Under consideration for publication in Formal Aspects of Computing

Optimizing Sorting Algorithms by Using Sorting Networks

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]Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Abstract. In this paper, we show how the theory of sorting networks can be applied to synthesize 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. By replacing it with faster sorting networks, we can improve the performance of these algorithms. We show that by considering the number of comparisons and swaps alone we are not able to predict any 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. Furthermore, a close control of how often registers have to be spilled to memory gives us a complete explanation of the performance of different sorting networks, allowing us to choose an optimal one for each particular architecture. Our experimental results show that using code synthesized from these efficient sorting networks as the base case for Quicksort libraries results in significant real-world speed-ups.

Keywords: sorting algorithms, sorting networks, instruction-level parallelism, out-of-order execution

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 [GZ06] or secure multi-party computations (see e.g. [EGT10]).

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 falls under a certain length M, it is sorted using insertion sort. The reasons for using

Correspondence and offprint requests to: Luís Cruz-Filipe, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 ODENSE M, Denmark

such base cases is that, both theoretically and empirically, insertion sort is faster than Quicksort for sorting small numbers of elements. Typical values used for 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 (not only Quicksort, but also e.g. merge sort), and use another 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. A set of consecutive comparators such that no two touch the same channel can be viewed as a "parallel layer" that can be executed simultaneously. 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 work, 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 similar to those in standard C libraries by optimized code synthesized from logical descriptions of sorting networks leads to significant improvements in execution times. Analogous experiments with an efficient array-based merge sort implementation [SW11] provide evidence that similar improvements also apply to other divide-and-conquer general-purpose sorting algorithms.

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 [LCC14], has not resulted in significant improvements, though. 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 actually be expected to perform slightly worse than unrolled insertion sort (where all the loops are replaced by explicit repetition of code). However, for small numbers (asymptotic) complexity arguments are not always a good predictor of real-world performance.

A different approach, taken in [FAN07], 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 work, 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. Furthermore, we show how particular characteristics of the hardware (such as e.g. the number of general purpose registers) can be taken into account, so that we systematically generate sorting networks that are better suited to it. In this manner, we are able to obtain speedups comparable to [FAN07], with more general requirements to the instruction set that are satisfied by virtually all modern CPUs, including those without vector operations. The success of our approach is based on the following 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.

The potential for parallelization is enhanced by a common capability of modern CPUs, namely *out-of-order*

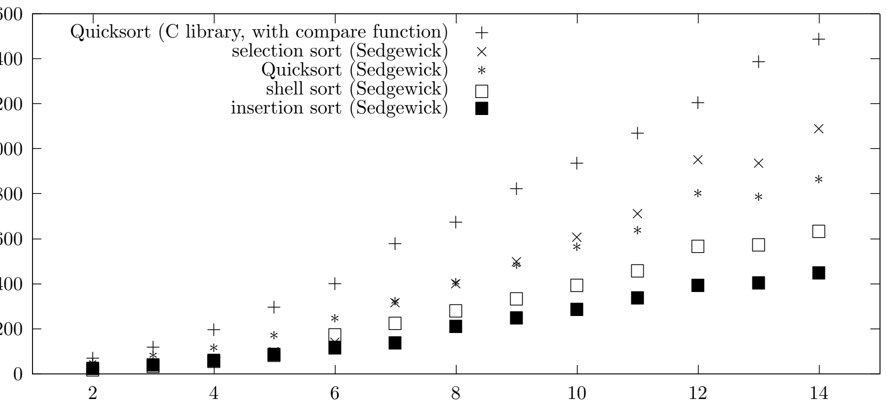

Fig. 1. Comparison of different sorting algorithms for small numbers of inputs.

execution, which allows for executing data-independent CPU instructions in a different order than specified. 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. Both of these optimizations are though restricted by the data-dependencies of the algorithms being compiled and, consequently, of only limited use for data-dependent sorting algorithms like insertion sort.

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. Section 4 shows that aggressively unrolling insertion sort yields a sorting network, and discusses how sorting networks can be implemented efficiently. We then replace insertion sort by an optimal sorting network, and empirically evaluate our contribution as a base case of both Quicksort and merge sort in Section 5. We also show that by increasing the limit M at which we resort to the base case we can further improve the performance of these algorithms; however, this requires considering further characteristics of the underlying hardware. In Section 6 we show how different measures of the quality a sorting network can be taken into account to generate performance-optimal sorting networks systematically, before concluding and giving an outlook on future work in Section 7. This paper extends work previously presented in [CCFNSK15].

2. Background

2.1. Quicksort with Insertion Sort for Base Case

Since its first publication by Hoare [Hoa62], Quicksort has been extensively used in practice, due to its efficiency in the average case, and several modifications have been 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 [SF96] 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 (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 [SW11]; the code was taken directly from the book's web page, `http://algs4.cs.princeton.edu/home/`. This analysis shows insertion sort to be the clear winner.

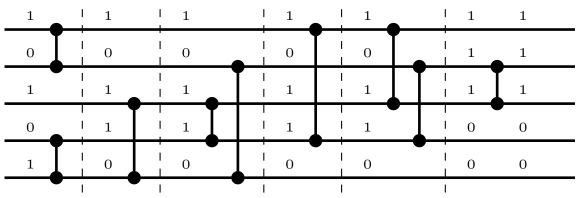

Fig. 2. A sorting network on 5 channels operating on the input 10101.

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 $\vec{x} \in D^n$, where D is any totally ordered domain, the *output* of C on $\vec{x}$ is the sequence $C(\vec{x}) = \vec{x}^n$, where $\vec{x}^\ell$ is defined inductively as follows:

- $\vec{x}^0 = \vec{x}$;
- $\vec{x}^\ell$ is obtained from $\vec{x}^{\ell-1}$ by swapping the elements in positions i_ℓ and j_ℓ, in case $x_{i_\ell} < x_{j_\ell}$;
- $\vec{x}^\ell$ is $\vec{x}^{\ell-1}$, if $x_{i_\ell} \geq x_{j_\ell}$.

A comparator network C is a *sorting network* if $C(\vec{x})$ is sorted for all $C \in D^n$. It is well known (see e.g. [Knu73]) that this property is independent of the concrete domain D, and in particular can be decided by considering only binary sequences.

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. Channels are depicted as horizontal lines, numbered from bottom to top, 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. The network in Figure 2 is size optimal, but not depth optimal.

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 $n \leq 8$ were already known in the 1960s, and are listed in [Knu73]; the values of t_9 and t_{10} were proven exact by Parberry [Par91], those of t_{11}–t_{16} by Bundala and Závodný [BZ14], and t_{17} was recently computed by Ehlers and Müller [EM15] using results from [CCFSK15a, CCFSK15b]. Finally, the values of s_9 and s_{10} were first given in [CCFFSK14, CCFFSK16].

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 several classic sorting algorithms can also be implemented as sorting networks, as described in [Knu73]. 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. As observed in [Knu73], when ordered according to layers, this network becomes identical to the one in Figure 3 (b).

