EPiC Series in Computing

Volume XXX, 2024, Pages 1–15

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Minimizing Sorting Networks at the Sub-Comparator Level

Luís Cruz-Filipe and Peter Schneider-Kamp

Dept. Mathematics and Computer Science
University of Southern Denmark
{lcf,petersk}@imada.sdu.dk

Abstract

Sorting networks are sorting algorithms that execute a sequence of operations independently of the input. Since they can be implemented directly as circuits, sorting networks are easy to implement in hardware – but they are also used often in software to improve performance of base cases of standard recursive sorting algorithms. For this purpose, they are translated into machine-code instructions in a systematic way.

Recently, a deep-learning system discovered better implementations than previously known of some sorting networks with up to 8 inputs. In this article, we show that all these examples are instances of a general pattern whereby some instructions are removed. We show that this removal can be done when a particular set of constraints on integers is satisfiable, and identify conditions where we can reduce this problem to propositional satisfiability. We systematically apply this general construction to improve the best-known implementations of sorting networks of size up to 128, which are the ones most commonly found in software implementations.

1 Introduction

Sorting networks are data-oblivious algorithms to sort a fixed number of inputs using only a single type of gate (the comparator), which compares two elements and sorts them. Differently from the common software algorithms, where the number of instructions executed depends on the actual input being sorted, sorting networks prescribe in advance which elements should be compared – meaning that, for some inputs, they will perform unnecessary comparisons. This rigid structure makes sorting networks amenable to highly efficient hardware and software implementations, where the cost of the extra operations is dwarfed by the speed at which they can be performed.

Finding efficient sorting networks has been a challenging research topic since the 1960s. Lower bounds on the minimal required number of comparators $S(n)$ for a sorting network with n inputs are mostly established by analysing the space of all possible combinations of a fixed number k of comparators and showing that none of them form a sorting network. Progress in this direction has been slow, and driven mostly by improvements in hardware or groundbreaking insights on how to explore symmetries of the problem to reduce the size of the search space. The only general theoretical result dates to the 1970s [28].

General constructions to build sorting networks are an extreme illustration of the difference between theoretical results and practice. The asymptotically best known construction requires

$O(n \log(n))$ comparators [1], but it requires an unfeasible amount of comparators for any reasonable number of inputs. By contrast, the systematic construction proposed by Batcher [2] requires $O(n^2)$ comparators, but yields usable networks for all values of n of practical interest.

While sorting networks are typically referred to using hardware concepts such as "circuits" and "gates", they are also increasingly used in software to deal with base cases of general sorting algorithms efficiently – one very prominent example is the implementation of Quicksort in the GNU C library, which resort to hard-coded sorting networks when the number of elements to be sorted falls below a fixed threshold. These implementations typically implement comparators as sequences of instructions; they can be significantly optimized by taking into account the low-level parallelism currently available in virtually all processors and concrete information about e.g. cache effects [11], as well as wide SIMD vector instructions available in state-of-the-art processors [29].

Very recently, computer experiments using deep learning showed potential for further optimization of these implementations [23] by identifying redundant instructions in the result of applying the de-facto standard translation of a comparator as a sequence of 4 branching-free instructions. In this article, we analyze the results of these experiments and show that the optimizations reported can all be formulated as instances of a general principle. This yields a systematic way to optimize any sorting network directly, by solving a constraint satisfaction problem for each comparator. We show that we can use an off-the-shelf SMT solver to decide these constraints, and further optimize the process by using a SAT solver to rule out a significant percentage of candidates for elimination.

Contribution. This article updates the state-of-the-art in efficient implementations of sorting networks, introducing a way forward to optimize the implementation of sorting networks by identifying redundant instructions at the sub-comparator level. We begin by systematizing the discovery from [23] and show how this systematization can be scaled from 8 up to 128 inputs and beyond, covering all cases relevant in practice. We show that a variant of the the well-established 0-1 principle (a comparator network that sorts all binary sequences also sorts any sequence of numbers) also holds for our systematization. Finally, we provide empirical evidence for the effectiveness and efficiency of our systematization.

Structure of the paper. We start by reviewing the basic concepts and history of sorting networks in Section 2, as well as fixing the notation used throughout this presentation. Section 3 focuses on the optimization described in [23], extending its applicability from networks of up to 8 inputs to networks of up to 32 inputs. In Section 4, we show how to scale the minimization even further to networks of 128 inputs and beyond, covering the practically important input sizes of 16–128 typically used in the implementation of base cases of general sorting algorithms such as Quicksort. We empirically evaluate our systematized minimization in Section 5 by means of extensive experiments. Finally, in Section 6, we consider a potential relaxation of the conditions of the systematized minimization, demonstrating an efficient and effective strategy for computing minimization in the relaxed setting, which however does not provide further benefits for sorting networks of up to 64 inputs. We conclude in Section 7 and outline directions for future research.

