Applying Sorting Networks to Synthesize Optimized Sorting Libraries[⋆]

Michael Codish[1], Luís Cruz-Filipe[2],
Markus Nebel[2], and Peter Schneider-Kamp[2]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Israel
[2] Dept. Mathematics and Computer Science, Univ. of Southern Denmark,Denmark

Abstract. This paper presents an application of the theory of sorting networks to facilitate the synthesis of optimized general-purpose sorting libraries. Standard sorting libraries are often based on combinations of the classic Quicksort algorithm with insertion sort applied as base case for small, fixed, numbers of inputs. Unrolling the code for the base case by ignoring loop conditions eliminates branching, resulting in code equivalent to a sorting network. This enables further program transformations based on sorting network optimizations, and eventually the synthesis of code from sorting networks. We show that, if considering the number of comparisons and swaps, the theory predicts no real advantage of this approach. However, significant speed-ups are obtained when taking advantage of instruction level parallelism and non-branching conditional assignment instructions, both of which are common in modern CPU architectures. We provide empirical evidence that using code synthesized from efficient sorting networks as the base case for Quicksort libraries results in significant real-world speed-ups.

1 Introduction

General-purpose sorting algorithms are based on comparing, and possibly exchanging, pairs of inputs. If the order of these comparisons is predetermined by the number of inputs to sort and does not depend on their concrete values, then the algorithm is said to be data-oblivious. Such algorithms are well suited for e.g. parallel sorting or secure multi-party computations.

Sorting functions in state-of-the-art programming language libraries (such as the GNU C Library) are typically based on a variant of Quicksort, where the base cases of the recursion apply insertion sort: once the subsequence to sort considered by Quicksort falls under a certain length M, it is sorted using insertion sort. The reasons for using such base cases is that, both theoretically and empirically, insertion sort is faster than Quicksort for sorting small numbers of elements. Typical values of M are 4 (e.g. in the GNU C library) or 8.

Generalizing this construction, we can take any sorting algorithm based on the divide-and-conquer approach (e.g. Quicksort, merge sort), and use another

[⋆] Supported by the Israel Science Foundation, grant 182/13 and by the Danish Council for Independent Research, Natural Sciences.

sorting method once the number of elements to sort in one partition does not exceed a pre-defined limit M. The guiding idea here is that, by supplying optimized code for sorting up to M inputs, the overall performance of the sorting algorithm can be improved. One obvious way to supply optimized code for sorting up to M inputs is to provide a unique optimized implementaton of sorting m elements, for each $m \leq M$.

This approach leads directly to the following problem: *For a given fixed number M, how can we obtain an efficient way to sort M elements on a modern CPU?* Similar questions have been asked since the 1950s, though obviously with a different notion of what constitutes a modern CPU.

Sorting networks are a classical model of comparison-based sorting that provides a framework for addressing such questions. In a sorting network, n inputs are fed into n channels, connected pairwise by comparators. Each comparator compares the two inputs from its two channels, and outputs them sorted back to the same two channels. Consecutive comparators can be viewed as a "parallel layer" if no two touch the same channel. Sorting networks are data-oblivious algorithms, as the sequence of comparisons performed is independent of the actual input. For this reason, they are typically viewed as hardware-oriented algorithms, where data-obliviousness is a requirement and a fixed number of inputs is given.

In this paper, we examine how the theory of sorting networks can improve the performance of general-purpose software sorting algorithms. We show that replacing the insertion sort base case of a Quicksort implementation as found in standard C libraries by optimized code synthesized from logical descriptions of sorting networks leads to significant improvements in execution times.

The idea of using sorting networks to guide the synthesis of optimized code for base cases of sorting algorithms may seem rather obvious, and, indeed, has been pursued earlier. A straightforward attempt, described in [10], has not resulted in significant improvements, though. In this paper we show that this is not unexpected, providing theoretical and empirical insight into the reasons for these rather discouraging results. In a nutshell, we provide an average case analysis of the complexity w.r.t. measures such as number of comparisons and number of swaps. From the complexity point of view, code synthesized from sorting networks can be expected to perform slightly worse than unrolled insertion sort. Fortunately, for small numbers (asymptotic) complexity arguments are not always a good predictor of real-world performance.

The approach taken in [7] matches the advantages of sorting networks with the vectorization instruction sets available in some modern CPU architectures. The authors obtain significant speedups by implementing parallel comparators as vector operations, but they require a complex heuristic algorithm to generate sequences of bit shuffling code that needs to be executed between comparators. Their approach is also not fully general, as they target a particular architecture.

In this paper, we combine the best of both these attempts by providing a straightforward implementation of sorting networks that still takes advantage of the features of modern CPU architectures, while keeping generality. We obtain speedups comparable to [7], but our requirements to the instruction set are satisfied by virtually all modern CPUs, including those without vector operations. The success of our approach is based on two observations.

– Sorting networks are data-oblivious and the order of comparisons is fully determined at compile time, i.e., they are free of any control-flow branching. Comparators can also be implemented without branching, and on modern CPU architectures even efficiently so.
– Sorting networks are inherently parallel, i.e., comparators at the same level can be performed in parallel. Conveniently, this maps directly to implicit *instruction level parallelism* (ILP) common in modern CPU architectures. This feature allows parallel execution of several instructions on a single thread of a single core, as long as they are working on disjoint sets of registers.