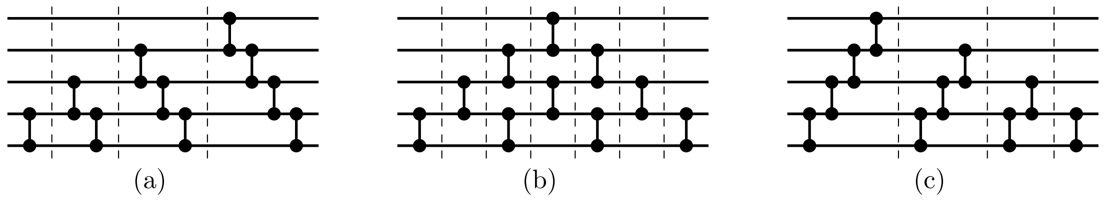

Fig. 3. Sorting networks for insertion sort (a) and bubble-sort (c) on 5 inputs, dashed lines separating iterations. When parallelized, both networks become the same (b).

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 instructions to be rearranged into 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`. The first two instructions are data-independent and can be executed in parallel, while the last one depends on the results of those, and has to be 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.

2.4. Benchmarking on a Cycle Level

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(P) instruction.

[1] For details on these features of modern microarchitectures see e.g. [FFY05, SRU99].

We employed two different measuring methodologies, depending on the resolution needed. For most benchmarks we used *simple average benchmarking*, where we measure the total number of CPU cycles used for a given number of repetitions (usually 100,000,000 randomly generated inputs). This measurement is performed by using the RDTSC instruction before and after the iteration through all the repetitions. The total number of cycles is then divided by the number of repetitions to obtain an average cycle count for one-time execution. The advantages of this method are its simplicity and its applicability to both branching and non-branching code; the drawback is that it includes overhead from iterating the code, as well as from preemption and hardware interrupt events. As a compiler for benchmarks using simple average benchmarking, we used LLVM 6.1.0 with clang-602.0.49 as frontend on Mac OS X 10.10.2. We also tried GCC 4.8.2 on Ubuntu with Linux kernel 3.13.0-36 for this benchmarking method, yielding comparable results.

For benchmarks of non-branching code where a higher precision and resolution was desired, we employed the *kernel level minimum benchmarking* technique described in an Intel white paper [Pao10]. Here, we prefixed a single execution of the code by four instructions: the CPUID instruction, which acts as a serializing guard w.r.t. to ILP and out-of-order execution, followed by the RDTSC instruction to read the time stamp counter register, and finally two MOV instructions to save the value to a local variable. The code was then suffixed by another four instructions: the RDTSCP instruction, which is in itself serializing, two MOV instructions to save to a local variable, and a final CPUID instruction to serialize also w.r.t. these instructions. In addition, we used optimizations from the afore-mentioned white paper regarding priming the instruction cache.

In order to avoid preemption and hardware interrupts, the code is executed in a custom kernel module, where these features are temporarily disabled. Given the limits on allocating kernel memory, we performed 100 iterations, each consisting of 10,000 repetitions of the code. The global minimum computed is, thus, the minimum of a total of 1,000,000 repetitions. The resolution of our measurement was found to be 3 cycles and reliability was found to be very high, with repeated benchmarks for the same code yielding identical global minima. This is in contrast with traditional repeat-and-get-minimum or repeat-and-average benchmarking approaches, which we empirically found to be less reliable and an order of magnitude less precise. Using kernel level minimum benchmarking, we are able to get high-precision measurements of rather short code pieces. The disadvantage of this method is that it can only be used for code that is data-independent such as e.g. non-branching implementations of sorting networks (see Section 4.2). As a compiler for benchmarks using kernel level minimum benchmarking, we used GCC 4.8.4 on Ubuntu with Linux kernel 3.15.0-51.

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 divide-and-conquer general-purpose sorting algorithm such as Quicksort or merge sort.

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 (which have been proven 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 that 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

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

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

$M = 0$ 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. By defining $t_n = a \cdot n + b$ and $\nabla t_n = t_n - t_{n-1}$, one finds (see [Sed77] for details) that, for $n > M + 1$,

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

Computing the closed form expressions for $\sum_{M+2 \le k \le n} \frac{\nabla t_k}{k+1}$ for the different choices of a and b, 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 a sorting network in the base case should result in an even worse behavior w.r.t. both parameters.

4. Implementing the Base Case Efficiently

In this section, we first show how to derive a data-oblivous implementation of the insertion sort base case, which corresponds to viewing it as a sorting network. Then, we optimize the implementation of sorting networks in general to obtain a more efficient implementation of the base case of divide-and-conquer sorting algorithms. For a more detailed presentation, the reader is referred to [CCFNSK15].

4.1. Unrolling Insertion Sort

Here, 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 [SW11], and, for illustration, assume that the fixed number of inputs is $n = 5$. From this point onwards, we number the channels from 0 to $n - 1$. 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)
}
```

Given that n is fixed, we can obviously unroll the outer loop, as the guard is only dependent on n. In order to unroll the inner loop, we move the data-dependent part of its guard to its body, obtaining the following, seemingly less efficient, data-oblivious code:

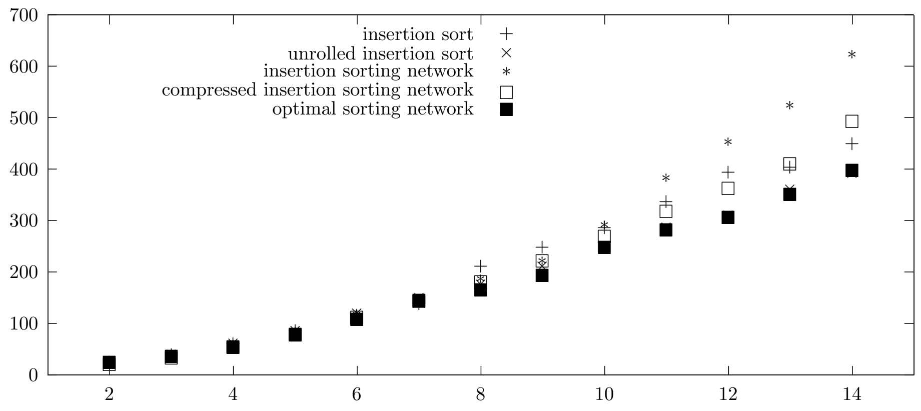

Fig. 4. Comparison of insertion sort with (unrolled) comparator based code for small numbers of inputs.

```
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, each corresponding exactly to one comparator in a sorting network. Indeed, this sequence is equivalent to the 8-layer sorting network in Figure 3 (a). Thus, we can apply the reordering of comparators that resulted in Figure 3 (b) to obtain an implementation with only 7 layers.

Figure 4 presents a comparison of a standard insertion sort (code from [SW11]) with several optimized versions, depicting the number of inputs (x-axis) together with the number of cycles required to sort them (y-axis), using standard average benchmarking 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 a version of insertion sort where only the outer loop has been unrolled. The other three curves correspond to code derived from different types of sorting networks: the "insertion sorting network" from Figure 3 (a), implemented by function `sort5_oblivious unrolled`; the "compressed insertion sorting network" from Figure 3 (b); 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 [Knu73] 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.

4.2. Implementing Sorting Networks Efficiently

The previous results explained the rather discouraging results obtained by a first 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 as in [LCC14]. 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.