2 Background and related work

Sorting networks have been studied from the 1960s, with the most comprehensive reference on the topic being the dedicated chapter in [21]. Sorting networks are also interesting for practical

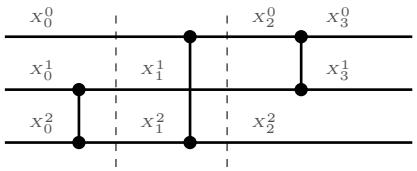

Figure 1: Sorting networks on 3 inputs X_0^0, X_0^1, and X_0^2, where X_ℓ^i and X_ℓ^j are the outputs of the ℓ-th comparator (i, j) and $X_3^0 \leq X_3^1 \leq X_2^2$ are the outputs.

applications [2, 12], in particular in the area of signal processing [5, 6, 20]. There are several directions of research in the topic, and we briefly summarize the most important contributions.

Figure 1 introduces the notation we use in this article by illustrating a sorting network that sorts a sequence of three inputs X_0^0, X_0^1, and X_0^2 into a sorted sequence of three outputs $X_3^0 \leq X_3^1 \leq X_2^2$ through the application of three comparators. The ℓ-th comparator (i, j) takes as input the current values on channel i (i.e., the value X_m^i for the maximum $m < \ell$) and channel j (i.e., the value X_n^j for the maximum $n < \ell$). By comparing and swapping, if necessary, it produces outputs $X_\ell^i = min(X_m^i, X_n^j)$ and $X_\ell^j = max(X_m^i, X_n^j)$ on channels i and j, respectively. We say that i is the top channel of comparator j while j is its bottom channel.

On the theoretical side, the two most studied problems deal with optimality: determining the minimum *size* (number of comparators) $S(n)$ and the minimum *depth* (number of *layers* of independent comparators that can run in parallel) $T(n)$ of a sorting network on n inputs. Surprisingly, finding exact values of $S(n)$ and $T(n)$ is extremely difficult, essentially due to the relative lack of general theoretical results: all known upper bounds are obtained by concrete sorting networks, while lower bounds are typically established by brute-force analysis of the search space of all candidate sorting networks of a given size or depth, combined with some clever pruning techniques. Progress on lower bounds is closely tied to new insights that help reducing this immense search space or advances in computational power. Contributing to the complexity of the problem is a result from 1990 [8] that states that even the problem of determining whether a particular configuration of comparators is a sorting network on n inputs requires testing nearly all binary inputs of length n.

Finding good sorting networks is not trivial, as there are few intuitions about how they work. Many of the best known sorting networks were found by trial-and-error, or by optimizing previously discovered ones [16, 18]. Systematic, recursive constructions can be combined with these to generate usable sorting networks for any n. Batcher's odd-even construction [2] and Parberry's pairwise sorting [26] work especially well when n is a power of 2, while Coles' construction [13] is optimized for the case when n is a perfect square. An interesting example arises from restricting comparators to work only on adjacent values; networks of these *miniswaps* [15] can in particular implement the well-known bubble sort and insertion sort algorithms.

All the above constructions yield sorting networks of size $S(n) = O(n^2)$, well above the theoretical $O(n \log n)$ achieved by many sorting algorithms. The AKS construction [1] yields networks of size $O(n \log n)$, showing that sorting networks can also achieve optimal speed; however, the constants hidden in the big-O notation are so huge that these networks are unusable in practice. More recently, several authors have also experimented with genetic algorithms to improve on known sorting networks since the 1990s [7, 22, 27], and SAT-solvers have been used to generate sorting networks from scratch [24].

The exact values of $S(n)$ for $n \leq 8$ were known already in the 1960s [16], where the best known sorting networks for these numbers of inputs were shown to be optimal. Lower bounds

for $S(3)$ and $S(5)$ were determined by brute force, with some symmetry arguments to reduce the number of subcases in the latter. The value of $S(7)$ was established by a computer program using similar arguments, and is one of the oldest computer proofs in record. The value of $S(4)$ follows from a mathematical argument that does not generalize to any other value of n, while the values of $S(6)$ and $S(8)$ can be obtained by applying a theoretical result first published in [28].

It took 50 years for additional progress to be made on the size optimality problem, when the value of $S(9)$ was established by two independent computer programs from the same authors [10]. The method combines a generation phase with a pruning phase based on the notion of *filter* [13] – a sequence of comparators that can be extended to an optimal-size sorting network – and an adaptation of ideas previously used to tackle the optimal-depth problem [4]. The authors later verified their results using a certified checker obtained from a Coq formalization of the problem [14]. A further extension of the same ideas established that the known 35-comparator sorting network for 11 inputs was also size optimal [19]. Both works applied the theoretical result from [28] to establish also the exact values for $S(10)$ and $S(12)$.

The related problem of computing $T(n)$ has proven to be slightly more tractable. While the exact values of $T(n)$ were again known for $n \leq 8$ already in the 1960s [21], the next breakthrough happened in 1989 when Parberry observed that the first layer of an optimal sorting network can be assumed to be fixed [25]. This observation made the problem tractable with the computing power available at the time, and determined the values of $T(9)$ and $T(10)$. Bundala and Zavodny [3, 4] later extended these ideas to two-layer filters, and were able to generate a small enough set that, combined with an encoding of the problem in SAT, established lower bounds for $T(n)$ with $11 \leq n \leq 16$ coinciding with the previously best-known sorting networks. A similar analysis of the last layers yielded the value of $T(17)$, while at the same time improving the best known networks for 19 and 20 inputs [9].

