Synthesis of Parallel Sorting Networks using SAT Solvers

Andreas Morgenstern
University of Kaiserslautern
morgenstern@cs.uni-kl.de

Klaus Schneider
University of Kaiserslautern
schneider@cs.uni-kl.de

Abstract

Sorting networks are implementations of parallel sorting algorithms that have many important applications like routing where they are more powerful than other networks like the Omega-network [Law75]. Sorting networks are essentially combinational hardware circuits that consist of only compare/exchange modules. While a recursive definition of an asymptotically optimal sorting network is known [AKS83], its implementation turned out to be totally inefficient in practice. Optimal implementations of sorting networks with a minimal depth (and thus minimal computation time) are known only up to sizes up to 10 elements. Hence, the design of optimal parallel sorting networks is still a fundamental open problem for larger sizes. In this paper, we investigate whether and how it is possible to use the power of modern SAT solvers to automatically synthesize optimal sorting networks (with minimal depth). We were able to reproduce the known results up to size 10, but could so far not obtain larger optimal networks.

1. Introduction

Sorting is without doubt one of those fundamental algorithmic problems that have myriads of applications. Many classic textbooks on algorithms like [CLR90] therefore consider this problem in great detail and list many algorithms for this purpose. Typically, the authors restrict their consideration on sequential algorithms, which is today, however, no longer adequate since multicore processor architectures are now used even in the cheapest computers. For this reason, there is a growing interest in parallel sorting algorithms that have been already considered many years ago [Bat68, Col88, Cyp89, SS89, Pow91, Sed96].

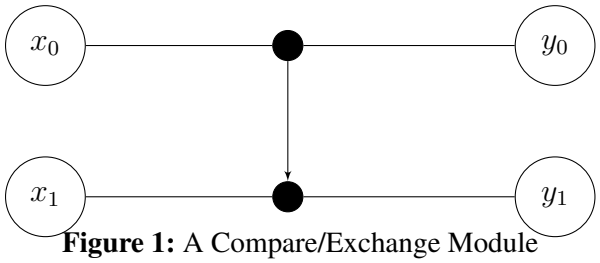

Figure 1: A Compare/Exchange Module

Sorting networks [CLR90, Sav98] turned out to be convenient mean to describe certain parallel sorting algorithms. A sorting network is thereby essentially a hardware circuit that maps the input values $x_0, \ldots, x_n$ that have to be sorted to its output ports $y_0, \ldots, y_n$. Sorting networks are

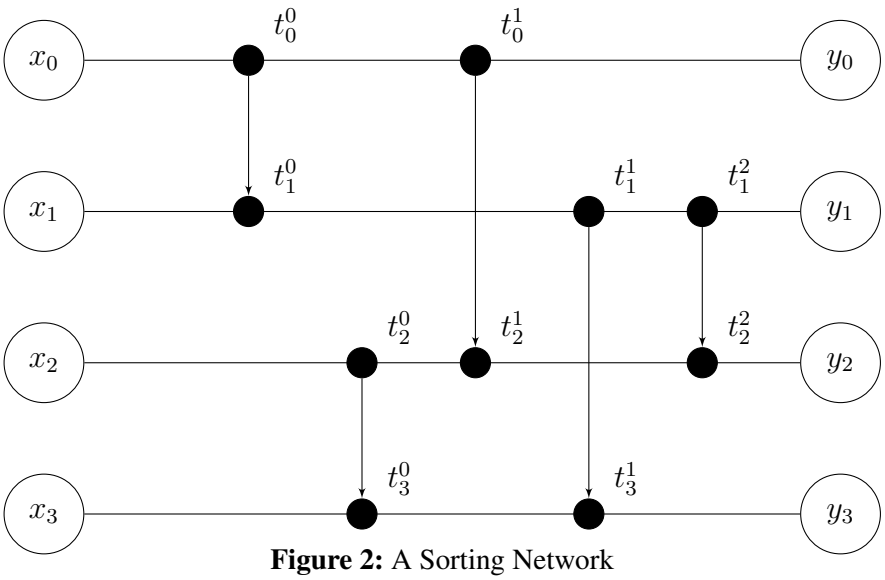

Figure 2: A Sorting Network

implemented only by connecting compare/exchange modules as shown in Figure 1. As can be seen, each compare/exchange module has two inputs x_0, x_1 and two outputs y_0, y_1. One of the outputs y_i is mapped to the maximum of the values x_0, x_1, while the other one is mapped to their minimum. The arrow in Figure 1 denotes the output which will result in the maximum, so we have $y_0 := \min\{x_0, x_1\}$ and $y_1 := \max\{x_0, x_1\}$ in Figure 1.