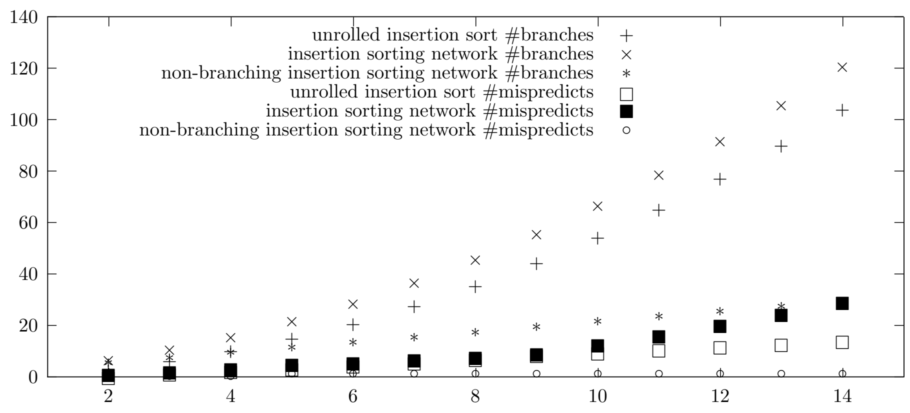

Fig. 5. Comparing the number of branches, encountered and mispredicted, in optimized sorting algorithms for small numbers of inputs.