Avoiding branching and exploiting ILP are tasks also performed through program transformations by the optimization stages of modern C compilers, e.g., by unrolling loops and reordering instructions to minimize data-dependence between neighbouring instructions. They are though both restricted by the data-dependencies of the algorithms being compiled and, consequently, of only limited use for data-dependent sorting algorithms, like insertion sort.

Throughout this paper, for empirical evaluations we run all code on an Intel Core i7, measuring runtime in CPU cycles using the time stamp counter register using the RDTSC instruction. As a compiler for all benchmarks, we used LLVM 6.1.0 with clang-602.0.49 as frontend on Max OS X 10.10.2. We also tried GCC 4.8.2 on Ubuntu with Linux kernel 3.13.0-36, yielding comparable results.

The remainder of the paper is organized as follows. Section 2 provides background information and formal definitions for both sorting algorithms and hardware features. In Section 3, we theoretically compare Quicksort and the best known sorting networks w.r.t. numbers of comparisons and swaps. We aggressively unroll insertion sort until we obtain a sorting network in Section 4, and in Section 5 we show how to implement individual comparators efficiently. We empirically evaluate our contribution as a base case of Quicksort in Section 6, before concluding and giving an outlook on future work in Section 7.

2 Background

2.1 Quicksort with Insertion Sort for Base Case

For decades, Quicksort has been used in practice, due to its efficiency in the average case. Since its first publication by Hoare [8], several modifications were suggested to improve it further. Examples are the clever choice of the pivot, or the use of a different sorting algorithm, e.g., insertion sort, for small subproblem sizes. Most such suggestions have in common that the empirically observed efficiency can be explained on theoretical grounds by analyzing the expected number of comparisons, swaps, and partitioning stages (see [13] for details).

Figure 1 presents a comparison of the common spectrum of data-dependent sorting algorithms for small numbers of inputs, depicting the number of inputs

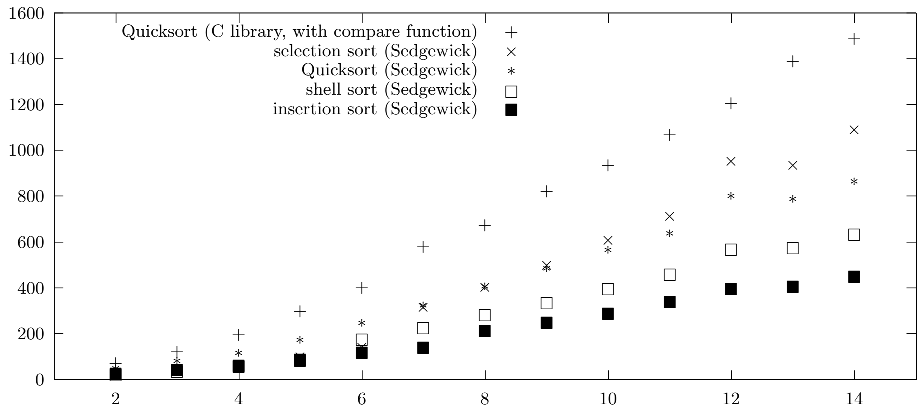

Fig. 1. Comparison of different sorting algorithms for small numbers of inputs.

(x-axis) together with the number of cycles required to sort them (y-axis), averaged over 100 million random executions. The upper curve in the figure is obtained from the standard Quicksort implementation in the C library (which is at some disadvantage, as it requires a general compare function as an argument). The remaining curves are derived from applying standard sorting algorithms, as detailed by Sedgewick [14]; the code was taken directly from the book's web page, http://algs4.cs.princeton.edu/home/. Insertion sort is the clear winner.

2.2 Sorting Networks

A *comparator network* on n channels is a finite sequence $C = c_1, \ldots, c_k$ of *comparators*, where each comparator c_ℓ is a pair (i_ℓ, j_ℓ) with $1 \leq i_\ell < j_\ell \leq n$. The *size* of C is the number k of comparators it contains. Given an input $\boldsymbol{x} \in D^n$, where D is any totally ordered domain, the *output* of C on $\boldsymbol{x}$ is the sequence $C(\boldsymbol{x}) = \boldsymbol{x}^n$, where $\boldsymbol{x}^\ell$ is defined inductively as follows: $\boldsymbol{x}^0 = \boldsymbol{x}$, and $\boldsymbol{x}^\ell$ is obtained from $\boldsymbol{x}^{\ell-1}$ by swapping the elements in positions i_ℓ and j_ℓ, in case $x_{i_\ell} < x_{j_\ell}$. C is a *sorting network* if $C(\boldsymbol{x})$ is sorted for all $C \in D^n$. It is well known (see e.g. [9]) that this property is independent of the concrete domain D.

Comparators may act in parallel. A comparator network C has *depth* d if C is the concatenation of $L_1, \ldots, L_d$, where each L_i is a *layer*: a comparator network with the property that no two of its comparators act on a common channel.

Figure 2 depicts a sorting network on 5 channels in the graphical notation we will use throughout this paper. Comparators are depicted as vertical lines, and layers are separated by a dashed line. The numbers illustrate how the input $10101 \in \{0,1\}^5$ propagates through the network. This network has 6 layers and 9 comparators.

There are two main notions of optimality of sorting networks in common use: *size* optimality, where one minimizes the number of comparators used in the network; and *depth* optimality, where one minimizes the number of execution steps, taking into account that some comparators can be executed in parallel.