Sorting networks can therefore be used to implement sorting algorithms that are based on compare/exchange operations which singles out algorithms like BucketSort, RadixSort and Counting-Sort. They follow the idea that sorting can be done by a sequence of compare/exchange operations to determine the right places of the input values in the sorted output sequence. The nice property for describing parallel sorting algorithms is that the independence of some of the compare/exchange operations can be easily seen by the connections of the sorting network.

For example, a sorting network is given in Figure 2. First, the values x_0 and x_1 are compared which gives us the intermediate results $t_0^0 := \min\{x_0, x_1\}$ and $t_1^0 := \max\{x_0, x_1\}$. The other compare/exchange modules compute other intermediate results that are used to successively compute the final sorted sequence at the outputs $y_0, \ldots y_3$.

For implementations, regardless whether as combinational/sequential hardware circuits or as parallel software, one typically wants to either reduce the total number of required compare/exchange modules or the depth of the sorting network. As for general hardware circuits, the *depth* is thereby the length of the longest chain of compare/exchange modules whose outputs are forwarded as inputs to the next compare/exchange module in the chain. While the total number of compare/exchange modules used refers to the overall size of the hardware circuit, the depth is typically of more interest, since it directly corresponds with the computation/delay time of the network. For example, the sorting network shown in Figure 2 has five compare/exchange modules, and depth three. Thus, the delay of three compare/exchange modules is sufficient to compute the entire sorted sequence.

Obviously, parallel sorting networks naturally allow to schedule the compare/exchange modules in single computation steps that could be used for the design of sequential circuits as well. Of course, using ASAP (as-soon-as-possible) scheduling yields shortest computation times that

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Upper	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9
Lower	0	1	3	3	5	5	6	6	7	7	7	7	7	7	7	7

Figure 3: Upper and Lower bounds for optimal depth sorting networks

clearly correspond with the depth of the sorting network.

Most sorting algorithms are inherently sequential, but many parallel sorting algorithms are known as well. Among them are Batcher's BitonicSort [Sav98, Bat68], Batcher's OddEvenSort [Bat68], the AKS network [AKS83, Pat87a], Cole's parallel MergeSort [Col88], a parallel version of the variant of ShellSort that is obtained by using BubbleSort for sorting the gap sequences (which is sometimes called CombSort) [Cyp89, Sed96], ShearSort [SS89], and also parallel versions of QuickSort [Pow91]. For the implementation of sorting networks, algorithms that make dynamic choices like the computation of partitions of QuickSort, are not well-suited. For this reason, also only some of the parallel sorting algorithms can be used for the implementation of sorting networks. The most important ones are Batcher's BitonicSort [Sav98, Bat68] and Batcher's OddEvenSort [Bat68]. These sorting networks have a depth of size $O((\log n)^2)$ since they implement MergeSort with depth $O(\log n)$ sequential merge steps where each merge step requires time $O(\log n)$. While quite useful for practical implementations, these parallel sorting networks are not optimal. A sorting network of the same depth $O((\log n)^2)$ is obtained by a variant of ShellSort [Sed96]. The asymptotically best known sorting network, the AKS network [AKS83, Pat87b] (named after their inventors) achieves depth $O(\log n)$ and size $O(n \log n)$ for n inputs, and reaches thereby the lower bound that is also reached by other parallel sorting algorithms like parallel QuickSort [Pow91]. It can be shown that no sorting network with a depth less than $O(\log n)$ can exist, so that the AKS network is asymptotically optimal. However, while theoretically optimal, the AKS network has little practical application because of a big constant hidden in the O-notation. It has been reported that it improves Batcher's bitonic sorter only for astronomically large sequence sizes. For this reason, there is still an interest in the design of parallel sorting networks that improve the known results both in theory and in practice.

