Accelerating Sorting Through the Use of Reconfigurable Hardware

Ney L. V. Calazans, Fernando G. Moraes, Katherine B. Quintans, Felipe B. Neuwald

Hardware Design Support Group (GAPH) – Faculty of Informatics (FACIN) - PUCRS
E-mail: {calazans, quintans, moraes, fbn}@inf.pucrs.br

Abstract
In this paper we present the first steps of a work dedicated to explore the acceleration of sorting algorithms with reconfigurable hardware. The rationale for solving the sorting problem in hardware is presented. We suggest ways to facilitate the use of sorting hardware in the real world of applications programming. One of the ongoing work main goals is the migration of the well-known quicksort algorithm to hardware. Accordingly, we discuss the algorithm and provide its mapping to hardware.Since the hardware implementation of a basic sorting algorithm takes up only a few thousand gates, we also consider the hardware implementation of parallel sorting techniques as a way to further improve the efficiency of our approach.

1 Introduction
Ordering data according to some criterion is one of the most basic tasks using in processing information [Akl89]. Although this ordering may be a goal in itself in some cases, the main reason for ordering data is usually to enhance the performance of other processing tasks.

Let us take a simple example to illustrate the usefulness of ordering. Suppose that the results of a context involving thousands of candidates to fulfill some hundreds of positions have to be published. It is normal to publish alphabetically ordered listings containing the names of the candidates that succeeded, or listings where the candidates are ranked by their score. However, the candidates database is created during the inscription phase of the context, where the order of record generation is most probably a random one. Thus, a classification is to be done at some (or several) points of the data processing. Note also that the classification criteria can vary according to the goal of the processing. If the positions to be filled are all identical, an alphabetical ordering is enough to inform the candidates. However, if the position is depending on the score of the candidate a ranking by score has to be generated. Finally, let us note that the reason to generate the aforementioned ordered listings is no other than enhancing the processing of other tasks, in this case the visual searching of their names by each candidate.

The data classification problem is usually stated formally using a little bit of set theory. It is normal to use the designation sorting for the theoretical problem associated to classification.

Problem Statement (Sorting) – Suppose given $S = \{s_1, s_2, s_3, ..., s_n\}$, a set of $n$ elements and a sequence of $p$ elements $Q = \{t_1, t_2, t_3, ..., t_p\}$, where $n \leq p$, and each $t_i$ is an element $s_i$ of $S$, with any number of repetitions of element of $S$ allowed in $Q$. Suppose also that there is a total order binary relation $<$ defined over the set $S$. The sorting problem consists in generating a new sequence $S' = \{t_1', t_2', t_3', ..., t_p'\}$, where each $t_i'$ corresponds to exactly one element $t_j$ of the sequence $Q$, and additionally, the ordering $<$ holds for any pair of elements of $S'$, i.e. $t_i' < t_{i+1}'$ for $i = 1, 2, 3, ..., p-1$.

Since in most cases sorting is just a preprocessing step of complex computations, the efficiency of sorting algorithms is critical to many applications dealing with huge amounts of data, such as geographical databases or the computation of individual income taxes for whole countries. The last 40 years have seen an enormous research effort in developing, implementing and testing sorting algorithms in real life problems [Akl89]. Yet, every month new results are published suggesting enhancements in previously published sorting algorithms, or problem specific solutions that enhance performance of some real life applications needing to use sorting techniques.

The GAPH research Group at the Faculty of Informatics in the PUCRS is currently investigating strategies for accelerating sorting using reconfigurable hardware prototyping platforms. This work presents the views of the authors in this subject, together with some preliminary design data on implementing a sequential sorting algorithm in hardware.

This paper is organized as follows. In Section 2 we justify our approach and propose a framework for allowing the use of the developed hardware sorting mechanisms by means of ordinary programming techniques. Section 3 explores in detail one of the most efficient sequential sorting algorithms to date, called quicksort [Hoa62]. Next, Section 4 depicts the mapping of the sequential quicksort algorithm to hardware constructs. In Section 5 we describe the prototyping environment for our hardware implementation, presenting alternatives for future
versions of the same and other algorithms. Finally, Section 6 gives some preliminary conclusions and proposes a set of directions for future work based on them.