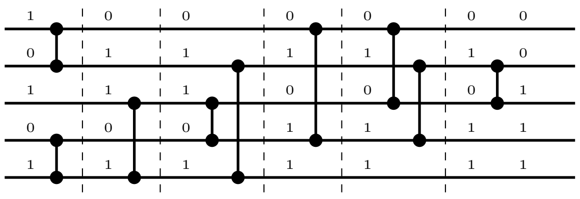

Fig. 2. A sorting network on 5 channels operating on the input 10101.

Given n inputs, finding the minimal size s_n and depth t_n of a sorting network is an extremely hard problem that has seen significant progress in recent years. The table below details the best currently known bounds. The values for $n \leq 8$ are already listed in [9]; the values of t_9 and t_{10} were proven exact by Parberry [11], those of t_{11}–t_{16} by Bundala and Závodný [1], and t_{17} was recently computed by Ehlers and Müller [5] using results from [3, 4]. Finally, the values of s_9 and s_{10} were first given in [2].

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
s_n	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60	73
											33	37	41	45	49	53	58
t_n	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9	10

Oblivious versions of classic sorting algorithms can also be implemented as sorting networks, as described in [9]. Figure 3 (a) shows an oblivious version of insertion-sort. The vertical dashed lines highlight the 4 iterations of “insertion” required to sort 5 elements. Figure 3 (b) shows the same network, with comparators arranged in parallel layers. Bubble-sort can also be implemented as a sorting network as illustrated in Figure 3 (c), where the vertical dashed lines illustrate the 4 iterations of the classic bubble-sort algorithm. When ordered according to layers, this network becomes identical to the one in Figure 3 (b).

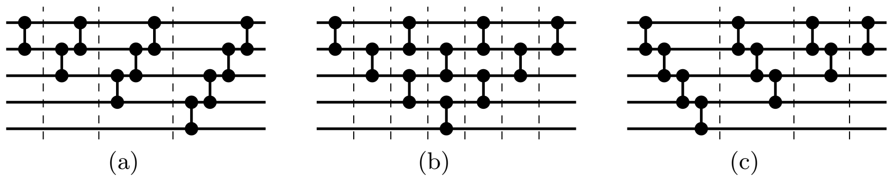

Fig. 3. Sorting networks for insertion sort (a) and bubble-sort (c) on 5 inputs, dashed lines separating iterations. When parallelized, both networks become the same (b).

2.3 Modern CPU Architectures

Modern CPU architectures allow multiple instructions to be performed in parallel on a single thread. This ability is called *instruction-level parallelism* (ILP), and is built on three modern micro-architectural techniques[1]:

– *superscalar instruction pipelines*, i.e., pipelines with the ability to hold and execute multiple instructions at the same time
– *dynamic out-of-order execution*, i.e., dynamic reordering of instructions respecting data dependencies
– *redundant execution units*, i.e., multiple Arithmetic Logic Units per core

Together, these features allow execution of instructions in an order that minimizes data dependencies, so that multiple redundant execution units can be used at the same time. This is often termed *implicit* ILP, in contrast to the explicit ILP found in vector operations.

Example 1. Consider the C expression `(x+y)*(z+u)`. Assume the variables `x`, `y`, `z`, and `u` are loaded in registers `eax`, `ebx`, `ecx`, and `edx`. Then the evaluation of the above expression is compiled to three machine instructions: `ADD eax,ebx; ADD ecx,edx; MUL eax,ecx`, with the result in `ecx`. Here, the first two instructions are data-independent and can be executed in parallel, while the last one depends on the results of those, and is executed in another CPU cycle.

Conditional branching instructions are the most expensive instructions on pipelined CPUs, as they require flushing and refilling the pipeline. In order to minimize their cost, modern CPU architectures employ dynamic branch prediction. By keeping the pipeline filled with the instructions of the predicted branch, the cost of branching is severely alleviated. Unfortunately, branch prediction cannot be perfect, and when the wrong branch is predicted, the pipeline needs to be flushed and refilled – an operation taking many CPU cycles.

In order to avoid branching instructions for "small" decisions, e.g., deciding whether to assign a value or not, modern CPU architectures also feature conditional instructions. Depending on flags set by e.g. a comparison, either an assignment of a value of a register will be performed, or the instruction will be ignored. In both cases, the pipeline is filled with the subsequent instructions, and the cost of the operation is smaller than a possible branch prediction failure.

Example 2. Consider the C statement `if (x == 42) x = 23;` with variable `x` loaded in `eax`. Without conditional move instructions, this is compiled to code with a conditional branching instruction, i.e. `CMP eax,42; JNZ after; MOV eax, 23`, where `after` is the address of the instruction following the `MOV` instruction. Alternatively, using conditional instructions, we obtain `CMP eax, 42; CMOVZ eax, 23`. This code not only saves one machine code instruction, but most importantly avoids the huge performance impact of a mispredicted branch.

[1] For details on these features of modern microarchitectures see e.g. [6, 15].