The construction of optimal parallel sorting networks for fixed sizes of sequences is also still an open problem, since the depths obtained by the generic algorithms like Batcher's sorters can also be improved for particular sizes. For sequences up to size 10, optimal parallel sorting networks are known [Par89], and up to size 16, lower and upper bounds are known as well (see Figure 3 taken from [Par89]).

In this paper, we investigate whether SAT-Solvers can be used to automatically synthesize optimal circuits. In particular, we wish to automatically synthesize optimal parallel sorting networks with optimal depths. To this end, we show how to encode all sorting networks of a particular depth as a propositional formula that is then checked by a publicly available SAT/SMT-solver (Yices). Any model, i. e. any truth assignment to the variables that satisfies the formula encodes then a correct parallel sorting network of the specified constant depth.

Using the standard SAT/SMT-solver Yices, we were able to reproduce the known results up to size 10 with an acceptable runtime. At the time writing, our experiments continue for larger sizes, and we hope to achieve results that were not known so far. But even if that should not be true at the end, we could demonstrate that the optimal sorting networks that were obtained in [Par89] have been computed with special algorithms on supercomputers (Cray-2), while we use a standard

SMT-solver and standard computers. We therefore want to emphasize that SAT/SMT-solvers are not only useful tools for verification, but also for synthesis.

The outline of the paper is as follows: in the next section, we present our encoding of sorting networks of a fixed depth by means of a propositional formula. For the synthesis of the particular formulas of desired sizes, we make use of the synthesis framework Averest, so that we describe the problem with the programming language Quartz, as explained at the end of Section 2. In Section 3, we then explain the use of the SAT/SMT solver Yices for computing the results that we list in Section 4.

2. Synthesis of Sorting Networks as a SAT Problem

In this section, we show how to encode a sorting network with a given depth d and a given length l of input sequences by a propositional formula. Any satisfying assignment to variables of this formula is then a correct sorting network of depth d and sequences of length l.

In order to obtain this propositional formula, we use the synthesis framework Averest to obtain a combinational circuit (or equivalently a propositional formula) from an initial description of the problem as a program in the Quartz language [Sch09]. Hence, instead of directly formulating the problem as a SAT-problem, we use some capabilities of our toolset to obtain a circuit from a remarkably simple parametrized program. Although we used the Averest toolset, similar results could be obtained with any other hardware specification language like VHDL or Verilog.

2.1. The Zero-One Principle

Before starting the description of the mentioned formula that encodes parallel sorting networks of depth d and a given length l, we note that we make use of the well-known zero-one principle [Knu98, CLR90]. This principle essentially says that we can restrict our consideration to the sorting of boolean lists:

Theorem 1 ([Knu98, CLR90])**.** *If a sorting network correctly sorts every sequence of 0's and 1's, then it sorts every arbitrary sequence of values.*

Thus, without loss of generality, we can concentrate on sequences with boolean values only. Thanks to the zero-one principle, this is no restriction. The sorting networks for boolean inputs that we automatically obtain by a SAT solver are also correct for sequences of k-bit integers (for arbitrary k).

2.2. Representing Sorting Networks with a Permutation Matrix

We can represent any sorting network with a permutation matrix, i. e. a two-dimensional array P as follows: if there is a compare/exchange module that connects wires i and j in depth k, then and only then we have $P_{k,i} = j$ and $P_{k,j} = i$. If a wire is not fed as input to a compare/exchange module, then we set $P_{k,i} = i$. Thus, the sorting network of Figure 2 would be represented by the permutation matrix given in Figure 4, when we assume an ASAP scheduling of the compare/exchange operations. Note that in depth 2, lines 0 and 3 are not connected to other lines.