While the number of comparators and the number of layers suffice to describe the complexity of sorting networks implemented in hardware, software implementations are sensitive to the actual instructions used to encode individual comparators. Modern architectures use instruction-level parallelism, which can be exploited by suitably sorting the comparators to improve the actual performance [17]. Additionally, when n is not too small, cache effects also start being relevant, allowing for even more optimizations [11]. The starting point for our work is a recent development using deep learning, which showed that the standard implementation can be optimized by removing some instructions that can be proven to be redundant [23].

3 Systematizing the optimization

In this section, we first describe the standard implementation of a sorting network, and analyze and systematize the minimization procedure from [23]. Then, we show how to implement the systematic construction as a forward pass and how to reallocate registers subsequently.

3.1 Systematization

Optimal sorting networks can be modularly translated to executable machine code by translating each individual comparator and concatenating the result. Assuming that the inputs of comparator (i, j) on channels i and j are stored in registers r_i and r_j, respectively, we can translate it into three instructions, corresponding to the standard implementation of a conditional swap:

1. copy the value in register r_i to a new register r_k;

2. if the value in register r_i is greater than or equal to that in register r_j, then copy the value from r_j into r_k;

3. if the value in register r_i is greater than or equal to that in register r_j, then copy the value from r_i into r_j.

This implementation has the disadvantage that it uses two conditionals. With the availability of conditional move instructions for most CPU architectures, a conditional swap is nowadays more commonly implemented instead by four (branching free) machine code instructions:

1. MOV r_i r_k

2. CMP r_i r_j

3. CMOVGE r_j r_k

4. CMOVGE r_i r_j

Here, MOV r_x r_y signifies that the value of r_x is assigned to r_y, CMP r_x r_y that the comparison flags are set accordingly to the values of r_x and r_y, and CMOVGE r_x r_y that the value of r_x is assigned to r_y if the comparison flags of the preceding CMP instruction indicate a greater than or equal.

AlphaDev [23] is a machine-learning system that was trained using deep reinforcement learning to discover implementations of sorting networks. An analysis of the solutions that it found shows that they all coincide with the straightforward translation described above, with one potential optimization: in some cases, the initial assignment for a comparator (i, j) is skipped. This assignment is indeed redundant if the auxiliary register r_k already contains the correct final value (i.e., the value of r_i) in case the swap is not executed; otherwise its value is irrelevant, as it is overwritten with the value of r_j. Furthermore, the analysis also reveals that, in all these cases, the auxiliary register k is the register that was used in the previous comparison on the top channel i.

As an example, consider once more the sorting network on three inputs from Figure 1. Here, for the third comparator, we need to check whether the value on channel 0 can be reused, potentially saving one instruction for implementing the comparator. Figure 2 illustrates this situation, where we have to show that $X_0^0 = X_2^0$ in the case where the comparator does not perform the swap, i.e., assuming that $X_2^0 < X_1^1$. Furthermore, the preceding comparators yield some constraints on the possible values of these variables, including but not limited to the fact that the top channel output of the second comparator is the minimum of its two inputs ($X_2^0 = min(X_0^0, X_1^2)$) and that the outputs of the first comparator are sorted ($X_1^1 \leq X_1^2$). In conclusion, for this example, we have to show that $X_2^0 < X_1^1 \wedge X_1^1 \leq X_1^2 \wedge X_2^0 = min(X_0^0, X_1^2) \rightarrow X_0^0 = X_2^0$. To prove this manually, we can simply observe that the first two inequalities imply that $X_2^0 < X_1^2$, which immediately implies the conclusion.

This observation suggests a systematic optimization method for implementations of sorting networks: for each comparator, check whether the register that was last used on its top channel already contains the right value, and in the affirmative case reuse it. This check can be performed by writing it as a constraint satisfaction problem (CSP) in integers, which can be handled by any state-of-the-art SMT solver supporting inequalities over integers.

At first sight, it is tempting to detect the pattern in Figure 2 and other examples from AlphaDev [23] by a fixed pattern matching. This, however, would prevent us from realizing the full potential of this optimization for larger sorting networks. Indeed, the network in Figure 3 shows an example with only five inputs where the removal of an assignment instruction relies

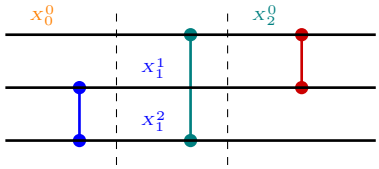

Figure 2: A sorting network for three inputs, where the comparators and channel values crucial to optimizing the third comparator are highlighted. The constraint for assignment removal is $X_2^0 < X_1^1 \wedge X_1^1 \leq X_1^2 \wedge X_2^0 = min(X_0^0, X_1^2) \rightarrow X_0^0 = X_2^0$.

on five comparators and includes both *min* and *max* equalities. This motivates us to develop a more general approach, which we describe in the next section.