Table 1. Average number of comparisons and swaps when executing optimal sorting networks with at most $M = 14$ inputs.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
comparisons	0	1	3	5	9	12	16	19	25	29	35	39	45	51
swaps	0.0	0.5	1.5	2.7	4.8	6.6	8.6	10.6	13.0	11.1	19.4	22.4	20.0	26.5

3 Quicksort with Sorting Networks for Base Case

The general theme of this paper is to derive, from sorting networks, optimized code to sort small numbers of inputs, and then to apply this code as the base case in a Quicksort algorithm. In this section, we compare precise average case results for the number of comparisons and swaps performed by a classic Quicksort algorithm and by a modification that uses sorting networks on subproblems of size at most 14. We choose 14 for this analysis, as it is the largest value n for which we could conveniently measure the number of comparisons and swaps for all $n!$ permutations. We used the best-known (w.r.t. size) sorting networks (optimal for up to 10 inputs) in order to obtain the most favorable comparison numbers for sorting networks. To this end, we assume the algorithm to act on random permutations of size n, each being the input with equal probability.

Let C_n (resp. S_n) denote the expected number of comparisons (resp. swaps) performed by classic Quicksort on (random) inputs of size n. Let furthermore $\hat{C}_n$ and $\hat{S}_n$ denote the corresponding quantities for Quicksort using sorting networks for inputs smaller than 15. It is standard to set up recurrence relations for those quantities which typically obey a pattern such as:

$$T_n(a,b) = \begin{cases} a \cdot n + b + \frac{1}{n} \sum_{1 \le j \le n} T_{j-1}(a,b) + T_{n-j}(a,b) & \text{if } n > M, \\ g(n) & \text{otherwise.} \end{cases}$$

Here, a and b have to be chosen properly to reflect the parameter's (comparisons, swaps) behavior, M determines the maximum subproblem size for which a different algorithm (insertion sort, sorting networks) is used, and g accounts for the costs of that algorithm. In order to analyze classic Quicksort as proposed by Hoare, we have to choose $a = 1$, $b = -1$ (resp. $a = \frac{1}{6}$, $b = \frac{2}{3}$) for comparisons (resp. swaps), together with $M = 0$ and $g(0) = 0$. For the analysis of our proposed modification using sorting networks for subproblems of small sizes, we set $M = 14$ together with the values for g as given in Table 1. Using standard algebraic manipulations, it is possible to solve this recurrence explicitly to obtain a formula for $T_n(a,b)$ in terms of n, M, a and b. Defining $t_n = a \cdot n + b$ and $\nabla t_n = t_n - t_{n-1}$, one finds (see [12] for details) that, for $n > M$,

$$T_n(a,b) = 2(n+1) \sum_{M+2 \le k \le n} \frac{\nabla t_k}{k+1} + \frac{n+1}{M+2} \left(t_{M+1} + T_{M+1}(a,b)\right) - t_n \,.$$

Computing the closed form expressions for $\sum_{M+2\leq k\leq n} \frac{\nabla t_k}{k+1}$ for the different choices of t_n, we finally get

$$C_n = 2n\ln(n+1) - 2.84557n + o(n) \qquad S_n = \frac{1}{3}n\ln(n+1) + 0.359072n + o(n)$$
$$\hat{C}_n = 2n\ln(n+1) - 2.44869n + o(n) \qquad \hat{S}_n = \frac{1}{3}n\ln(n+1) + 0.524887n + o(n)$$

We see that, when increasing n, both parameters get worse by our modification of classic Quicksort. Even for small n and optimal size sorting networks, there is no advantage w.r.t. the numbers of comparisons or swaps. In conclusion, we cannot hope to get a faster sorting algorithm simply by switching to sorting networks for small subproblems – at least not on grounds of our theoretical investigations. And, by transitivity, replacing insertion sort by sorting networks in the base case should result in an even worse behavior w.r.t. both parameters.

4 Unrolling the Base Case

In this section, we show how to unroll an implementation of insertion sort, step by step, until we finally obtain code equivalent to a sorting network. We take the basic insertion sort code from Sedgewick [14], and, for illustration, assume that the fixed number of inputs is $n = 5$. We experimented also with optimized variants (e.g. making use of sentinels to avoid the `j>0` check), but did not find any of them to be faster for small inputs given a modern C compiler.

```
#define SWAP(x,y) {int tmp = a[x]; a[x] = a[y]; a[y] = tmp;}
static inline void sort5(int *a, int n) {
  n=5
  for (int i = 1; i < n; i++)
    for (int j = i; j > 0 && a[j] < a[j-1]; j--)
      SWAP(j-1, j)
}
```

Applying partial evaluation and (outer) loop unrolling results in:

```
static inline void sort5_unrolled(int *a) {
  for (int j = 1; j > 0 && a[j] < a[j-1]; j--)
    SWAP(j-1, j)
  for (int j = 2; j > 0 && a[j] < a[j-1]; j--)
    SWAP(j-1, j)
  for (int j = 3; j > 0 && a[j] < a[j-1]; j--)
    SWAP(j-1, j)
  for (int j = 4; j > 0 && a[j] < a[j-1]; j--)
    SWAP(j-1, j)
}
```

The condition in the inner loop is data-dependent, hence no sound and complete program transformation can be applied to unroll them. To address this, we move the data-dependent part of the loop condition to the statement in the body of the loop, while always iterating the variable `j` down to 1.

```
static inline void sort5_oblivious(int *a) {
  for (int j = 1; j > 0; j--)
    if (a[j] < a[j-1]) SWAP(j-1, j)
  for (int j = 2; j > 0; j--)
    if (a[j] < a[j-1]) SWAP(j-1, j)
  for (int j = 3; j > 0; j--)
    if (a[j] < a[j-1]) SWAP(j-1, j)
  for (int j = 4; j > 0; j--)
    if (a[j] < a[j-1]) SWAP(j-1, j)
}
```