The permutation matrix will be written as a Quartz program with a $d \times l$ array with values from $0 \ldots l-1$ which will be one part of the input of our circuit.

i	$P_{0,i}$	$P_{1,i}$	$P_{2,i}$
0	1	2	0
1	0	3	2
2	3	0	1
3	2	1	3

Figure 4: Permutation matrix for the sorting network of Figure 2

2.3. Representing Compare/Exchange Modules

By the definition of the permutation matrix, wire i is the output of a compare/exchange module iff for some j we have $P_{k,i} = j$ and $P_{k,j} = i$, i.e. iff $P_{k,P_{k,i}} = i$. In this case, the value of wire i is obtained by the following formula[1]:

$$\bigwedge_{i=0}^{l-1} C_{0,i} = \begin{cases} \min\{x_i, x_{P_{0,i}}\} & \text{if } i \leq P_{0,i} \\ \max\{x_i, x_{P_{0,i}}\} & \text{else} \end{cases} \tag{1}$$

$$\bigwedge_{k=1}^{d-1} \bigwedge_{i=0}^{l-1} C_{k,i} = \begin{cases} \min\{C_{k-1,i}, C_{k-1,P_{k,i}}\} & \text{if } i \leq P_{k,i} \\ \max\{C_{k-1,i}, C_{k-1,P_{k,i}}\} & \text{else} \end{cases} \tag{2}$$

or equivalently, if we remember that the inputs are boolean values, and hence, that also the results are boolean values:

$$\bigwedge_{i=0}^{l-1} C_{0,i} = \begin{cases} x_i \wedge x_{P_{0,i}} & \text{if } i \leq P_{0,i} \\ x_i \vee x_{P_{0,i}} & \text{else} \end{cases} \tag{3}$$

$$\bigwedge_{k=1}^{d-1} \bigwedge_{i=0}^{l-1} C_{k,i} = \begin{cases} C_{k-1,i} \wedge C_{k-1,P_{k,i}} & \text{if } i \leq P_{k,i} \\ C_{k-1,i} \vee C_{k-1,P_{k,i}} & \text{else} \end{cases} \tag{4}$$

2.4. Specifying that the Input is a Valid Permutation Matrix

We have to make sure that $P_{k,i}$ encodes a sorting network regardless whether that sorting network is correct. Note that by our encoding, in each column k, each wire i is either connected with some other wire j or with itself (which means that it is not the input of a compare/exchange module). Thus, we only have to consider inputs $P_{k,i}$ where the following holds

$$\bigwedge_{k=0}^{d-1} \bigwedge_{i=0}^{l-1} P_{k,P_{k,i}} = i \tag{5}$$

2.5. Specifying that the Result is sorted

Specifying that the result of the network is a sorted sequence can be rather easily done. To this end, recall that our inputs are all boolean values. Thus, the result is sorted whenever for some index i,

[1] We use here the standard convention that the minimum of two values will be outputted to the lower of the two indices.

the element at index i is 1, then all elements at indices $j \geq i$ must also be 1, or equivalently written as a propositional formula (note that $C_{d-1,i}$ is output wire i):

$$\bigwedge_{i=0}^{l-2} C_{d-1,i} \rightarrow C_{d-1,i+1} \tag{6}$$

2.6. Representing Sorting Network Checking as a Quartz Program

Finally, we describe how we can implement a Quartz program that checks whether a given $d \times l$ matrix `P` represents a correct sorting network. Clearly, we could run that Quartz program for a particular input matrix `P` to check that particular matrix `P`. However, what is more interesting, we can make use of formal verification methods of our toolset Averest that can translate Quartz programs and their specifications to model-checking problems. As we are interested in the synthesis of sorting networks, we specify that for fixed chosen parameters d and l, no input matrix `P` represents a correct sorting network. Hence, any counterexample to the model-checking problem is a correct sorting network. This is done by the Quartz program shown in Figure 5.