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 conditional 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)
}
```

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`). In other words, each comparator is implemented by exactly four non-branching machine code instructions.

Alternatively, we can implement the comparator applying the folklore idea of swapping values using `XOR`s 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 variant compiles down to five instructions (`MOV`, `CMP`, `CMOV` and two `XOR`s). We benchmarked these two implementations 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.

We further 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

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

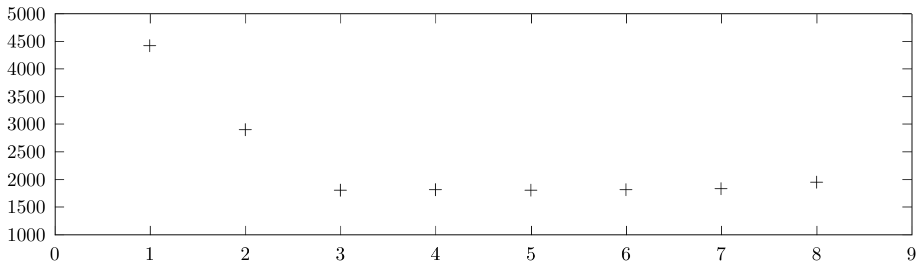

Fig. 6. ILP on comparator networks of length 1000 with differing levels of parallelism.

(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, again using simple average benchmarking 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 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 built 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 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, using simple average benchmarking 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.

5. Practical Implications in the Context of Quicksort and Merge Sort

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 [SW11], and (2) the fastest non-branching code derived from optimal (size) sorting networks – marked respectively as + and • in Figure 7.

In Figure 8 we depict the results of sorting lists of 10,000 elements. The y-axis measures the number of cycles (using simple average benchmarking 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 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

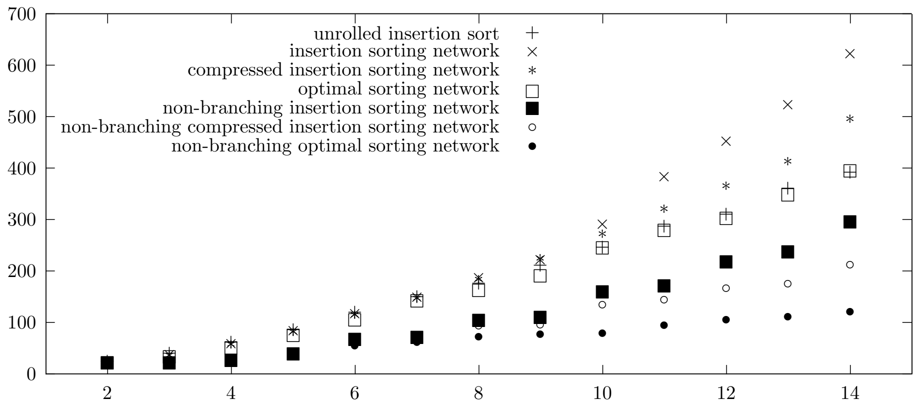

Fig. 7. Comparison of sorting networks for small numbers of inputs: non-branching sorting networks are fastest.

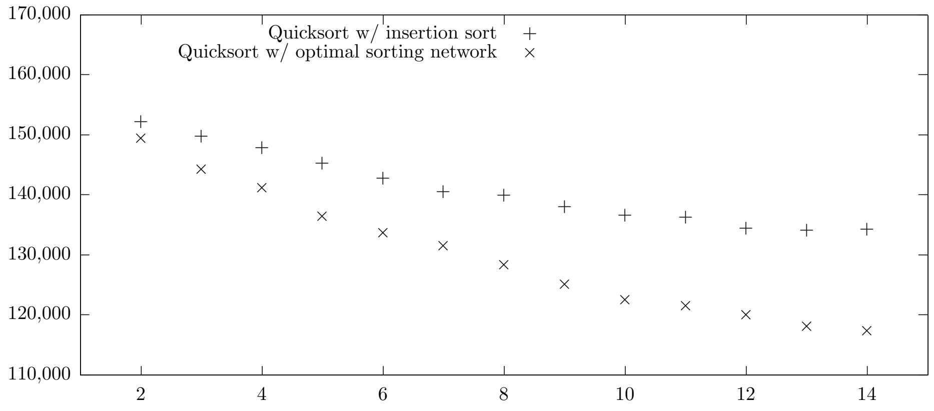

Fig. 8. Quicksort: comparing insertion sort at the base case with non-branching optimal sorting networks at the base case. Plotting base case size (x axis) and number of cycles (averaged over one million random runs).

the case for value 2 (on the x-axis). For insertion sort we see a 2–12% reduction in runtime depending on the limit, while for non-branching sorting networks we achieve instead a 7–23% reduction in runtime.

Further experiments with systematics constructions, in particular with Batcher's odd-even sorting networks [BB11, Bat68], have shown that this trend of runtime reductions continues well beyond 14 inputs. We investigated the runtime behavior of quicksort with base case sizes of up to 128 inputs and array sizes of 1,000 to 1,000,000 elements, finding that virtually all the achievable runtime reduction was already realized with base case sizes of at most 32 and that runtime would soon start increasing again for larger base case sizes. For example, for 10,000 elements (as used for Figure 8) the minimum was reached at a base case size of 33.

In order to evaluate the applicability of this optimization to other divide-and-conquer general-purpose sorting algorithms, we performed similar experiments for an optimized array-based merge sort implementation from [SW11]. As for Quicksort, we varied the limit at which merge sort delegates to the base case, and measured the cycles used for sorting 10,000 elements. As can be seen in Figure 9, using non-branching optimal sorting networks provides significant runtime reductions, on the order of 2–20%. Note that this holds for all the tested limit sizes, even in the face of merge sort being more sensitive to the choice of this limit. In particular, as merge sort is known to work particularly well for sequences whose length is a power of two, the limit sizes should should not be close to such values. This is illustrated by the data presented in Figure 9, where the locally smallest improvements are found for limit sizes of 4 and 8, while the largest improvements are found for 6 and 11.

