finally stores the result to a file. The low level reconfiguration handling and communication with the reconfigurable device, respectively, is realized in corresponding libraries and Linux kernel modules.

The *hardware-software communication* module is an interface between the FPGA and its environment like the control CPU. On the hardware side, it provides an input FIFO into which the input data from the environment can be written as well as an output FIFO from which the output data can be read by the environment.

Our reconfigurable sorting modules require point-to-point connections to other partial modules and to the static part of the system. However, runtime reconfiguration is still unusual in commercial systems, and often comes along with a resource overhead for providing a configuration interface and a communication infrastructure that is suitable to integrate modules into a system by partial reconfiguration at runtime. Furthermore, there is still a lack of adequate design tools to support the generation of systems that are capable to use partial reconfiguration as an integrated feature of the system.

For efficiency and reliability reasons, we applied the *ReCoBus-Builder* tool flow approach (see [14],[15], and [16]) in our implementation. It supports the design of dynamically partially reconfigurable systems, especially the synthesis of communication hard-macros, floor-planing and generation of partial modules. It is based on a technique named *I/O bars* for providing point-to-point communication links between the dynamic and the static part of a system and between some reconfigurable modules in a very flexible and resource efficient manner.

The realization of the mentioned technique is very specific to the actual reconfigurable architecture, but besides support for the Xilinx Virtex-II family, which we applied in our implementation, most other popular Xilinx devices are supported, too. In all instances, a horizontal set of wires that are aligned in parallel is reserved. These wires are routed in a regular fashion. Such a set of wires is called an *I/O bar*. Only at the resource slots that require a connection to a particular bar, additional logic is instantiated to provide connection terminals to the modules that have to access this bar. A route through the incoming bar wires or a cut of the routing track can be used to allow a module to read incoming data, process the data, and send the results further towards the bar. Consequently, this technique is ideal for achieving a high throughput in sequential data processing applications.

By using the look-up tables and the separately usable flip-flops, up to eight module input ports can be provided together with eight module output ports within the same CLB. Note, that the double amount of connection terminals is provided as compared to the so-called *bus macros* as proposed by Xilinx [17]. Based on this technique, a bus system called *R*eCoBus may also be realized, which allows to implement all established bus protocols (including AMBA, CoreConnect, Avalon, or Wishbone), however this was not appropriate for our purposes. Our implementation just makes use of the point-to-point connections in order to connect reconfigurable modules and the static and reconfigurable parts.

The different partial modules are vertically aligned to each other. For passing data from module to module as well as for communication between static and partial parts of the design ReCoBus hard-macros called IO bars are used. In Fig. 5, the partially reconfigurable part of the sorting system and the static one are shown based on an view of the Xilinx FPGA Editor. Here, the placement of the ReCoBus hard-macros is highlighted explicitly.

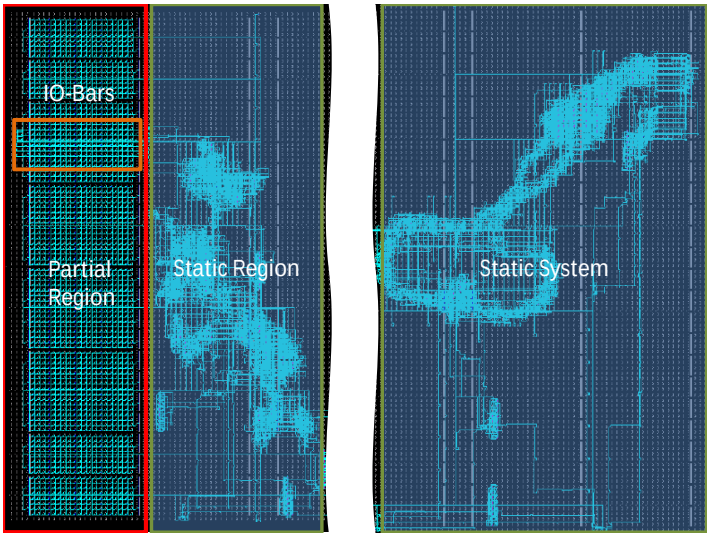


Fig. 5. Xilinx FPGA editor view of the implemented sorting systems: On the left, the partial part, on the right, the static part.

For using sorting circuits on the FPGA some additional modules are required, which process the data before and after a sorting iteration. In Fig. 6 all the modules belonging to the *Pre- and Post-processing* unit are shown.