2 Rationale for Hardware Sorting

We start this Section discussing the generality of sorting algorithms, as well as factors that affect it. Next, we introduce some computational requirements for executing sorting. Finally, we depict our approach to implement and make available hardware implementations of sorting.

2.1 Generality of sorting

The formal sorting problem statement does not require any specific ordering criterion to be used while sorting data. This matches the real world, where we may have text that must be sorted in ascending or descending alphabetical order, or images whose pixels we are to sort in ascending or descending order of color wavelength. In practice, we may even need to use a hierarchy of sorting criteria. In this way, a sorting algorithm intended for general use should not depend upon a given total order binary relation. There are two commonly employed strategies to provide such generality:

1. Associate one or more integer keys with every single data to be sorted, so that the algorithm works upon the data keys only, assuming the natural order of integers as the total order relation $<$. Additionally, a binary information can be used to parameterize the algorithm for performing sort either in ascending or descending order.

2. Provide the user of sorting with means to externally specify a comparison procedure that, given two elements to sort, decides the relative ordering between them. The sorting algorithm is expected to call this procedure whenever needed. Although both strategies are widely used and the second is certainly the most flexible, we will assume herein the first one, since the second does not easily fit into hardware implementations. Furthermore, most real-life databases rely upon key indexing as the primary means to classify records into memory and disk files. An additional benefit of the first strategy is that the size of the key is not related to the size of the data to order, only to the number of records in the database. This tends to drastically reduce memory needs of sorting algorithms, without impairing performance.

2.2 Requirements of sorting

Sorting can be considered a memory hungry computational task, since for performance reasons the biggest possible amount of data records should reside in main memory during the algorithm execution. For example, it has been reported that the maximum amount of records that can be sorted into a 256Mbytes Unix workstation is around 1.5 million [Arp97]. This assumes a database indexed by a single integer key of 10 bytes and a pointer to the record on disk. The memory usage considers the operating system (OS) memory overhead, around 38-17% in practical cases. The exact OS overhead depends on the total memory, and was measured in machines with 64-256Mbytes, respectively.

Although the time taken to sort big databases is clearly dominated by disk I/O operations, some works report that around 20% of the time to run the sort algorithm is due to CPU time, when using a quicksort implementation [Nyb94]. Thus, sorting can in this case be seen as a processing intensive application. We note here that different values are found in recently enhanced algorithms (4% of CPU time and 96% of I/O time [Arp97]).

2.3 Proposed Approach

We believe that applications in need of high performance classification procedures can greatly benefit from hardware implementation of sorting algorithms. Also, we believe that the last statement is true only as long as these implementations are available to the ordinary programmer through some simple interface. This programming interface must hide implementation details, or better still, must hide even the fact that the sorting algorithm is actually running in hardware.

Some recent technological advances enable our approach.

First, there is a trend to standardize the communication between the main processor and peripherals in Personal Computers and/or UNIX Workstations. A good example is the PCI bus standard [Sol98]. Standardized buses serve as a primary means for communicating information among the main processor and peripherals. In our approach, one or more peripherals house hardware implementations of sorting, sharing access to resources with the processor, typically main memory and disks. The general architecture we propose is depicted in Figure 1.

A second technological advance that helps in making effective the use of hardware acceleration is the availability of low-cost reconfigurable hardware platforms with high memory capacity and high communication bandwidth with the main processor. Most of these platforms are built with SRAM-based Field-Programmable Gate Arrays (FPGAs), a recently introduced technology of increasing commercial success. A list of commercial reconfigurable platforms exists in http://www.optimagic.com/boards.html.
To give an idea of the cost-benefit compromises of these products, let us investigate one example platform. It is possible to acquire a board with capacity to simultaneously support 8 instances of our quicksort hardware implementation plus 16Mbytes of memory for data and a communication bandwidth with the main processor driven by a 32bit/33MHz PCI bus interface. The cost of such a platform is around US$850.00. More expensive available platforms allow multiplying hardware by a factor as big as 20, memory by as much as 16 and bandwidth by as much as 4.