Now we can now apply (inner) loop unrolling and obtain:

```
static inline void sort5_oblivous_unrolled(int *a) {
  if (a[1] < a[0]) SWAP(0, 1)
  if (a[2] < a[1]) SWAP(1, 2)
  if (a[1] < a[0]) SWAP(0, 1)
  if (a[3] < a[2]) SWAP(2, 3)
  if (a[2] < a[1]) SWAP(1, 2)
  if (a[1] < a[0]) SWAP(0, 1)
  if (a[4] < a[3]) SWAP(3, 4)
  if (a[3] < a[2]) SWAP(2, 3)
  if (a[2] < a[1]) SWAP(1, 2)
  if (a[1] < a[0]) SWAP(0, 1)
}
```

All the statements in the body of `sort5_oblivous_unrolled` are now conditional swaps. For readability, we move the condition into the macro. `COMP`s on the same line indicate that they originate from the same iteration of insertion sort:

```
#define COMP(x,y) { if (a[y] < a[x]) SWAP(x,y) }
static inline void sort5_fig3a(int *a) {
  COMP(0, 1)
  COMP(1, 2)  COMP(0, 1)
  COMP(2, 3)  COMP(1, 2)  COMP(0, 1)
  COMP(3, 4)  COMP(2, 3)  COMP(1, 2)  COMP(0, 1)
}
```

This sequence is equivalent to the sorting network in Figure 3 (a). Thus, we can apply the reordering of comparators that resulted in Figure 3 (b) to obtain the following implementation, where we reduce the number of layers to 7 (here, `COMP`s on the same line indicate a layer in the sorting network):

```
static inline void sort5_fig3b(int *a) {
  COMP(0, 1)
  COMP(1, 2)
  COMP(0, 1)  COMP(2, 3)
  COMP(1, 2)  COMP(3, 4)
  COMP(0, 1)  COMP(2, 3)
  COMP(1, 2)
  COMP(0, 1)
}
```

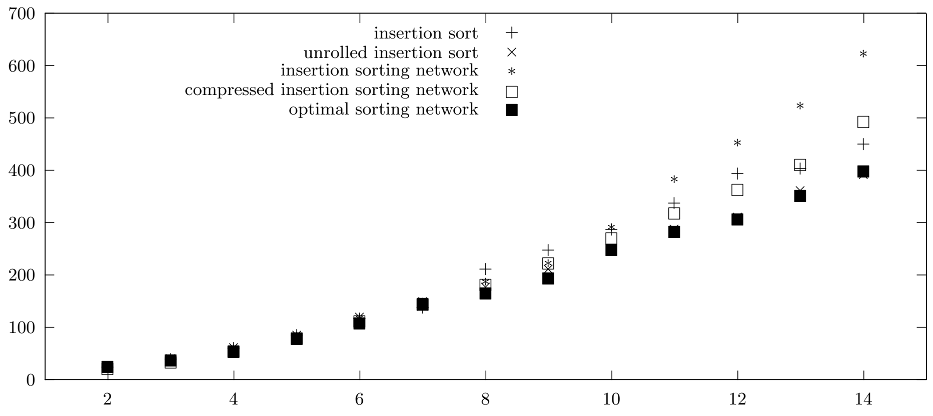

Fig. 4. Comparison of insertion sort with (unrolled) comparator based code for small numbers of inputs.

Figure 4 presents a comparison of a standard insertion sort (code from [14]) with the several optimized versions, depicting the number of inputs (x-axis) together with the number of cycles required to sort them (y-axis), averaged over 100 million random executions. The curve labeled "insertion sort" portrays the same data as the corresponding curve in Figure 1. The curve labeled "unrolled insertion sort" corresponds to the unrolled version of insertion sort (in the style of function `sort5_unrolled`). The other three curves correspond to code derived from different types of sorting networks: the "insertion sorting network" from Figure 3 (a) and function `sort5_fig3a`; the "compressed insertion sorting network" from Figure 3 (b) and function `sort5_fig3b`; and the "optimal sorting network", corresponding to the use of a best (smallest) known sorting network.

From the figure, it is clear that standard sorting network optimizations such as reordering of independent comparators [9] give a slight performance boost. But there is another clear message: even going beyond standard program transformations by breaking data-dependence and obtaining a sequence of conditional swaps (i.e., a sorting network), we do not manage to make any significant improvements of the performance of sorting implementations for small numbers of inputs. Furthermore, even when using size-optimal sorting networks, we obtain no real benefit over compiler-optimized insertion sort. This is in line with the theoretical results on average case complexity discussed in the previous section.

5 Implementing Sorting Networks Efficiently

The results in the previous two sections explained the rather discouraging results obtained by a naive attempt to use sorting networks as the base case of a divide-and-conquer sorting algorithm: they are simply not faster than e.g. insertion sort – at least when implemented naively. In this section we show how to exploit two main properties of sorting networks, together with features of modern CPU architectures, and obtain speed-ups of a factor higher than 3 compared to unrolled insertion sort.

We first observe that, as sorting networks are data-oblivious, the order of comparisons is fully determined at compile time, i.e., their implementation is free of any control-flow branching. Unfortunately, the naive implementation of each comparator involves branching to decide whether to perform a swap. The path taken depends entirely on the specific inputs to be sorted, and as such branch prediction necessarily does not perform very well.