The *Serializer* unit buffers all input data coming from the *hardware-software communication* module and converts the buffered keys to bitstreams such that they can be processed by serial comparators. This module contains a register set and gets one key as input and buffers it in one of his empty registers. When all keys are buffered and all registers are full, the data is serialized and it is input into the sorting circuit. For this purpose, the module shifts all its registers with the keys one position to the left in every clock cycle until the registers are empty. The shifted output bits are then injected into the sorting circuit.

The *Deserializer* module also contains a register set and converts sorted output streams coming from the sorting circuit to output keys. After a sorting iteration, the module reads in every clock cycle the single bits from the sorting circuit and saves the significant bits in its registers. Then the registers are shifted one position to the left. If the sorted output streams are completely read, the module writes key after key into the output FIFO of the *hardware-software communication* module.

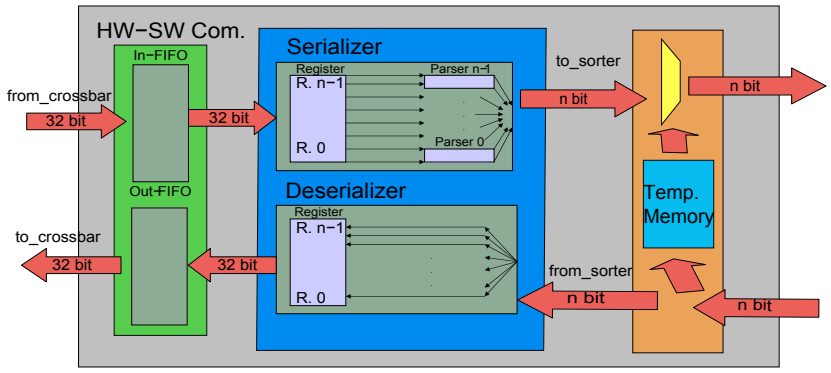


Fig. 6. On the left side, the *Serializer and Deserializer* units, on the right side, the *Load and Restore* module, which is only loaded when executing the dynamic Bitonic sorter.

In the dynamic Bitonic sorter case, the partial module called *Load and Restore* unit is loaded into the reconfigurable device. The module is settled between *de-/serialize* and the sorting circuit. Every time a sorting piece has been executed, the next partial sorting module is loaded into the reconfigurable region. Meanwhile the current sequence of the processed keys needs to be saved in memory. This is the first task of the *Load and Restore* unit. The second task is to input the current sequence of the keys into the next part of the sorting circuit as soon as the corresponding sorting module has been reconfigured. Additionally, this module also checks if the current sequence is already sorted. In this case, it forwards the final output of the sorting circuit to the *Serializer* unit in order to be transferred back to the control CPU. When the static Bitonic sorter is to be executed, this module is unnecessary and is used by the corresponding sorting circuit.

The interconnection of the sorting circuit is generated via the *HDL Generator for Sorting Circuits*. A crucial component of our sorting circuit is the implementation of the comparator, which sorts two keys. In this paper, word-serial comparators are used for area efficiency reasons. It handles two input keys as two bit streams and performs the compare-swap operations between bit pairs of the input keys. Beginning with the most significant bits of the input streams, it processes bit pair after bit pair in every clock cycle. Thus, a serial comparator needs w clocks to compare two w-bit keys. Using a word-parallel comparator would provide faster results, but also would consume significant FPGA resources at the same time. The resources of a word-parallel comparator scale linearly with the word length w of the compared keys. The implementation of the word-serial comparator so far uses 4 to 6 Look-Up Tables (LUT). It is independent of the word length of the input keys. Thus, its resource usage remains constant for any word length. The condition for correct usage of this word-serial comparator is an uninterrupted insertion of bit pairs of two input keys into the comparator in every clock cycle.

VII. Experimental Results

A. Static Bitonic Sorter

We implemented and tested static Bitonic sorters with input sequences of lengths from 1 to 300 of 32-bit integer keys. Our main FPGA consists of 33,792 slices. In our maximal implementation, 50% of the resources were consumed by the sorting circuit part, and the other half by the *Serializer* and *Deserializer* unit. The reconfigurable resource usage, the number of Slices, Look-up Tables (LUT), Flip Flops (FF), increased linearly with the number of input keys. The *Serializer* and *Deserializer* units took $n \cdot 53$ slices, $n \cdot 64$ LUTs, and $n \cdot 19$ FFs. The resource usage of the hardware-software communication part remained bounded by 405 slices, 507 LUTs, and 517 FFs. The experiments show that the theoretically expected sorting time is matched, when the communication cost between control CPU and reconfigurable hardware is neglected. The measured times meet the theoretical optimum of the sorting times: for example, the static Bitonic sorter with 128 inputs executes $\frac{1}{2} \cdot \log 128 \cdot (\log 128 + 1) = 28$ stages. A comparator performs its comparison in just one clock cycle and needs 20 ns to compare two bits under a 50 MHz clock. Thus, the sorting of 128 keys took $28 \cdot 20\,\text{ns} + 32 \cdot 20\,\text{ns} = 1200\,\text{ns}$, where the term $32 \cdot 20\,\text{ns}$ depends on the word length of the keys. This is exactly the measured running time of the Bitonic sorting process in hardware.