These figures are to be compared with the cost of current systems used to break the barriers of sorting runtime, Symmetric Multiprocessors (SMP) and Network of Workstations (NOWs). In [Arp97], the authors investigate the sorting of huge amounts of data and compare the cost-benefit figures of several record-breaking systems, costing around US$1 million and sorting 6Gbytes in one minute. We intend to evaluate hardware sorting performance, as well as price-performance figures in order to compare our approach to these and other similar results reported in the literature.

![Architecture proposed to allow hardware acceleration of sorting algorithms in general-purpose programming.](image1)

Although we are dealing here with sorting algorithms only, our approach is expected to help migrating other commonly used algorithms to hardware as well. Typical examples are generic versions of selection, merging and searching algorithms [Akl89]. The long-term goal of our work is this field is the development of such algorithms in hardware, integrating implementations with a software library that can be used in ordinary programming. The library is expected to comprise the code to statically or dynamically configure an attached hardware platform to run any of these algorithms in hardware on demand. To us, static configuration means hardware configuration at program loading time, while dynamic configuration means runtime configuration, upon call of the algorithm. One immediate optimization of the dynamic scheme is verifying if hardware configuration is needed at all, due to the possibility of a previously configured hardware be already resident in the hardware platform. Although the second scheme is more interesting for the user, it is clearly more complex to implement due to the need to dynamically managing hardware configurations.

3 The Quicksort Algorithm

Quicksort, proposed by Hoare [Hoa62], is considered one of the most efficient existing sorting algorithms. It is based on the divide and conquer paradigm, as well as on sequencing by exchanging elements in the sequence under sorting [Sch96].

In the recursive version of the algorithm, described in [Cor90] and reproduced in Figure 2 in the form of pseudocode, the implementation is composed by two procedures, named `QuickSort` and `Partition`. The functionality consists in choosing a certain element, called pivot and partitioning the input vector of integer keys into two vectors. The first one contains all elements smaller than or equal to the pivot, while the second contains all elements greater than the pivot. This process is recursively repeated until the whole input vector is sorted.

```c
void QuickSort(int *v, int p, int r) {
    int q;
    if (p < r) {
        q = Partition(v,p,r);
        QuickSort(v,p,q);
        QuickSort(v,q+1,r);
    }
}
int Partition(int *v, int p, int r) {
    int pivot, i, j;
    pivot = v[p];
    i = p-1;
    j = r+1;
    while TRUE {
        repeat j = j-1 until v[j] <= pivot;
        repeat i = i+1 until v[i] >= pivot;
        if (i < j)
            exchange v[i] ↔ v[j];
        else
            return j;
    }
}
```

![Recursive version of quicksort.](image2)

In theory, the pivot can be chosen randomly. However, to accelerate sorting, the selection is done around the middle of the input vector. The heuristic here is to obtain a good balance between subvector sizes at each step, given a random distribution of values. A perfectly balanced set of subvectors potentially reduces the depth of the recursion tree. In the worst case, the pivot is, at each step, in one of the ends of the (sub)vector. In
this case the algorithm still works correctly [Sch96], although with degraded performance.

The behavioral dependence of the Partition function affects asymptotic performance in a significant way, since the worst case for it (totally unbalanced partitioning) determines a complexity of \(O(n^2)\) for quicksort with an input size of \(n\) elements. On the other hand the best case for the partitioning function dictates a complexity of \(O(n \log n)\). Fortunately, it can be demonstrated that the average case time (the most probable in practical problems) presents the same complexity as the best case and not of the worst case [Cor90].

Another sensitive aspect of quicksort arise when, for each call of Partition, instead of choosing \(v[p]\) as the pivot, one chooses \(v[r]\). In this case, if the initial order of the sequence to sort is such that at each recursive step \(v[r]\) is always the greatest element of the subvector, the algorithm degenerates and an eternal loop occurs [Cor90].