The module expects a $d \times l$ matrix `P` as input (marked by `?`) and generates two boolean outputs `noperm` and `unsorted`. The body of the module consists of a large for-loop that is executed in one macro step, so that the module can be translated to a combinational circuit and its verification problem becomes a SAT problem. Within this macro step, the outermost for-loop enumerates all possible l-bit bitvectors `x`. To this end, the unsigned integer `x` obtains all values from 0 up to 2^{l-1}. In each iteration of the outermost for-loop, we declare a local array `input` that will contain the input sequence that corresponds to `x` that has to be sorted. Moreover, we declare a local array `C` that holds the values of the wires of the sorting network for this input. For this reason, the first for-loop assigns `input[i] = nat2bv(x,l())i`, so that `x` is converted to a l-bit bitvector and its i-th bit is stored in `input[i]`.

The next two for-loops implement the behavior of the sorting network as discussed in Section 2.3. Recall that a matrix is a valid permutation matrix iff we have `P[k][P[k][i]] == i` for all `i` and `k`. Thus, we also check in these two for-loops if this is true, and emit output `noperm` if `P` is not a valid permutation.

Finally, we check whether the local array `input` generated for the current value of `x` has been sorted. To this end, we check formula 6 and emit the output `unsorted` iff it is violated. Since the entire for-loop is executed in one step, output `unsorted` *is true iff there is one input sequence* `x` *that is not correctly sorted by* `P`. Let us therefore note that `unsorted` $\Leftrightarrow \exists$`x`.$\neg$sorted(`x`) holds, where sorted has the intended meaning.

Our Quartz module is correct iff for all inputs `P`, the assertion `!noperm -> unsorted` given at the end of the program is satisfied. Equivalently, the module will not be correct, iff there is a $d \times l$ matrix `P` where the assertion `!noperm -> unsorted` is not satisfied. In this case, `!noperm & !unsorted` is true which means that `!noperm & !`$\exists$`x`.$\neg$sorted(`x`), i.e., `!noperm &` $\forall$`x`.sorted(`x`) holds. This means that our Quartz module is not correct, iff there is a $d \times l$ matrix `P` that is a valid permutation matrix that correctly sorts all input sequences.

Hence, we reduced the synthesis of correct $d \times l$ sorting networks to the correctness of our Quartz module, and each counterexample corresponds with a correct $d \times l$ sorting network. Note that the use of a programming language like Quartz that allows a hardware synthesis is now of

```
macro d() = 5;  // depth of sorting network
macro l() = 5;  // size of sequence to be sorted

module SortingNetwork([d()][l()]nat{l()} ?P,bool noperm,unsorted) {
    // enumerate all possible 2^l() boolean input sequences
    // and assign them to the local boolean array input
    // that is sorted using the intermediate local values C[k][i]
    // with the same permutation matrix P
    for(x=0..exp(2,l())-1) {    // enumerate all inputs
        event [l()]bool input;  // local input array obtained by bitvector x
        event [d()][l()]bool C; // intermediate value of wires for this input

        // generate local input sequence to be sorted
        for(i=0..l()-1)
            input[i] = nat2bv(x,l()){i};

        // compute values of wires C[0][i]
        for(i=0..l()-1)
            if(P[0][P[0][i]]==i) // i and P[0][i] are connected in depth 0
                if(i<=P[0][i])
                    // C[0][i] is min of input[i] and input[P[0][i]]
                    C[0][i] = input[i] & input[P[0][i]];
                else
                    // C[0][i] is max of input[i] and input[P[0][i]]
                    C[0][i] = input[i] | input[P[0][i]];
            else emit(noperm);

        for(k=1..d()-1)
            for(i=0..l()-1)
                if(P[k][P[k][i]]==i) // i and P[k][i] are connected in depth k
                    if(i<=P[k][i])
                        // C[k][i] is min of C[k-1][i] and C[k-1][P[k][i]]
                        C[k][i] = C[k-1][i] & C[k-1][P[k][i]];
                    else
                        // C[k][i] is max of C[k-1][i] and C[k-1][P[k][i]]
                        C[k][i] = C[k-1][i] | C[k-1][P[k][i]];
                else emit(noperm);

        // check whether input x has been sorted by P
        if(!forall(i=0..l()-2) (C[d()-1][i] -> C[d()-1][i+1]))
            emit(unsorted);
        }
    // specification to assure that P is a correct sorting network
    assert(!noperm -> unsorted);
}
```

Figure 5: Synthesis of Sorting Networks Represented as SAT Problem in Quartz

n	2	3	4	5	6	7	8	9	10
depth	1	3	3	5	5	6	6	7	7
time	0m0s	0m0s	0m0s	0m1s	0m4s	1m20s	14m32s	15h50m	21d20h

Figure 6: Runtimes of Yices to Synthesize Optimal Sorting Networks

essential importance, since all that remains to do is to translate the module to a corresponding combinational hardware circuit so that modern SAT solvers can be used to check its correctness. Due to our reduction, the SAT solver will generate a correct $d \times l$ sorting network if one exists for the chosen parameters d, l.