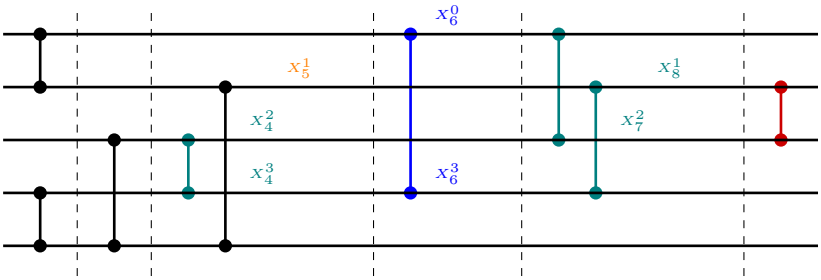

Figure 3: A sorting network for five inputs with comparators and channel values crucial to optimizing the ninth comparator highlighted. The constraint for assignment removal is $X_8^1 < X_7^2 \wedge X_4^2 = min(X_3^2, X_2^3) \wedge X_4^3 = max(X_3^2, X_2^3) \wedge X_6^3 = max(X_1^0, X_4^3) \wedge X_6^0 \leq X_6^3 \wedge X_7^2 = max(X_6^0, X_4^2) \wedge X_8^1 = min(X_5^1, X_6^3) \rightarrow X_5^1 = min(X_8^1, X_7^2)$.

3.2 Implementation

We implemented the systematic construction described in the previous section as a forward pass through the sorting network, represented as a list of comparators. This allows us to collect all the constraints from the prefix of the network that might be relevant in determining whether one of the inputs of a previous comparator can be reused. In particular, the ℓ-th comparator (i, j), contributes with the constraints $X_\ell^i = min(X_m^i, X_n^j)$ and $X_\ell^j = max(X_m^i, X_n^j)$ for maximum $m, n < \ell$, as well as $X_\ell^i \leq X_\ell^j$.

As we cannot predict whether the register storing the output of a comparator will be reused by a later comparator, we initially use new registers for the outputs of the comparators. This implies that for a network N of $k = |N|$ comparators on n inputs, the number of registers used is $n + k$ rather than $n + 1$. Thus, for the shortest known network for 14 inputs with 51 comparators, even a CPU with 64 general purpose registers would have to store some of the values in main memory.

To minimize the number of registers needed, we perform a live register analysis following standard ideas of live variable analysis. We start with the output registers holding the results of the comparator network as the rearmost live set. In a backward pass, for each comparator, we compute the preceding live set by first removing the output register from the current live set and then adding the inputs to it.

Based on these live sets, in a forward pass, we substitute fresh output registers (i.e., ones that have not been reused using our systematic construction) with variables that have been freed. Here, the set of freed variables is collected by considering the set difference between the current live set and the succeeding live set. During the forward pass, we keep a substitution that maps the originally assigned registers to the reallocated registers, replacing all occurrences of substituted registers in the program on the fly to avoid another forward pass.

(Liveness analysis and register reallocation are standard components of optimizing C compilers. In principle, we could opt to emit C code [11] using C variables instead of registers and rely on the compiler. While we implemented and tested this approach, we opted for our own liveness analysis and reallocation implementation for the presentation of our results in order to reliably benchmark and quantify the impact of our systematized optimization on register usage.)

The resulting reallocated program uses a number of registers of at least $n + 1$ and surely less than $n + k$. In practice, this means that a CPU with 32 general purpose registers is fully sufficient for the shortest known network for 21 inputs, while a CPU with 64 general purpose registers is sufficient for the shortest known network for 55 inputs.

4 Scaling the optimization

The bottleneck in our systematic construction for optimizing implementations of sorting networks lies in the use of an SMT solver to decide whether a given inequality holds given a set of constraints. In this section, we first show how to reduce the size of the generated CSPs by a simple technique for networking slicing. Then, we show how to reduce the CSPs to SAT problems, further improving the efficiency and, thereby, scalability of our systematized optimization.

4.1 Slicing the network

The examples in Figures 2 and 3 show that not all comparators contribute with constraints crucial to proving that an assignment may safely be removed. While we currently know of no way to predict accurately which constraints are needed, we can reduce their number by identifying comparators for which we can prove that they only generate *irrelevant* constraints, in the sense that they are sure not to contribute with constraints crucial for the proof.

The main intuition behind this identification is that we can view a comparator network as a program, and apply principles of program slicing [30]. Concretely, this amounts to starting from the ℓ-th comparator for which we want to check the applicability of our optimization and performing a backward pass through the network. Initially, the set of channel values to consider are the inputs to the ℓ-th comparator, and the set of potentially relevant comparators contains only the ℓ-th comparator. During a backward pass, we collect the inputs of all comparators that produce one of the channel values in our set, adding the comparator to the set of potentially relevant comparators. At the end of the backward pass, this set is the *network slice* relative to the ℓ-th comparator.

Figure 4 presents the results of applying network slicing relative to the seventh comparator (marked in red) to the sorting network from Figure 3. While the two comparators following the seventh comparator are naturally excluded from the slice, the fifth comparator is also excluded because its outputs are not used as inputs to either the sixth or seventh comparator. For larger comparator networks of size S and for $\ell \ll S$, network slicing significantly reduces the number of constraints generated and, thus, the size of the CSPs to solve.