The recursive version of quicksort is the most intuitive one. However, if efficiency is sought, we have to avoid the overhead generated by recursive calls and the consequent context switching activities in software. To obtain higher performance, we may eliminate recursion totally, simulating it by the explicit use of stack structures and substituting recursion by iteration.

The non-recursive quicksort appears in Figure 3. This version works as follows: after the Partition function finishes execution, the actual parameters of QuickSort are no longer needed, except to compute the arguments for the following two recursive calls. Thus, instead of stacking the parameters implicitly at each recursive call, we can explicitly compute and stack the new actual parameters for each one of the two next calls of Partition. In this case, we create an explicit stack that contains the ends of each subvector that still need to be sorted. We stack the bigger subvector and process the smaller subvector immediately after iterating [Ten95]. The choice of processing smaller vectors first is a heuristic to reduce the needs for allocating memory for the stack.

A hardware quicksort requires the use of the non-recursive version as basis for the implementation. This occurs because no hardware mechanism is available for implementing recursion. Some hardware description languages such as VHDL do allow recursive descriptions. However, this resource can only be used for simulation purposes, not for automated synthesis. Synthesizable VHDL requires simulation of recursion by means of hardware stacks.

We have implemented, tested and profiled both recursive and non-recursive versions of the algorithm in software (using the C language). Our implementations were carried on Personal Computers (PCs), UNIX workstations and hardware prototyping platforms with a DSP microprocessor and FPGAs. Profiling was used to determine the processing bottlenecks of the software implementation with the goal of identifying what to migrate and how to migrate these bottlenecks to hardware. A behavioral VHDL version has also been implemented and validated using the Aldec Active-HDL simulator.

```c
#define MAXSTACK 128 /* Size of the stack */
void QuickSort(int *v, int tamanho){
    struct limits_type newlimits;
    struct stack_type stack;
    stack.top = -1;
    newlimits.left = 0;
    newlimits.right = tamanho-1;
    push(&stack, &newlimits);
    /* Repeat while there is some unsorted subvector on the stack */
    while(!empty(&stack)){
        pop(&stack, &newlimits);
        if(newlimits.left > newlimits.right){
            /* process the next subvector */
            j = Partition(v, newlimits.left, newlimits.right);
            if(j-newlimits.left > newlimits.right-j){
                /* put lower subvector in the stack */
                i = newlimits.right;
                newlimits.right = j-1;
                push(&stack, &newlimits);
               /* process lower subvector */
                newlimits.left = j+1;
                newlimits.right = i;
            }
            else {
                /* put upper subvector in the stack */
                newlimits.left = j+1;
                push(&stack, &newlimits);
               /* process the lower subvector */
                newlimits.left = i;
                newlimits.right = j-1;
            }/* end if statement */
        }/* end while statement */
    }/* end of QuickSort*/
}

Figure 3 – Non-recursive version of quicksort.

4 Quicksort Hardware Design

We have implemented a quicksort core in synthesizable VHDL, and we are currently in the final steps of its hardware validation on a prototyping platform. This Section describes the mapping of the sequential quicksort algorithm to hardware structure. The next Section will describe the adaptation of the quicksort core to a simple, standalone hardware prototyping platform based on Xilinx FPGAs.

4.1 Mapping of Quicksort to Hardware

Our implementation mixes behavioral and structural constructions of the VHDL language, in order to achieve the best possible synthesized hardware. The design is synchronous.
The quicksort core we implemented is depicted in Figure 4. It is divided into three main blocks: **Partition**, **Quick Datapath** and a finite state machine controller playing the role of a **Control Unit**.

The external ports of the design include a memory interface (at the right side of the core in Figure 4). It is composed by a chip enable and read-write pair of control signals and ordinary bi-directional data and unidirectional address buses. The memory stores both, the vector under sorting and the control stack. The rest of the control signals are the also conventional clock and reset signals, plus a signal to inform the external world the sorting has finished (the endq signal).

Internally, the pair **Control Unit** and **Quick Datapath** corresponds to the QuickSort procedure in Figure 3. Although we have tried several structural configurations for implementing this part of the design, its sequential structure leads to a datapath structure that is not much different from a conventional small-scale CISC processor [Mor99]:

- A set of five registers to keep the values of variables i and j, the right and left limits of the subvector under sorting and the stack pointer.
- A simple ALU and two subtractors as functional units.

The only significant structural difference of this part of the design is the fact that one line of the algorithm requires the simultaneous execution of three arithmetic operations (the if statement condition inside the most internal while statement in Figure 3). This parallelism is possible thanks to the extra subtractors.

The **Control Unit** is a finite state machine (FSM) with 36 states that depends upon three qualifiers coming from the **Quick Datapath** (signaling results of comparisons, used to take decisions regarding the control flow of the algorithm) and a signal informing the end of the (sub)vector partitioning task. The **Control Unit** generates the signal µinst, a microinstruction word to command the **Quick Datapath** block, together with a signal to trigger the activity of the **Partition** block (signal startp).

All arithmetic operations are executed among registers. Memory operations are limited to reading and writing a single data element at a time.

The functionality of the **Control Unit** is as follows. Upon reception of a reset signal the stack pointer is initialized with 0 (predecrementing will take it to the bottom of the memory in the first place). The array indexes are initialized (the newlimits structure) by reading the first position in memory, expected to contain the size of the vector to sort. The remaining states execute the algorithm in Figure 3 in a step by step fashion. The execution takes exactly one clock cycle for each step of the algorithm not operating with memory data, and two clock cycles otherwise. Note that the QuickSort block only manipulate vector indexes, data manipulation being the task of the **Partition** block. A call to the **Partition** block freezes the **Control Unit** and **Quick Datapath** blocks until the **Partition** block assigns the signal endp.

The **Partition** block implementation is also in the form of a datapath and a control FSM, as depicted
However, the structure is simpler and more amenable to parallel actions. The Datapath includes three data registers and two address registers.

Figure 5 – General structure of the partition block

The data registers are quite ordinary, with clock, reset and load enable inputs, while the address registers have an additional capability to perform either self-incrementing and self-decrementing of their stored data. The functional units are also three, one data comparator, one address comparator and an adder-shifter, also for address manipulation. Access to the vector in memory by the Partition block is granted when the startp signal is asserted by the quicksort Control Unit.

The Control FSM comprises 20 states, and depends upon qualifiers coming from the data (signal \(A \leq \neg B\)) and address (signal \(A \leq \neg B\)) comparators. It also depends on the already mentioned startp signal.

The algorithm implemented by the Partition block Control FSM is almost exactly the one described in Figure 2. Some differences occur in the treatment of limits of the vector, due to subtle distinctions between the recursive and non-recursive versions.

A partial simulation, showing the operation of the Partition block under control of the quicksort Control Unit appears in Figure 6.

The synthesis of our implementation, together with the external controller to be described in the next Section fit in a single 10,000 gates FPGA, consuming 80% of the logic resources of the device, and an estimated size of 6,000 equivalent logic gates (statistics furnished by the Xilinx Foundation Physical Synthesis System).

4.2 Some design decisions

An analysis of the quicksort algorithm reveals that in the worst case the stack can grow as much as the size of the vector under sorting, although this is a very rarely found situation in practice. Our first design decision was then to locate the stack in memory only, which has the disadvantage of slowing down the algorithm execution and taking up half of the memory space available. The advantage is on the generality of the implementation, which will never run out of
stack space. Better solutions can be devised, by a careful dimensioning of the stack (eventually inserting it in the core, if the stack is small enough) and an associated mechanism to allow the stack to grow as needed with a graceful decay in performance.

Implementing the stack inside the reconfigurable hardware would have a great impact on the performance of the algorithm, once roughly half of the memory accesses to addresses would be unnecessary. However, if the stack is big, this can greatly affect the availability of hardware resources inside the reconfigurable hardware.