3. Experimental Results

We used the SMT solver Yices, version 1.0.28 as our final solver to decide the correctness of our Quartz module for particular instances of the parameters d, l. Although Yices is an SMT solver, we only used the propositional part of Yices. Whenever Yices found a model for the formula, this model is a solution to our problem, namely a permutation matrix, or equivalently an encoding of a correct sorting network.

All experiments have been performed on a 3.0 GHz QuadCore Pentium Duo with 16 GB of RAM. Yices needed only a small amount of the available memory, for larger sizes and depths, not more than 1 GB was required. The runtime Yices needed to synthesize a sorting network with optimal depth for certain lengths l are given in Figure 6 (listing runtime in days (d) , hours (h), minutes (m) and seconds (s)).

Hence, as can be seen, the runtime of Yices to automatically synthesize optimal sorting networks is reasonable up to size 9. For size 10 for which optimal bounds are also known, the runtime was more than 21 days. However, while the runtimes explode, this does not hold for the memory consumption. While writing this paper, our server is still running to determine an optimal bound for size 11 while still consuming approximately 1 GB of memory which is the same memory consumption which was used for input size 10. Hence we still hope that it can be completed in a reasonable amount of time.

The correctness of our synthesis procedure can be easily checked by feeding the resulting matrix `P` as input to our Quartz module for simulation. The simulator will then check whether `P` will correctly sort all sequences. This way, it is easy to check the correctness of the synthesized results, even though they must be correct if the Quartz compiler and the used SAT solver are correct.

Due to lack of space, we can not draw the results for all synthesized networks. For example, an optimal network obtained for $l = 5$ and depth $d = 5$ is shown in Figure 7.

Of course, we reduced d as far as possible, to determine the lowest depth for a given l. Thus, we ran Yices on some instances that are not satisfiable. In these cases, the runtimes of Yices on the unsatisfiable instances were comparable to the satisfiable instances.

4. Conclusions

In this paper, we have shown that state-of-the-art SAT solvers are able to automatically synthesize optimal hardware circuits. In particular, we demonstrated this by the example of sorting networks that we encoded as propositional formulas for fixed lengths l of the input sequences and fixed