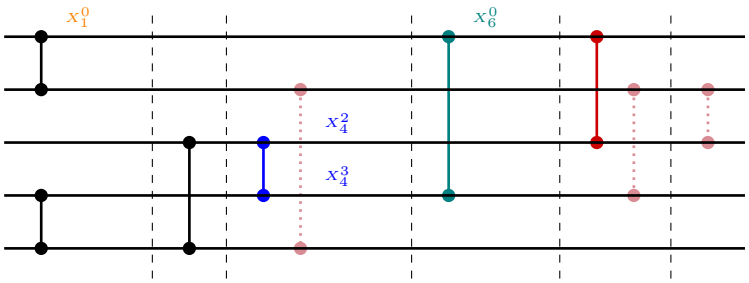

Figure 4: The sorting network for five inputs from Figure 3 with comparators and channel values crucial to optimizing the seventh comparator highlighted. Comparators irrelevant to this optimization are drawn with pink dotted lines.

4.2 Reducing to SAT

Network slicing reduces the size and solving time of CSPs on average. However, it only has a moderate effect on the runtime for the largest instances, i.e., when attempting to apply our optimization to the ℓ-th comparator for ℓ close to the size of the sorting network. In these cases, network slicing only removes a comparatively small number of comparators, effectively limiting the applicability of our systematized optimization to e.g. 64 inputs. To scale the applicability of our systematized optimization further, we show how we can make a first analysis replacing CSPs and SMT solvers with more efficient SAT problems and SAT solvers.

Concretely, consider the optimization problem associated with a comparator (i, j). To show that we can remove the assignment instruction $r_k = r_i$ safely, we must show that the state of r_k after the next sequence of comparisons remains the same. The relevant case is when no conditional assignments are made, and we need to show that $r_k = r_i$ holds (otherwise the value of r_k is overwritten, and is therefore immaterial).

The part of the program executed up to this point gives us a set of constraints of the form $r_m \leq r_n$ and $r_\ell = min(r_m, r_n)$, where ℓ, m, n are natural numbers. Furthermore, we also assume that r_k was used in the previous comparison involving the lower input to the comparator, so this set includes a constraint of the form $r_i = min(r_k, r')$ for some register r'.

Finally, since we only need to consider the case when the condition of the assignments is false, we can assume an additional constraint $r_i < r_j$. Denoting the set of all these constraints by Γ, our problem is showing that $\Gamma \models r_k = r_i$ when all registers are assigned integer values.

While this CSP is amenable to SMT solving, as the size of the network grows, the problem quickly becomes intractable. The following lemma shows that we can reduce the CSP to a SAT problem, which has lower complexity and will be tractable for all practical applications.

Lemma 1. *Suppose that* $\Gamma \not\models r_k = r_i$. *Then there is a counterexample where* $r_\ell \in \{0, 1\}$ *for all* ℓ.

Proof. Assume that $\Gamma \not\models r_k = r_i$. Then there is an assignment mapping each variable r_ℓ to an integer a_ℓ such that all constraints in Γ hold and $a_k \neq a_i$. From the constraint $r_i = min(r_k, r')$, it must be the case that $a_i \leq a_k$. Together with $a_k \neq a_i$, we thus have $a_i < a_k$.

We now define a new assignment by mapping r_ℓ to 0 if $a_\ell \leq a_i$ and to 1 otherwise. Clearly this assignment falsifies $r_k = r_i$; we now show that it satisfies all constraints in Γ.

- If the constraint is of the form $r_m \leq r_n$, this means that $a_m \leq a_n$. If $a_n \leq a_i$, then both r_m and r_n are mapped to 0, and the constraint holds; likewise, if $a_m > a_i$ then both

r_m and r_n are mapped to 1, and the constraint holds. The only remaining case is that $a_m \leq a_i$ and $a_n > a_i$, where r_m is mapped to 0 and r_n to 1, and the constraint again holds.

- If the constraint is of the form $r_\ell = min(r_m, r_n)$, we observe that a_ℓ must be equal to either a_m or a_n. If both $a_m, a_n \leq a_i$, then both r_m and r_n (and therefore r_ℓ) are mapped to 0, and the constraint holds, while if $a_m, a_n > a_i$ then r_m, r_n, and r_ℓ are all mapped to 1 and the constraint again holds. If $a_m \leq a_i$ and $a_n > a_i$, then $a_\ell = a_m$, and again r_m and r_ℓ are mapped to 0, while r_n is mapped to 1, and the constraint holds. The argument when $a_n \leq a_i$ and $a_m > a_i$ is similar.

- If the constraint is the additional constraint $r_i < r_j$, then r_j is assigned a value $a_j > a_i$ in the original assignment, meaning that it is now assigned to 1, while r_i is assigned to 0, so the constraint holds. □

As a consequence, if the SAT solver returns UNSAT, we can conclude that the initial assignment can be removed from the implementation of comparator (i, j). The converse is trivially true, since any counterexample over $\{0, 1\}$ is also a counterexample over the whole domain of integers.