Both sets of experiments suggest that similar improvements would likely be obtained for other divdide-and-conquer general-purpose sorting algorithms.

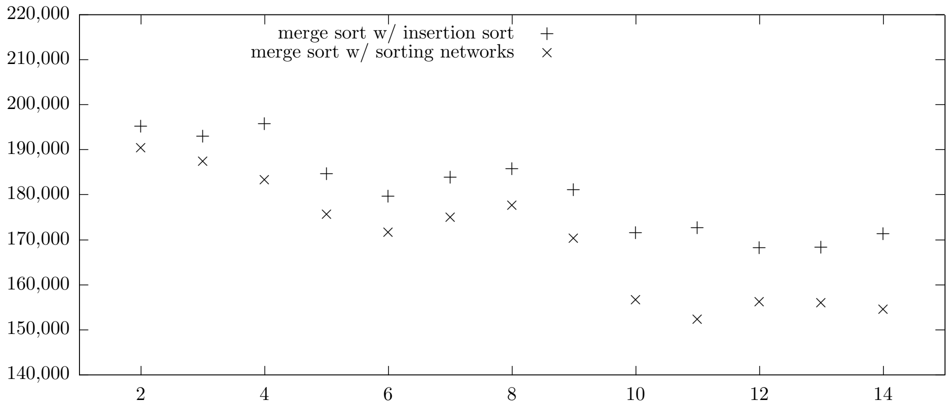

Fig. 9. Merge sort: comparing insertion sort at the base case with non-branching optimal sorting networks at the base case. Plotting base case size (x axis) and number of cycles (averaged over one million random runs).

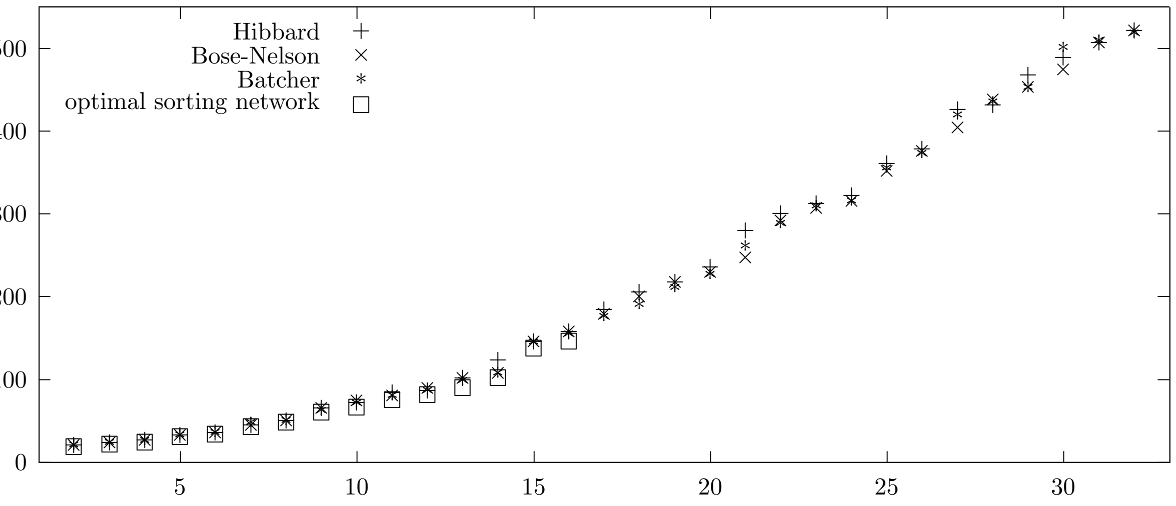

Fig. 10. Runtime in cycles for optimal (up to 16 inputs) sorting networks and three systematic constructions (up to 32 inputs)

6. Selecting and Optimizing Sorting Networks for Efficient Implementation

In the previous sections we have seen that non-branching implementations of optimal sorting networks sort small numbers of inputs several times faster than (unrolled) insertion sort, that this advantage increases as we increase the number of inputs, and that these speed-ups can be used to speed up the base case of Quicksort by reverting to sorting networks at around 32 inputs. In this section we explore which sorting networks one should select for obtaining an efficient implementation. To this end, in addition to the smallest known sorting networks (which have been proven to be size optimal only up to 10 inputs [CCFFSK16]), we study the systematic constructions of Batcher [BB11, Bat68], Bose & Nelson [BN62] and Hibbard [Hib63], which can be used to generate sorting networks for arbitrarily large numbers of inputs. We use the implementations of these constructions from [Gam11]. We also show how these constructions can be uniformly adapted to yield optimized real-world performance by cleverly rearranging comparators.

In Figure 10 we benchmark the execution times in cycles for the families of sorting networks mentioned above from 2 to 32 inputs, comparing with the optimal size sorting network. Here, we employ the kernel level benchmarking method described in Section 2.4.

The first observation is that for the values of up to 16, the optimal sorting networks are the most efficient ones, reflecting the fact that the number of comparators is highly responsible for their performance. However, we also observe a non-continuous change in performance at around 14 to 15 inputs. The reason for this can

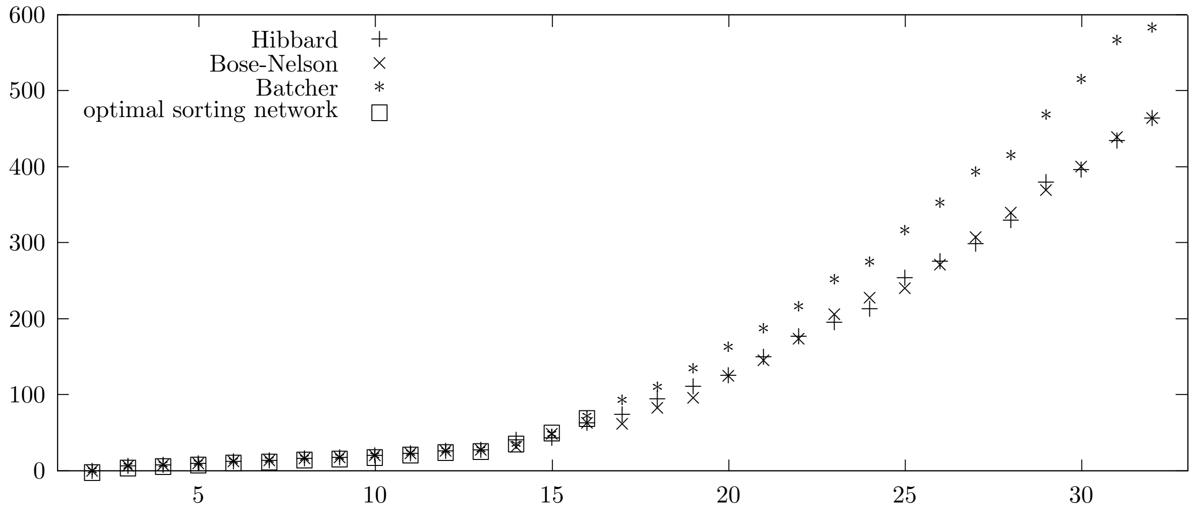

Fig. 11. Number of memory accesses (read + write) for optimal sorting networks and three systematic constructions

be found in the number of general-purpose registers available in the CPU. The CPU architecture we use provides 16 general purpose registers; given that out-of-order execution and ILP are executing up to three comparators concurrently, there is a need for three registers to keep intermediate results. The remaining 13 registers are available to keep the values of up to 13 channels. When working with more than 13 channels, the values have to be intermittently written to memory and read again when needed (this is called *register spilling*). Figure 11 shows the number of memory accesses for the different sorting networks. As expected, for up to 13 inputs there is exactly one read and one write for each channel; the number of memory accesses then grows superlinearly from 14 inputs.