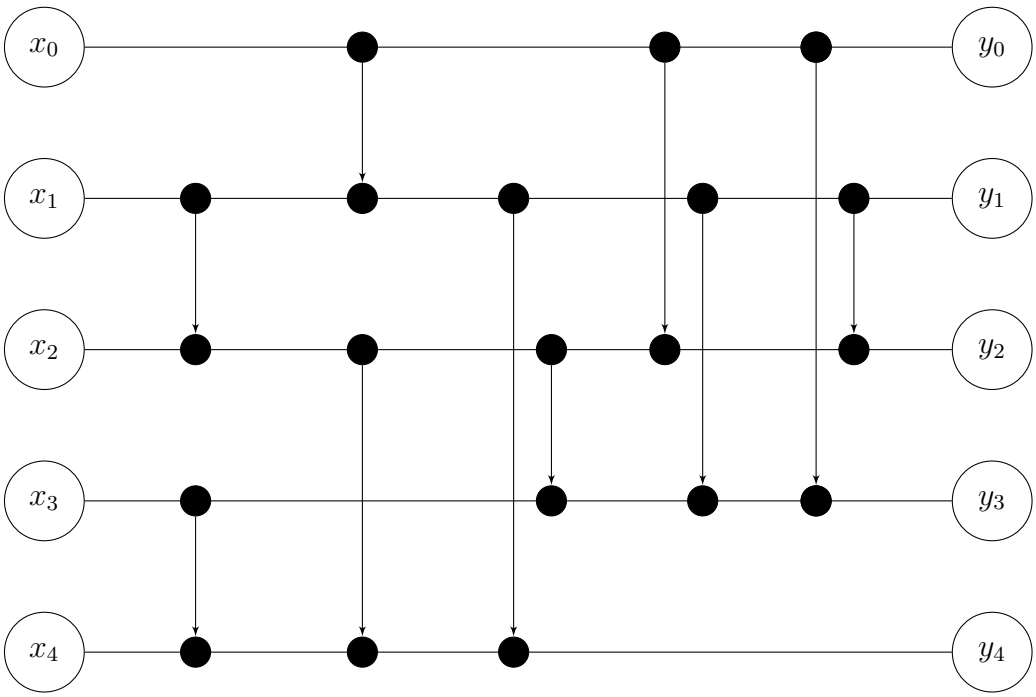

Figure 7: The Synthesized Sorting Network for length 5 and depth 5

depths d of the sorting network. The SAT solver can then either check that there is no sorting network with these parameters (since d is below the lower bound) or it can produce a provably correct parallel sorting network for parameters l and d. Using the least d for a given l yields an optimal sorting network. To this end, we encoded the problem by means of a Quartz program that was synthesized to a combinational hardware circuit. Using this circuit, we generated a (quantified) propositional formula as an input to a SAT solver. We successfully generated sorting networks for input sizes of up to 10 which matched the known optimal bounds. Experiments with larger input sizes are currently running, and we hope to report on them at the workshop.

References

[AKS83] Ajtai, M., J. Komlos, and E. Szemeredi: *An $o(n \log(n))$ sorting network*. In *Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1983.

[Bat68] Batcher, K.E.: *Sorting networks and their applications*. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[CLR90] Cormen, T.H., C.E. Leiserson, and R.L. Rivest: *Introduction to Algorithms*. MIT Press, 1990.

[Col88] Cole, R.: *Parallel merge sort*. SIAM Journal on Computing, 17(4):770–785, August 1988.

[Cyp89] Cypher, R.: *A lower bound on the size of shellsort networks*. In Leighton, F.T. (editor): *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 58–63, Santa Fe, New Mexico, USA, 1989. ACM.

[Knu98] Knuth, D.E.: *The Art of Computer Programming*, volume 2. Addison-Wesley, 1998.

[Law75] Lawrie, D.H.: *Access and alignment of data in an array processor*. IEEE Transactions on Computers (T-C), 24:1145–1155, December 1975.

[Par89] Parberry, I.: *A computer assisted optimal depth lower bound for sorting networks with nine inputs*. In *Supercomputing*, pages 152–161, Reno, Nevada, USA, 1989. ACM. http://doi.acm.org/10.1145/76263.76280.

[Pat87a] Paterson, M.S.: *Improved sorting networks with O(log(N)) depth*. Algorithmica, 5(1):75–92, 1987.

[Pat87b] Paterson, M.S.: *Improved sorting networks with $o(\log(n))$ depth*. Technical report CS-RR-089, University of Warwick, Conventry, UK, 1987.

[Pow91] Powers, D.M.W.: *Parallelized quicksort and radixsort with optimal speedup*. In Mirenkov, N.N. (editor): *Parallel Computing Technologies*, pages 167–176, Singapore, 1991. World Scientific.

[Sav98] Savage, J.E.: *Models of Computation - Exploring the Power of Computing*. Addison Wesley, 3rd edition, 1998.

[Sch09] Schneider, K.: *The synchronous programming language Quartz*. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[Sed96] Sedgewick, R.: *Analysis of shellsort and related algorithms*. In Díaz, J. and M.J. Serna (editors): *European Symposium on Algorithms*, volume 1136 of *LNCS*, pages 1–11, London, United Kingdom, 1996. Springer.

[SS89] Scherson, I.D. and S. Sen: *Parallel sorting in two-dimensional VLSI models of computation*. IEEE Transactions on Computers, 38(2):238–249, 1989.