Luckily, we can also implement comparators without branching. To this end, we use a conditional assignment (defined by the macro `COND` below), which can be compiled to the conditonal move (`CMOV`) instruction available on modern CPU architectures. This approach proved to be very fruitful. For illustration, from the optimal-size sorting network for 5 inputs portrayed in Figure 2, we synthesize the following C function `sort5_best`, where each row in the code corresponds to a layer in the sorting network:

```
#define COND(c,x,y) { x = (c) ? y : x; }
#define COMP(x,y) { int ax = a[x]; COND(a[y]<ax,a[x],a[y]); \
                                   COND(a[y]<ax,a[y],ax  ); }

static inline void sort5_best(int *a) {
  COMP(0, 1)  COMP(3, 4)
  COMP(2, 4)
  COMP(2, 3)  COMP(1, 4)
  COMP(0, 3)
  COMP(0, 2)  COMP(1, 3)
  COMP(1, 2)
}
```

The comparator macro that compares and conditionally swaps the values at indices `x` and `y` works as follows:

1. Keep a copy of the value at index `x`.
2. Compare (once) the value at index `y` with the stored value from `x`.
3. If the value was greater, copy the value at index `y` to index `x`. Otherwise, do nothing.
4. If the value was greater, write the old copied value from `x` to index `y`. Otherwise, do nothing.

Correctness follows directly by case analysis. If the value at index `y` was not greater than the value at index `x`, the two conditional assignments do not change anything, and all we did was an unnecessary copy of the valued at index `x`. If the value at index `y` was greater than the value at index `y`, we essentially perform a classic swap using `ax` as the temporary variable.

Given a sufficient optimization level (`-O2` and above), the above code is compiled by the LLVM (or GNU) C compiler to use two conditional move (`CMOV`) instructions, resulting in a totally branching free code for `sort5_best`. As can be expected, the other two instructions are a move (`MOV`) and a compare (`CMP`) instruction. In other words, each comparator is implemented by exactly four non-branching machine code instructions.

Alternatively, we could implement the comparator applying the folklore idea of swapping values using XORs to eliminate one conditional assignment:[2]

```
#define COND(c,x,y) { x = (c) ? y : x; }
#define COMP(x,y) { int ax = a[x]; COND(a[y]<ax,a[x],a[y]); \
                                    a[y] ^= ax ^ a[x]; }
```

This alternative comparator performs a conditional swap as follows:

1. Keep a copy of the value at index `x`.
2. If the value at index `y` is greater than the value at index `x`, copy the value at index `y` to index `x`.
3. Bitwise XOR the value at index `y` with the copied old and the new value at index `x`.

Step 3 works because, if the condition holds, then `ax` and the value at index `x` cancel out, leaving the value at `y` unchanged, while otherwise the value at `y` and `ax` cancel out, effectively assigning the original value from index `x` to index `y`.

We also implemented this variant, and observed that it compiles down to five instructions (`MOV`, `CMP`, `CMOV`, and two `XORs`). We benchmarked the two variants and observed that they are indistinguishable in practice, with differences well within the margin of measurement error. Thus, we decided to continue with this second version, as the `XOR` instructions are more "basic" and can therefore be expected to behave better w.r.t. e.g. instruction level parallelism.

A third approach would be to define branching-free minimum and maximum operations,[3] and use them to assign the minimum to the upper channel and the maximum to the lower channel of the comparator. We tested this approach, but found that it did not compile to branching-free code. Even if it did, the number of instructions involved would be rather large, eliminating any chance of competing with the two previous variants.

The reader might wonder whether a different `SWAP` macro could similarly speed up the working of standard insertion sort. The answer is a clear no, as the standard swapping operation is implemented by only three operations. Tricks like using `XORs` only increase the number of instructions to execute, while not reducing branching in the code. We implemented and benchmarked several alternative `SWAP` macros, finding only detrimental effects on measured performance.

Figure 5 compares three sorting algorithms for small numbers of inputs: (1) the unrolled insertion sort (also plotted in Figure 4); (2) code derived from a standard insertion sorting network (also plotted in Figure 4); (3) the same insertion sorting network but with a non-branching version of the `COMP` macro. We compare the number of branches encountered and mispredicted (averaged over 100 million random executions). From the figure it is clear that the number of branches encountered (and mispredicted) is larger for both unrolled insertion

[2] See https://graphics.stanford.edu/~seander/bithacks.html#SwappingValuesXOR

[3] See https://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax

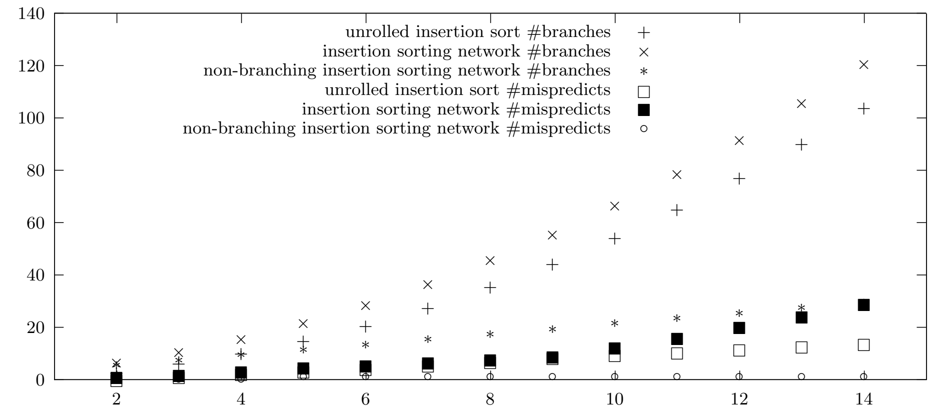

Fig. 5. Comparing the number of branches, encountered and mispredicted, in optimized sorting algorithms for small numbers of inputs.

sort and a naive implementation of sorting networks. In contrast, the branching-free implementation exhibits a nearly constant level of branches encountered and mispredicted. These branches actually originate from the surrounding test code (filling an array with random numbers, computing random numbers, and checking that the result is actually sorted).