Looking more closely at Figure 10, we further observe that the number of comparators in a sorting network is only a part of the explanation of its real-world performance when there are more than 13 inputs. Consider in particular the case of the Batcher and the Bose–Nelson sorting networks for 21 inputs. The Batcher network has just 107 comparators while the Bose–Nelson network has 118 comparators, but the runtime for the Batcher network is larger (261 cycles) than for the Bose–Nelson network (248 cycles). To understand this discrepancy, we also need to look at the number of memory accesses required by each network (Figure 11), which we find to be 187 for Batcher, compared to only 145 for Bose–Nelson.

These particular sorting networks can be seen in Figures 12 and 13. Observing them, we see that Batcher's construction has a tendency to go repeatedly from the bottom to the top channel, thus requiring a number of channels close to the number of inputs to be kept in registers. On the other hand, in Bose–Nelson's construction the comparators tend to clump around the same channels, which helps out-of-order execution perform a better job; and the outermost channels stop being used, so that the last third of the network actually does not require any register spilling.

We now show how we can optimize these systematic constructions for sorting networks on more than 16 channels taking into account the considerations above. We focus first on Bose & Nelson's construction, as it is visually easier to understand, and then show how to adapt the same ideas to Batcher's odd-even sorting networks.

6.1. Bose–Nelson sorting networks

We start by looking at the particular case of the 29-channel Bose–Nelson sorting network. There is no particular mystery about the choice of 29, but it works well for our presentation as it is high enough to serve as an example, and it is an odd number, which reduces the symmetry of the network.

Figure 14 depicts the Bose–Nelson sorting network on 29 channels as generated from [Gam11]. Parallelism is maximized by layering comparators so that as many comparisons as possible take place at the same time;

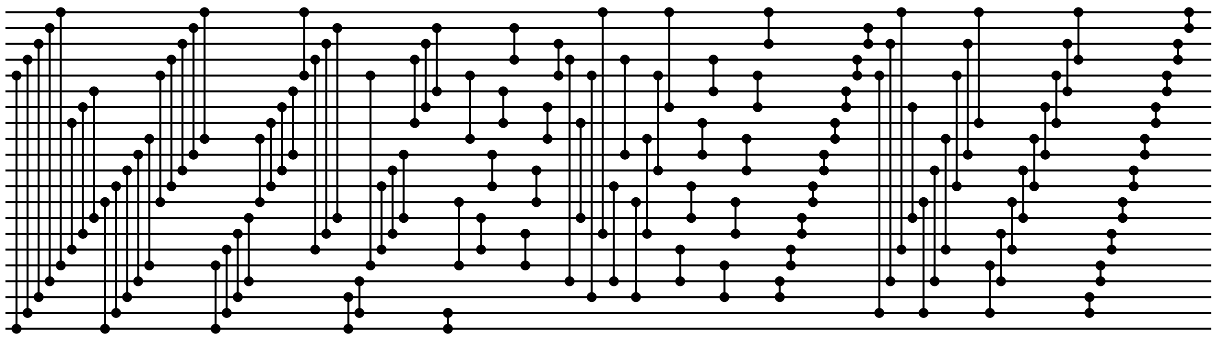

Fig. 12. Batcher sorting network on 21 channels

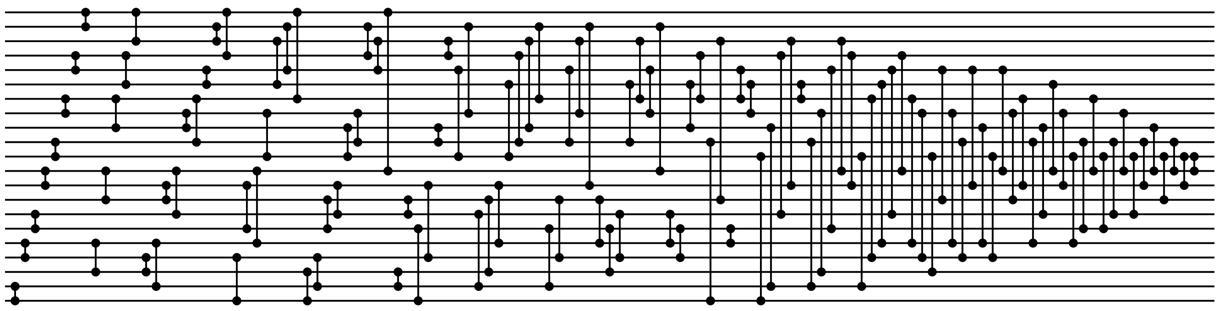

Fig. 13. Bose–Nelson sorting network on 21 channels

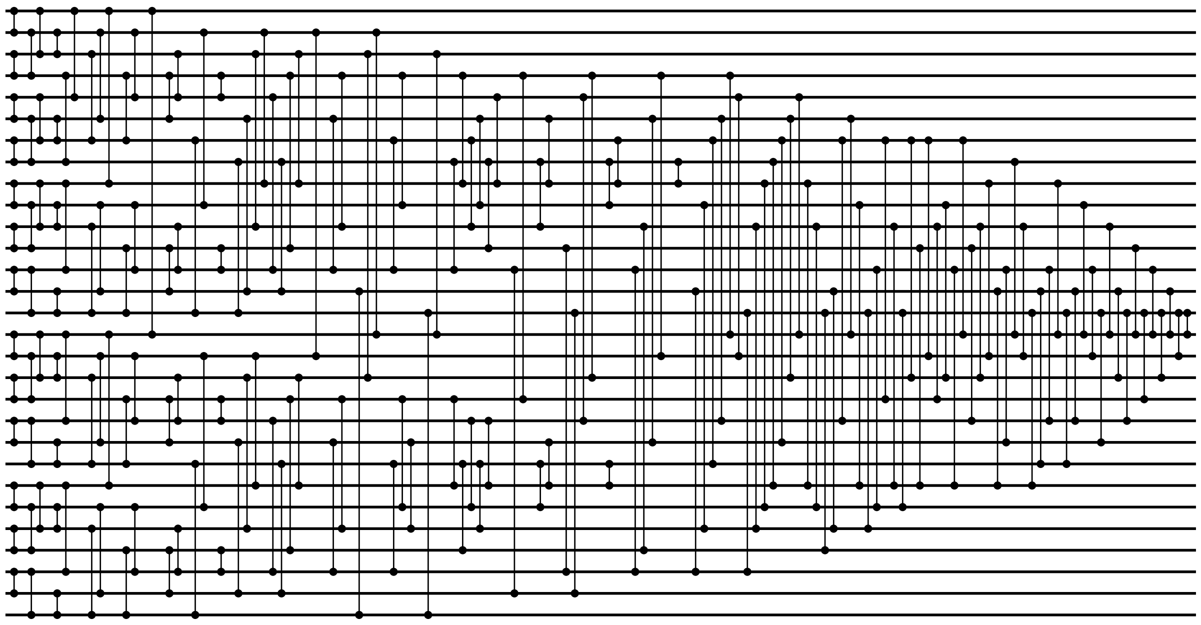

Fig. 14. The usual presentation of the Bose–Nelson sorting network on 29 channels, maximizing parallelism.

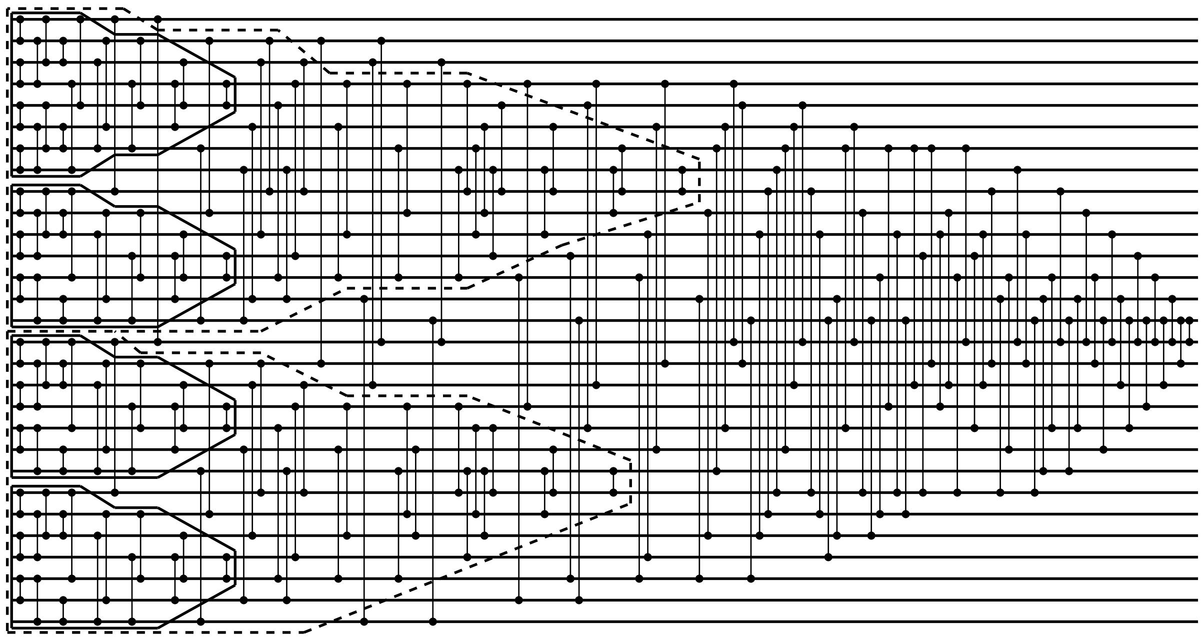

Fig. 15. The same network as in Figure 14, but with the recursive calls on 7 or more channels marked as boxes. Observe that the beginning of the mergers is typically interleaved with the end of the previous recursive calls.

in particular, the first four steps perform 14 comparisons in parallel. As computation proceeds, the outermost channels stop being used, and the number of values involved in comparisons gradually decreases.

This family of sorting networks is built recursively by means of a *merger*: an (n,m)-merger is a comparator network that merges two sorted sequences of lengths n and m into one sorted sequence of length $n+m$. The algorithm for building a Bose–Nelson sorting network on n channels is then simply:

1. build a sorting network on the first $\left\lfloor \frac{n}{2} \right\rfloor$ channels;
2. build a sorting network on the last $\left\lceil \frac{n}{2} \right\rceil$ channels;
3. juxtapose the previous networks and append an $\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil\right)$-merger.

By applying this construction recursively (taking as base cases the empty network on 1 channel and a single comparator on 2 channels) we can generate a network on any number of channels. Typically, the merger is also "pushed back" in order to maximize parallelism.