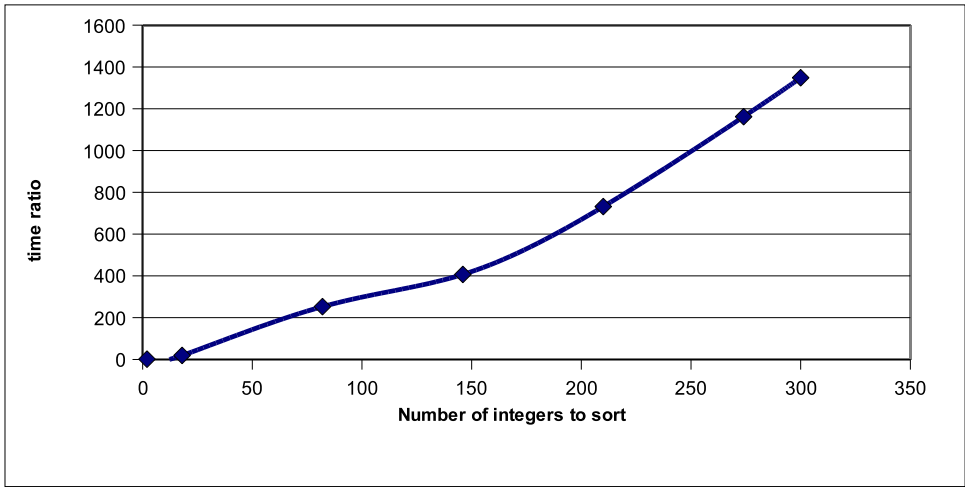


Fig. 7. Speedup factor by Bitonic sorting in hardware: Ratio between the running times of sequential merge sort on a CPU and the static Bitonic sorter in hardware (without the communication cost) depending on the number of input keys. The bullets on the line mark the measured scenarios.

Fig. 8. Running time of the static Bitonic sort (including communication costs) and the sequential merge sort on a CPU.

We also measured the running time of sequential merge sort on a 2.8 GHz CPU for the same number of input keys and calculated its average running time. The ratio of the two running times shows that Bitonic sort is several magnitudes faster than the sequential merge sort (see Fig. 7).

However, the cost of communication between control CPU and reconfigurable hardware is still too large on our experimental board, and also on currently available reconfigurable platforms in general. When the communication cost is not neglected, then the sequential merge sort on a 2.8 GHz CPU is still faster for sorting sequences of lengths 1 through 300 (see Fig. 8). But it can be estimated that if the input sequence has length 400 or more, the Bitonic sort would also be faster even when the communication cost between CPU and reconfigurable device are taken into account.

Thus, a communication channel with a higher bandwidth is needed to further increase the performance. This problem may be approached by designing a novel platform specialized for sorting purposes with adequate parallel data access. But for our purposes of testing the resource usage and usefulness of the Bitonic sorter and its adaptation to dynamically partially reconfigurable platforms, this communication bottleneck did not limit the goal of our approach. The experiments still validate that sorting can speedup sorting by several magnitudes.

B. Dynamic Bitonic Sorter

In the implementation of the dynamic Bitonic sorter, the size of the circuit realization is reduced enormously. In our implementation for sorting sequences of length 300, again 50% of the area is occupied by the *Serializer*, *Deserializer*, and *Load and Restore* unit, but now only 3% of the area are sufficient for the sorting circuit. In the static approach no more space was left, but in the dynamic approach 47% of the resources are now freely available. Thus, the main goal of our new approach was successful. The newly available space might be used by other application extensions or in order to sort longer sorting sequences.

Furthermore, BRAMs of the FPGA or extern memory of the ESM could be used for buffering the data during reconfiguration. Furthermore, a wider hardware-software communication buses could make *Serializer* and *Deserializer* modules unnecessary and thus also lead to the support of more inputs.

The sorting time is only twice the sorting time of the static Bitonic sorter. The reasons for this delay are the hardmacros in the partial regions because they store the data transferred between partial modules for one additional clock cycle in their flip-flops. Though the running time is doubled, it is still enormously faster than traditional approaches. However, the additional reconfiguration time is a bottleneck. Reconfiguration of only one partial module took 33 ms to 35 ms for one macro column, which takes almost as long as the whole sorting. It can be assumed that future dynamically reconfigurable platforms will tackle the reconfiguration bottleneck.