Our second observation is that sorting networks are inherently parallel, i.e., comparators at the same level can be performed simultaneously. This parallelism can be mapped directly to instruction level parallelism (ILP). The ability to make use of ILP has further performance potential. In order to demonstrate this potential, we constructed artificial test cases with varying levels of data dependency. Given a natural number m, we construct a comparator network of size 1000 consisting of subsequences of m parallel comparators. We would expect that, as m grows, we would see more use of ILP.

In Figure 6, the values for m are represented on the x-axis, while the y-axis (as usual) indicates the averaged number of CPU cycles. Indeed, we see significant performance gains when going from $m = 1$ to $m = 2$ and $m = 3$. From this value

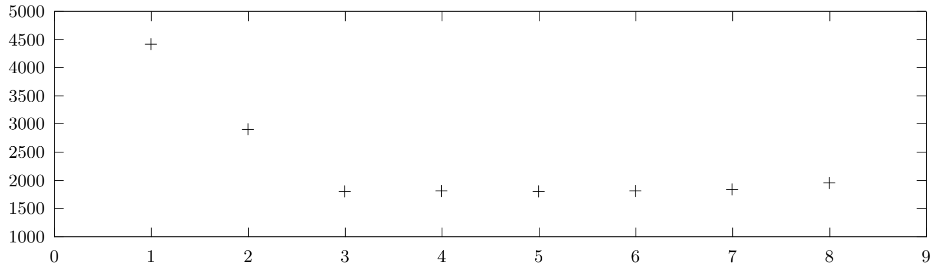

Fig. 6. ILP on comparator networks of length 1000 with differing levels of parallelism.

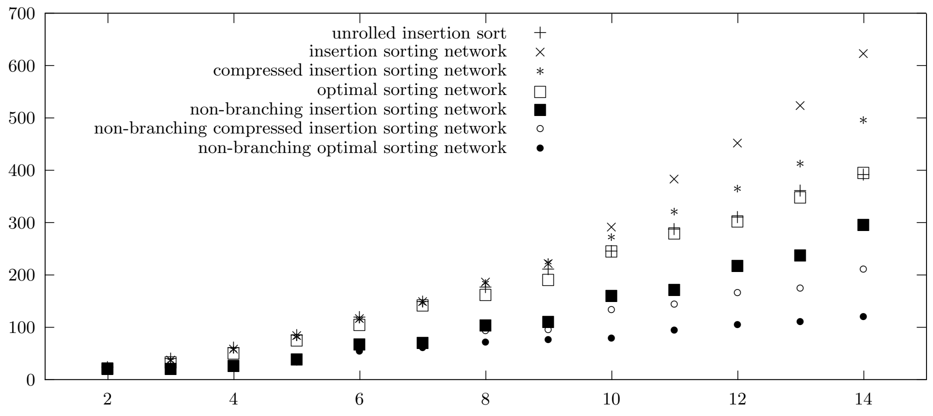

Fig. 7. Comparison of sorting networks for small numbers of inputs: non-branching sorting networks are fastest.

onwards, performance stays unchanged. This is the result of each comparator being compiled to 5 assembler instructions when using optimization level `-O3`. Then we obtain slightly under 2 CPU cycles per comparator.

Combining the gains from ILP with the absence of branching, we obtain large speed-ups for small inputs when comparing to both insertion sort and naive implementations of sorting networks. In Figure 7, we show the magnitude of the improvements obtained. Once again we plot the number of inputs on the x-axis against the number of cycles required to sort then on the y-axis, averaged over 100 million random executions. We consider the unrolled insertion sort, the three sorting networks from Figure 4 (insertion sorting network, compressed insertion sorting network, and optimal sorting network), and these same three sorting networks using non-branching comparators (non-branching insertion sorting network, non-branching compressed insertion sorting network, and non-branching optimal sorting network). The figure shows that using the best known (optimal) sorting networks in their non-branching forms results in a speed-up by a factor of more than 3.

6 Quicksort with Sorting Network Base Case

We now demonstrate that optimizing the code in the base case of a Quicksort algorithm translates to real-world savings when applying the sorting function. To this end, we use as base cases (1) the (empirically) best variant of insertion sort unrolled by applying program transformations to the algorithm from [14], and (2) the fastest non-branching code derived from optimal (size) sorting networks.