5 Empirical evaluation

The systematic construction presented in this articles was implemented as a proof-of-concept using the Python programming language. Our implementation is able to produce machine code instructions and C functions. Furthermore, for each successful removal of an assignment instruction, a visualization of the sorting network and the involved comparators and channel values in the style of Figures 2 to 4 can be output in LaTeX format. The source code and log data underlying the results reported on in this section can be found in a GitHub repository: https://github.com/schneiderkamplab/snopt

All experiments reported on in this section were performed on a server running Ubtunu 22.04 with a 5.15.0 Linux kernel. The server is equipped with an AMD EPYC 7501 32-Core Processor able to run 64 threads in parallel at 2.0 GHz. We used Python 3.10.13 with the 4.12.4.0 z3-solver, the 0.4.1 nnf, and the 0.1.8.dev12 python-sat modules.

5.1 Comparing to the state of the art (2 to 8 inputs)

Table 1 compares how many instructions can be removed by AlphaDev [23] and our systematized optimization, respectively. Our systematization provides the same number of saved instructions for all numbers of inputs except for 5 and 7.

For 5 inputs, the sorting algorithm found by AlphaDev reuses one of the registers assigned when copying values from the memmory. This optimization is currently not implemented, since our implementation currently only allows the reuse of registers assigned as outputs of comparators. There is no reason to believe that our implementation could not be amended to also consider the reuse of registers assigned for inputs.

For 7 inputs, our systematized optimization already finds 2 further instructions that can be removed, demonstrating the inherent incompleteness of AlphaDev's approach, even for relative small numbers of inputs.

#inputs	#comp.	naive	AlphaDev [23]		this article	
			#instr.	#saved	#instr.	#saved
2	1	**8**	**8**	0	**8**	0
3	3	18	**17**	1	**17**	1
4	5	**28**	**28**	0	**28**	0
5	9	46	**42**	4	43	3
6	12	60	**57**	3	**57**	3
7	16	78	76	2	**74**	4
8	19	92	**91**	1	**91**	1

Table 1: Comparison of the results of AlphaDev [23] and our systematized optimization on the best known sorting networks for 2 to 8 inputs.

5.2 Going beyond the state of the art (9 to 128 inputs)

To test our approach more systematically, we applied our systematic optimization to the sorting networks on 9–128 inputs generated by Batcher's systematic construction [2]. Figure 5 visualizes the number of instructions that could be removed for each number of inputs, and shows that the former grows roughly linearly with the latter.

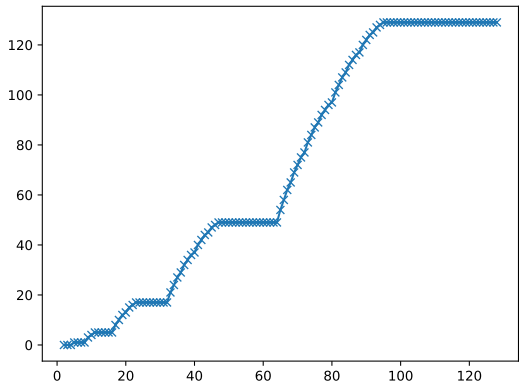

Figure 5: Number of instructions that could be removed for the sorting networks on 2 to 128 inputs obtained using Batcher's construction.

The step-like appearance of the increase is likely grounded in particulars of Batcher's construction, which relies heavily on powers of 2. Indeed, the "jumps" in the plot occur when the number of inputs is only slightly larger than close to powers of 2.

We also experimented with other systematic constructions, finding larger possibilities for runtime reduction. That said, the total number of instructions after reduction was always favouring Batcher's construction, as it involved fewer comparators to start with.

5.3 Impact of the optimization on register use

Figure 6 visualizes the number of additional registers needed for storing values when using the systematized optimization. Clearly, there is a small but considerable overhead in register use associated with our optimization. Thus, it is prudent to experiment a bit in order to

find an optimal balance between the decreased number of instructions and the availability of adequately-sized general purpose registers.

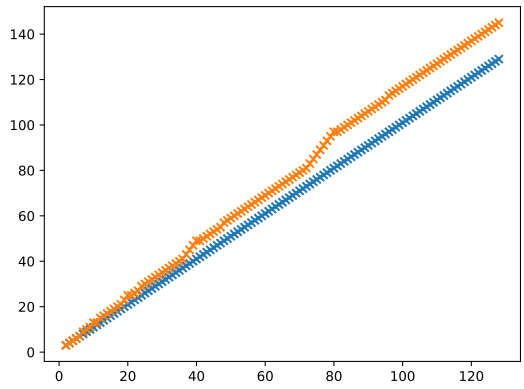

Figure 6: Number of registers used for a given number of inputs. The blue line represents the number of registers used in the unoptimized setting, i.e., $n + 1$ for n inputs. The orange line represents the number of registers used with the systematized optimization after register reallocation. For this figure, we used sorting networks of 2 to 128 inputs using Batcher's construction [2].

5.4 Network slicing for scaling the optimization

Figure 7 depicts the improvement in runtime when applying network slicing, as introduced in Section 4.1. We observe that runtime is reduced by approximately one-third when using this analysis. While this result demonstrates the utility of network slicing, this first scaling technique is obviously not essential to be able to apply our systematized optimization.