VIII. Conclusion

Sorting has manyfold applications (e. g., in database and computer graphics systems) and parallel sorting algorithms outperform traditional sequential sorting algorithms by orders of magnitudes. On the one side, parallel sorting algorithms are ideal for hardware realizations as they increase performance enormously. On the other side, their area usage increases also very fast with the number of inputs. In this paper, we propose a novel idea to overcome this drawback by adopting the parallel sorting algorithms to partially dynamically reconfigurable systems. We present a modified Bitonic sorter, a well-known parallel sorting method, such that not the complete circuit is loaded onto the reconfigurable device, but just smaller parts of it. Thus, not the complete sorting circuit must be loaded at once on the reconfigurable device or be implemented as a huge ASIC, but can be executed step by step. Furthermore, we also provide a detailed description of our design and implementation procedure. Our experimental results show two interesting results: First, the experiments show that our adopted sorting approach can be realized with partially dynamically reconfigurable platforms and shows an enormous benefit in sorting time. Subtracting the communication and reconfiguration costs, the sorting algorithms have a runtime that is in accordance with the theoretical analyses. New reconfigurable platforms are possible which provide much higher parallel access to input and output data and much lower reconfiguration times in a few clock cycles. Then, serializing and de-serializing steps would not take up so many reconfigurable resources and become unnecessary, and the total running time is speedup by several magnitudes. Second, the area usage of the sorting circuit is successfully reduced enormously, from 50% to 3% for input sequences of length 300. This is a great step forward, because much less space is sufficient for sorting the same input sequence. Thus, dynamic partial reconfiguration applied to sorting is a real benefit.

References

[1] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, pp. 770–785, 1988.

[2] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \cdot \log n$ parallel steps," *Combinatorica*, vol. 3, pp. 1–19, 1983.

[3] M. S. Paterson, "Improved sorting networks with $O(\log n)$ depth," *Algorithmica*, vol. 5, pp. 75–92, 1990.

[4] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Conf. Proc. 32*, 1968, pp. 307–314.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1998.

[6] C. D. Thompson, "The VLSI complexity of sorting," *IEEE Transactions on Computers*, vol. C-32, pp. 1171–1184, 1983.

[7] G. Bilardi and F. P. Preparata, "An architecture for bitonic sorting with optimal VLSI performance," *IEEE Transactions on Computers*, vol. C-33, pp. 646–651, 1984.

[8] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang, "Performance of sorting algorithms on the SRC 6 reconfigurable computer," in *Proc. IEEE Int. Conf. on Field-Programmable Technology (FPT)*, 2005, pp. 295–296.

[9] C. Layer and H. Pfleiderer, "A reconfigurable recurrent bitonic sorting network for concurrently accessible data," in *Proc. 14th Int. Conf. on Field-Programmable Logic and Applications (FPL'04)*, 2006, pp. 648–657.

[10] B. Zhang, Y. Xu, and M. Zhu, "A parallel sorting algorithm and its hardware implementation based on FPGA," *Journal of Information & Computational Science*, vol. 1, no. 3, pp. 417–422, 2004.

[11] R. Marcelino, H. Neto, and J. M. P. Cardoso, "Sorting units for FPGA-based embedded systems," in *Proc. 20th IFIP World Computer Congress, Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, 2008, pp. 11–22.

[12] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich, S. Fekete, and J. van der Veen, "The Erlangen Slot Machine: A highly flexible FPGA-based reconfigurable platform," in *IEEE Symp. on FPGAs and Custom Computing Machines (FCCM)*, 2005, pp. 319–320.

[13] J. Angermeier, D. Göhringer, M. Majer, J. Teich, S. P. Fekete, and J. van der Veen, "The Erlangen Slot Machine – A platform for interdisciplinary research in dynamically reconfigurable computing," *Information Technology*, vol. 49, pp. 143–148, 2007.

[14] D. Koch, C. Beckhoff, and J. Torrison, "Fine-grained partial runtime reconfiguration on Virtex-5 FPGAs," in *Proc. 18th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 69–72.

[15] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder – A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proc. 18th Int. Conf. on Field Programmable Logic and Applications (FPL'08)*, 2008, pp. 119–124.

[16] D. Koch, reCoBus homepage: http://recobus.de.

[17] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architecture, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proc. 16th Int. Conf. on Field Programmable Logic and Application (FPL'06)*, 2006, pp. 1–6.