In Figure 8 we depict the results of sorting lists of 10,000 elements. The y-axis measures the number of cycles (averaged over one million random runs), and the x-axis specifies the limit at which Quicksort reverts to a base case. For example, the value 8 indicates that the algorithm uses a base case whenever it is required

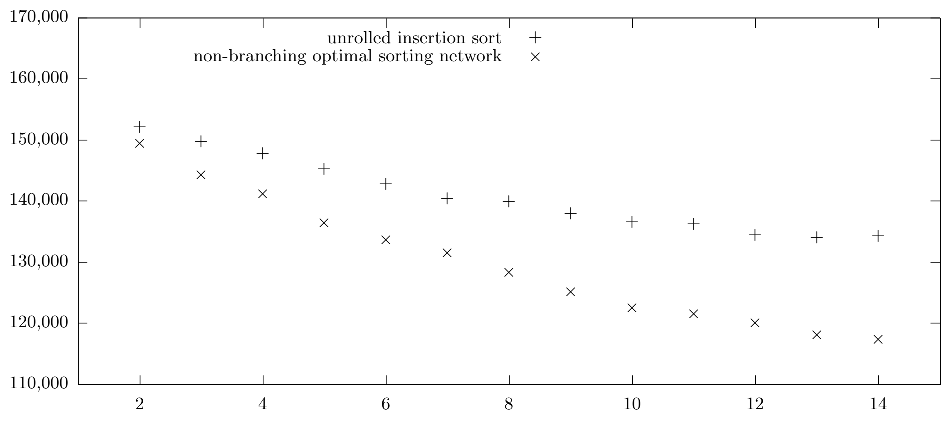

Fig. 8. Quicksort: comparing insertion sort at the base case with non-branching optimal sorting networks at the base case. Plotting base case size (x axis) and number of cycles (averaged over one million random runs).

to sort a sequence of length *at most* 8 elements. The value 2 corresponds to the case where the base case has no impact. To quantify the impact of the choice of base case, we compare to the case for value 2 (on the x-axis). For insertion sort we see a 2–12% reduction in runtime depending on the limit, and for non-branching sorting networks we achieve instead 7–23% reduction in runtime.

7 Conclusion

In this paper, we showed, both theoretically and empirically, that using code derived naively from sorting networks is not advantageous to sort small numbers of inputs, compared to the use of standard data-dependent sorting algorithms like insertion sort. Furthermore, we showed that program transformations are of only limited utility for improving insertion sort on small numbers of inputs.

By contrast, we showed how to synthesize simple yet efficient implementations of sorting networks, and gave insight into the microarchitectural features that enable this implementation. We demonstrated that we do obtain significant speed-ups compared to naive implementations such as [10]. A further empirical comparison between our implementation and the one described in [7] (not detailed in this paper) shows similar performance and scaling behavior. However, our approach allows the exploitation of instruction-level parallelism without the need for a complex instruction set-specific algorithm, as required by [7]. We also provided further evidence that efficient sorting networks are useful as a base case in divide-and-conquer sorting algorithms such as, e.g., Quicksort.

Our results also show that using different sorting networks has measurable impact on the efficiency of the synthesized C code. While previous research on finding optimal sorting networks has focused on optimal depth or optimal size, in the future we plan to identify criteria that will lead to optimal performance in this context. What are the parameters that determine real-world efficiency of the synthesized code, and how can we find sorting networks that optimize these parameters? We also plan to explore other target architectures, such as GPUs, and to benchmark our approach as base case for other sorting algorithms, such as merge sort.

References

1. D. Bundala and J. Závodný. Optimal sorting networks. In A.-H. Dediu, C. Martín-Vide, J.L. Sierra-Rodríguez, and B. Truthe, editors, *LATA 2014*, volume 8370 of *LNCS*, pages 236–247. Springer, 2014.
2. M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *ICTAI 2014*, pages 186–193. IEEE, December 2014.
3. M. Codish, L. Cruz-Filipe, and P. Schneider-Kamp. The quest for optimal sorting networks: Efficient generation of two-layer prefixes. In F. Winkler, V. Negru, T. Ida, T. Jebelan, D. Petcu, S.M. Watt, and D. Zaharie, editors, *SYNASC 2014*, pages 359–366. IEEE, 2015.
4. M. Codish, L. Cruz-Filipe, and P. Schneider-Kamp. Sorting networks: the end game. In A.-H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *LATA 2015*, volume 8977 of *LNCS*, pages 664–675. Springer, 2015.
5. T. Ehlers and M. Müller. New bounds on optimal sorting networks. In A. Beckmann, V. Mitrana, and M.I. Soskova, editors, *CiE 2015*, volume 9136 of *LNCS*, pages 167–176. Springer, 2015.
6. J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools.* Morgan Kaufman, 2005.
7. T. Furtak, J.N. Amaral, and R. Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *SPAA '07*, pages 348–357. ACM, 2007.
8. C.A.R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.
9. D.E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.
10. B. Lopez and N. Cruz-Cortes. On the usage of sorting networks to big data. In H.R. Arabnia, M.Q. Yang, G. Jandieri, J.J. Park, A.M.G. Solo, and F.G. Tinetti, editors, *Advances in Big Data Analytics: The 2014 WorldComp International Conference Proceedings.* Mercury Learning and Information, 2014.
11. I. Parberry. A computer-assisted optimal depth lower bound for nine-input sorting networks. *Mathematical Systems Theory*, 24(2):101–116, 1991.
12. R. Sedgewick. The analysis of quicksort programs. *Acta Inf.*, 7:327–355, 1977.
13. R. Sedgewick and P. Flajolet. *An introduction to the analysis of algorithms.* Addison-Wesley-Longman, 1996.
14. R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, 2011. 4th Edition.
15. J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond.* Springer, 1999.