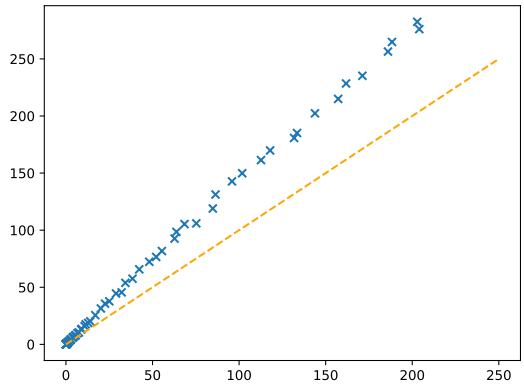

Figure 7: Scatter plot of runtimes (in seconds) when using slicing on the (linear) x-axis and the without slicing on the (linear) y-axis. For this figure, we used sorting networks of 2 to 64 inputs using Batcher's construction [2].

In our implementation, we also experimented with limiting the size of the slices, i.e., we

fixed the number of comparators that are included in the slice for the ℓ-th comparator to $\lfloor \frac{\ell}{k} \rfloor$ for some k. Experimenting with $k \in \{2, 4, 8, 16, 32\}$, we found that $k = 8$ provides a further runtime reduction of one-fifth without compromising completeness for up to 128 inputs. For $k \in \{16, 32\}$ we observe runtime reductions around two-fifths, but fail to identify one-third and two-thirds of the assignment instructions, respectively.

5.5 SAT solving for scaling the optimization

Figure 8 visualizes the improvement in runtime when using a SAT solver instead of an SMT solver, capitalizing on the results presented in Section 4.2. For larger sorting networks, we observe that the backend based on SAT is at least a decimal order of magnitude faster than the backend based on SMT. With the relative improvement increasing for larger instances, this second scaling technique is clearly essential for being able to apply our optimization for sorting networks of 128 inputs and beyond.

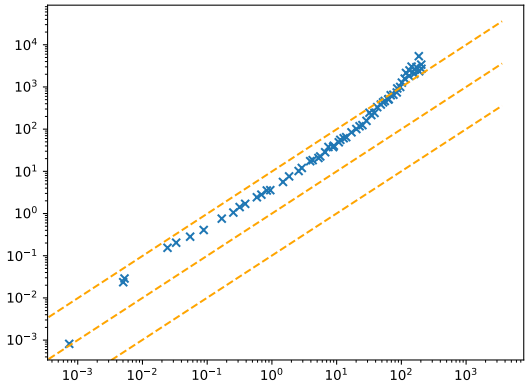

Figure 8: Scatter plot of runtimes (in seconds) for the optimization in seconds with the SAT backend on the (logarithmic) x-axis and the SMT backend on the (logarithmic) y-axis. The orange diagonals represent (from top to bottom) a 1-order-of-magnitude difference favoring SAT, equality between SAT and SMT, and a 1 order-of-magnitude difference favoring SMT. For this figure, we used sorting networks of 2 to 64 inputs using Batcher's construction [2].

6 Further optimizations

Our reduction to SAT relies on the fact that we are checking the auxiliary register lastly used for a comparison on the lower input to the current comparator. Without this restriction, Lemma 1 no longer holds – it is easy to find sets of constraints of the form we consider that have counterexamples over the integers but not over $\{0, 1\}$.

We reran our experiments for up to 64 inputs to see if removing this restriction would lead to further assignments being removed. The strategy was:

- For each available register, use SAT to check whether there is a counterexample to $\Gamma \models r_i = r_j$ over $\{0, 1\}$. If this is the case, this register cannot be reused.
- If there is no counterexample, check whether Γ contains a constraint $r_i = min(r_j, r')$ for some r'. If this is the case, then this register can be reused (and the assignment removed).

- Otherwise, run an SMT solver to check whether there is a counterexample to $\Gamma \models r_i = r_j$ over the integers. If this is the case, this register cannot be reused, otherwise it can be reused and the assignment removed.

In all the tests we ran, no additional assignments were removed. Our tests showed that the last step was executed often, and every time it disagreed with the result of the SAT solver. These results show two things: on the one hand, "false" removals flagged by SAT do occur in practice; on the other hand, however, the only register that works as a replacement for a fresh one is the one identified by analyzing the results of AlphaDev [23] – in which case Lemma 1 applies. We do not know if this is a general phenomenon, or whether it is due to most sorting networks tested being instances of systematic constructions. We plan to investigate this matter further in future work.

7 Conclusions and future work

In this article, we have shown that implementations of sorting networks can be optimized by identifying redundant instructions at the sub-comparator level, systematizing the very recent deep learning-assisted discovery from AlphaDev [23]. We introduce network slicing and prove a variant of the 0-1 principle, allowing us to scale our optimization from 8 up to 128 inputs and beyond. As a consequence, we cover all cases relevant in practice such as when using sorting networks as base cases for general sorting algorithms such as Quicksort. In practice for a base case size of 32, this means that for each invocation, at least 17 machine instructions can be saved. These savings accumulate to a significant runtime and energy reduction for sorting larger sequences.