The implemented quicksort core has been designed in order to allow the easy retargeting of the hardware to changing specifications of data and address width. By simply changing two lines in the original VHDL description and resynthesizing the hardware, a new working version is obtained. The size of the hardware version is limited only by available reconfigurable hardware resources, which can be scaled as well during synthesis, by simply choosing a device with more hardware resources.

5 Hardware Prototyping of Quicksort

To simplify the implementation and testing of the first version of the hardware quicksort, we have chosen a standalone hardware prototyping platform and design environment. The prototyping platform is a set of two boards, XS-40 and XStend, produced by the enterprise Xess, Inc. [http://www.xess.com/FPGA] It is an educational platform we employ largely in undergraduate courses [Cal99] and in research for small implementations.

5.1 Quicksort Interface to External World

The XS40 board we used contains 1 Xilinx XC4010E FPGA with 400 configurable logic blocks (supporting up to 10,000 equivalent logic gates), an Intel 80C51 microcontroller (not used in the current design), 32K of SRAM (expandable to 64 or 96Kbytes), a fixed 12MHZ clock and a 7-segment display. The Xstend board is basically an I/O extension to the XS40, containing 2 push-buttons, 8 dip-switches, 8 leds, 2 7-segment displays, plus external connectors for video, stereo audio and keyboard. Attaching the boards to a PC parallel port allows the communication with the host computer to take place.

The interface between the reconfigurable hardware and the memory consists in an 8-bit data bus and a 15-bit address bus, plus three controls signals (chip enable, write enable and output enable). With only 8 bits for data, the hardware is limited to work with small size keys, instead of the more commonly found multibyte keys [Arp97]. One solution to this problem is to multiplex data reading and writing, with a great impact on performance. To enhance performance, next implementations will make use of PCI hardware prototyping platforms, with parallel paths between the reconfigurable hardware and the processor-memory subsystem of either 32 or 64bits.

We have implemented a controller to adapt signals from the quicksort core to memory, and to provide feedback on the results of the algorithm execution. This feedback is provided through the limited I/O resources to input the vector, resetting the hardware and stepping through the results one
at a time (using the dip-witches and push-buttons) and displaying data and status information (through the 7-segment displays).

The address space of our implementation can be easily expanded to 15 and even to 16 bits (in case of the use of expanded memory versions of the board), but the external controller has to undertake significant changes before dealing with data more than 8 bits long. The general structure of the prototype system implemented to validate the quicksort core is illustrated in Figure 7.

6 Conclusions and Future Work

We have presented the first steps towards the implementation of sorting algorithms in hardware to allow their use in regular programming environments. The first implementation of quicksort was completely simulated and is now in the final phase of hardware timing debugging on a standalone prototyping platform. We now discuss some short and long-term works to be developed from now on.

Future processors and microprocessors are expected to include reconfigurable logic as part of its internal hardware, to allow greater flexibility and enhanced performance in solving specific problems [Wai97]. By addressing the implementation of hardware tightly coupled to the microprocessor system bus, we expect our approach to constitute a series of experiments on prototyping the internal interface between future in-chip configurable logic and the core processor.

Our experience showed that the quicksort algorithm is not the most adequate one for hardware implementations, since its nature is intrinsically sequential, offering little place to large-scale parallelism. We are currently starting to investigate parallel sorting algorithms [Aga96] and their hardware implementations.

We are currently acquiring a hardware prototyping platform that can be plugged in the PCI bus of PCs to test our quicksort hardware implementations in more realistic situations. This work will involve the development of device drivers and the use of PCI core hardware for high performance communication between the reconfigurable hardware and the main processor of host computers. The external controller will have to be completely redesigned, of course, but the quicksort core described in Section is not expected to change at all.

7 Acknowledgements

Ney Calazans gratefully acknowledges the support of the CNPq to this work through grant no. 520091/96-5, as well as from the FAPERGS through grant no. 96/1575.1. Fernando Moraes gratefully acknowledges the support of the CNPq through grant no. 522939/96-1.

8 References