In Figure 15 we box the recursive calls on 7 or more channels. Observe that these do not interact; this suggests that we slightly modify the above construction as follows: if $n > 13$ (the number of available registers), we *append* the networks built recursively instead of juxtaposing them. This yields the network shown in Figure 16.

Running this version of the Bose–Nelson sorting network yields a performance gain of around 7%. The lower level of parallelism does not affect execution time negatively, as we can only execute three instructions simultaneously anyway, but we gain a lot from not needing to update the values in the registers constantly.

Experiments show that the optimizations performed by the compiler and out-of-order execution will again interleave the end of each recursively built sorting network with the beginning of the next one (or the next merger), while preserving the benefit gained from working as much as possible on the same set of at most 13 channels. Therefore, we do not need to worry about the nearly sequential code at the end of each of the smaller sorting networks produced by this algorithm.

Figure 17 shows the gain in performance by systematically rearranging the comparators in the Bose–Nelson sorting networks in the manner described. We are using 13 as the limit to the size of the recursive calls we parallelize rather than sequentialize, as this is the number of registers available in our hardware, but this number can of course be changed to any adequate value. The data reveals an increase in performance of the order of up to 15%.

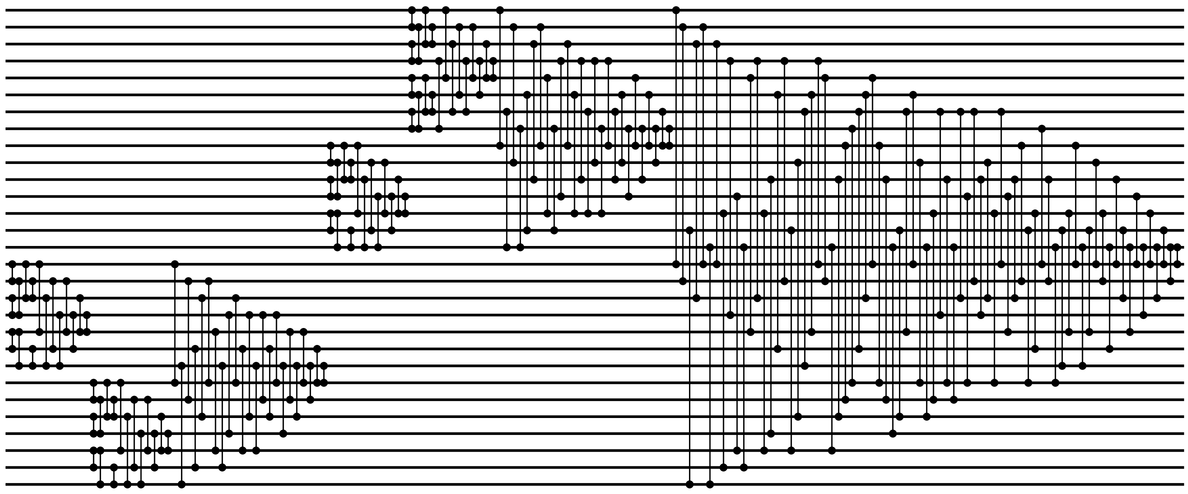
Fig. 16. The Bose–Nelson sorting network on 29 channels, optimized for performance on 13 registers. The larger sorting networks built in the recursive calls are now clearly visible.

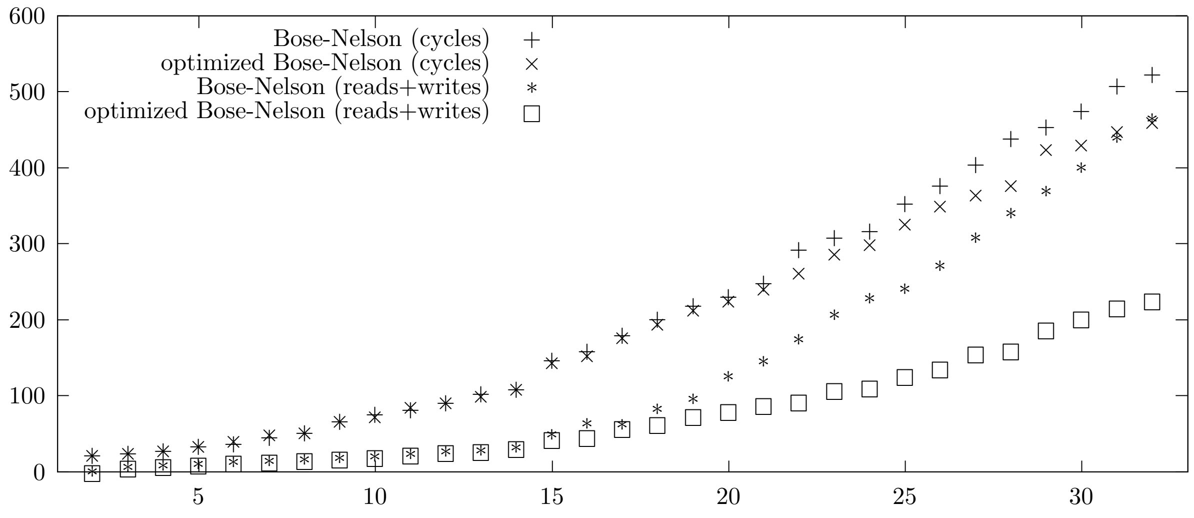

Fig. 17. Number of cycles and memory accesses (read + write) standard and optimized Bose–Nelson networks

6.2. Batcher sorting networks

Looking at Figures 12 and 13, it may not be obvious that the idea used for optimizing Bose–Nelson sorting networks is applicable to Batcher's construction. However, this family of networks is also constructed by a similar recursive algorithm, the main difference being that the recursive sorting networks are built on the *odd* and *even* channels, rather than on the top half and low half. As a consequence, the merger network used is substantially different and, as remarked above, uses all channels up to the last (parallel) step.

The recursive construction of Batcher sorting networks is as follows:

1. build a sorting network on the $\lceil \frac{n}{2} \rceil$ odd-numbered channels;
2. build a sorting network on the $\lfloor \frac{n}{2} \rfloor$ even-numbered channels;
3. juxtapose the previous networks and append an $\left(\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil\right)$-merger

where the merger in the last step is Batcher's odd-even merger [Bat68]. Again, recursively applying this construction with the same base cases as before allows us to construct sorting networks on any number of

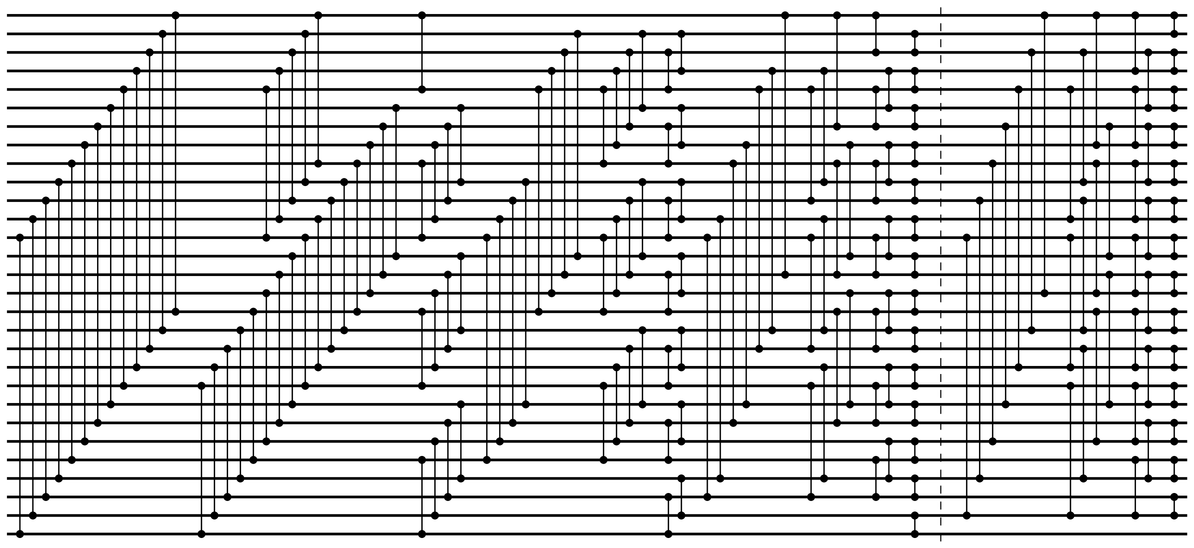

Fig. 18. Batcher's sorting network on 29 channels. The dashed line separates the two first recursive calls from the final merge step.

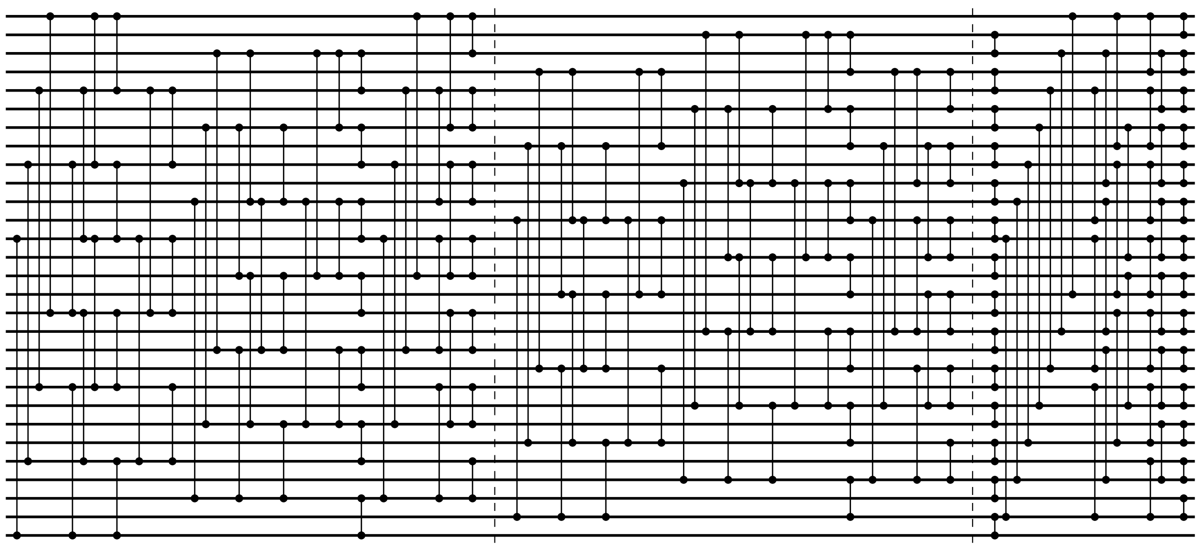

Fig. 19. Batcher's sorting network on 29 channels optimized for 13 registers – all recursive calls are appended, rather than juxtaposed. The dashed lines separate the two first recursive calls; note how the two first blocks are nearly identical, but shifted from the odd to the even channels.

channels. Furthermore, these networks can be built with maximal parallelism by an imperative algorithm found in [Knu73] (Section 5.2.2, Algorithm M).