Future research should investigate whether different sorting networks yielded by different constructions, non-systematic generation methods such as genetic algorithms and/or comparator reordering exhibit differences in their sub-comparator level optimization potential and register usage. Furthermore, the relation between our optimization as presented in this article and a dual case relying on bottom instead of top channel outputs might be investigated, with the most interesting question revolving around whether these dual optimization might be combined to further improve sub-comparator level efficiency. Last but certainly not least, there are theoretical and practical issues to consider regarding implementation-optimal vs size-/depth-optimal sorting networks.

References

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Procs. STOC*, pages 1–9. ACM, 1983.

[2] Kenneth E. Batcher. Sorting networks and their applications. In *Procs. AFIPS*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[3] Daniel Bundala, Michael Codish, Luís Cruz-Filipe, Peter Schneider-Kamp, and Jakub Závodný. Optimal-depth sorting networks. *J. Comput. Syst. Sci.*, 84:185–204, 2017.

[4] Daniel Bundala and Jakub Zavodny. Optimal sorting networks. In Adrian-Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Procs. LATA*, volume 8370 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2014.

[5] Chaitali Chakrabarti. Sorting network based architectures for median filters. *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, 40(11):723–727, 1993.

[6] Chaitali Chakrabarti and Li-Yu Wang. Novel sorting network-based architectures for rank order filters. *IEEE Trans. Very Large Scale Integr. Syst.*, 2(4):502–507, 1994.

[7] Sung-Soon Choi and Byung Ro Moon. Isomorphism, normalization, and A genetic algorithm for sorting network optimization. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant G. Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *Procs. GECCO*, pages 327–334. Morgan Kaufmann, 2002.

[8] Moon-Jung Chung and Bala Ravikumar. Bounds on the size of test sets for sorting and related networks. *Discret. Math.*, 81(1):1–9, 1990.

[9] Michael Codish, Luís Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp. Sorting networks: To the end and back again. *J. Comput. Syst. Sci.*, 104:184–201, 2019.

[10] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *J. Comput. Syst. Sci.*, 82(3):551–563, 2016.

[11] Michael Codish, Luís Cruz-Filipe, Markus Nebel, and Peter Schneider-Kamp. Optimizing sorting algorithms by using sorting networks. *Formal Aspects Comput.*, 29(3):559–579, 2017.

[12] Richard Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987.

[13] Drue Coles. Efficient filters for the simulated evolution of small sorting networks. In Terence Soule and Jason H. Moore, editors, *Procs. GECCO*, pages 593–600. ACM, 2012.

[14] Luís Cruz-Filipe, Kim S. Larsen, and Peter Schneider-Kamp. Formally proving size optimality of sorting networks. *J. Autom. Reason.*, 59(4):425–454, 2017.

[15] N. G. de Bruijn. Sorting by means of swappings. *Discret. Math.*, 9(4):333–339, 1974.

[16] Robert W. Floyd and Donald E. Knuth. The Bose–Nelson sorting problem. In J.N. Srivastava, editor, *A Survey of Combinatorial Theory*, pages 163–172. North-Holland, 1973.

[17] Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In Phillip B. Gibbons and Christian Scheideler, editors, *Procs. SPAA*, pages 348–357. ACM, 2007.

[18] Milton W. Green. Some improvements in nonadaptive sorting algorithms. Technical report, Stanford Research Institute, Menlo Park, California, 1969.

[19] Jannis Harder. An answer to the bose-nelson sorting problem for 11 and 12 channels. *CoRR*, abs/2012.04400, 2020.

[20] Víctor Jiménez-Fernández, Carlos Ventura-Arizmendi, Denisse Martínez-Navarrete, and Francisco González Martínez. Digital architecture for a median filter of image based on sorting network. In *Procs. CISST*, pages 56–59. World Scientific and Engineering Academy and Society, 2011.

[21] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[22] John R. Koza., F.H. Bennett, J.L. Hutchings, S.L. Bade, Martin A. Keane, and D. Andre. Evolving sorting networks using genetic programming and the rapidly reconfigurable xilinx 6216 field-programmable gate array. In *Procs. Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 404–410, 1997.

[23] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257—263, 2023.

[24] Andreas Morgenstern and Klaus Schneider. Synthesis of parallel sorting networks using SAT solvers. In Frank Oppenheimer, editor, *Procs. MBMV*, pages 71–80. OFFIS-Institut für Informatik,

2011.

[25] Ian Parberry. A computer-assisted optimal depth lower bound for nine-input sorting networks. *Math. Syst. Theory*, 24(2):101–116, 1991.

[26] Ian Parberry. The pairwise sorting network. *Parallel Process. Lett.*, 2:205–211, 1992.

[27] Vinod K. Valsalam and Risto Miikkulainen. Using symmetry and evolutionary search to minimize sorting networks. *J. Mach. Learn. Res.*, 14(1):303–331, 2013.

[28] David C. van Voorhis. Toward a lower bound for sorting networks. In Raymond E. Miller and James W. Thatcher, editors, *Procs. COCO*, The IBM Research Symposia Series, pages 119–129. Plenum Press, New York, 1972.

[29] Jan Wassenberg, Mark Blacher, Joachim Giesen, and Peter Sanders. Vectorized and performance-portable quicksort. *Softw. Pract. Exp.*, 52(12):2684–2699, 2022.

[30] Mark D. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.