In order to optimize register access, we can again append, rather than juxtapose, recursive calls that build sorting networks on more than 13 channels. The only difference with respect to Bose & Nelson's construction is that these networks will now act on interleaved, rather than adjacent, sets of channels. Figures 18 and 19 illustrate the corresponding variants of Batcher's sorting network on 29 channels. Figure 20 shows that systematically rearrangeing the comparators in this fashion achieves performance gains comparable to those described for Bose–Nelson sorting networks. Furthermore, Batcher sorting networks now systematically outperform those ones, even in the previous anomalous cases of e.g. 21 inputs.

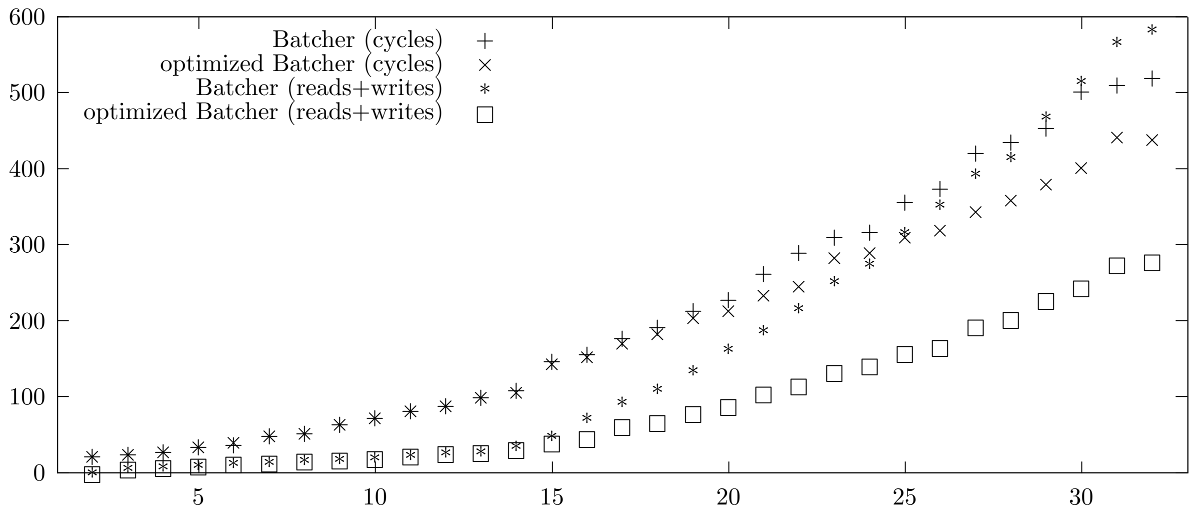

Fig. 20. Number of cycles and memory accesses (read + write) standard and optimized Batcher networks

6.3. Achieving optimality

We also tested our optimizations to the Hibbard family of sorting networks [Hib63], obtaining similar results. Hibbard's algorithm directly generates the Bose–Nelson sorting networks when the number of channels n is a power of 2; for other values of n, it generates the network obtained from the Bose–Nelson sorting network on $2^{\lceil \log_2 n \rceil}$ channels by removing all comparators acting on at least one channel with index greater than n.

However, our methodology for optimizing networks for a given number of available general-purpose registers only works well on networks that have a recursive structure, as it is only in those cases that we are able to identify systematically smaller subnetworks that can be replaced by more efficient ones. Therefore, it is not clear whether and how these can be used for e.g. the optimal-size sorting network on 16 channels. We point out that the optimized variant of Batcher's sorting network on 16 channels actually performs as well as this network.

The interesting aspect is that neither Batcher's nor Bose & Nelson's recursive algorithms rely on the structure of the smaller sorting networks for their correctness. In other words, we can combine *any* two sorting networks on n and m channels by means of an (n, m)-merger to obtain a sorting network on $n + m$ channels. This allows us to construct an optimally-performing sorting network (for r registers) as follows:

1. let $k_1 = \lceil \frac{n}{2} \rceil$
   (a) if $k_1 < \min(r, 16)$, construct an optimal-size sorting network on the k_1 odd-numbered channels
   (b) else, recursively construct a sorting network on the k_1 odd-numbered channels
2. let $k_2 = \lfloor \frac{n}{2} \rfloor$
   (a) if $k_2 < \min(r, 16)$, construct an optimal-size sorting network on the k_2 even-numbered channels
   (b) else, recursively construct a sorting network on the k_2 even-numbered channels
3. append the two previous networks and a (k_1, k_2)-merger.

To demonstrate this construction, we are using Batcher's recursive construction, as it outperforms Bose & Nelson's (both in the setting of this and of the previous subsection), and thus gives smaller execution times; but the same adaptation indeed also works for the latter networks.

We benchmark the performance of these networks again by using kernel level minimum benchmarking. The results are summarized in Figure 22 for Batcher's construction and Figure 21 for Bose–Nelson's. The slight decrease in the number of comparators gives a performance advantage in several instances and, as expected, always performs at least on par with the optimized variants of the sorting networks.

Using the optimized variant of Batcher's construction as a base case improves the performance of Quick-

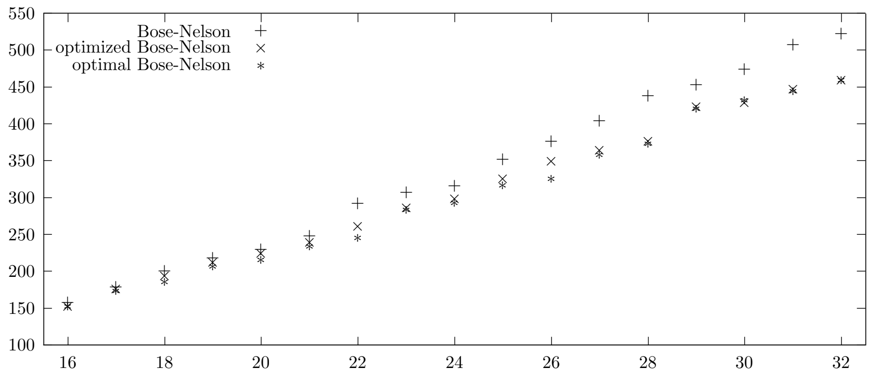

Fig. 21. Number of cycles for Bose–Nelson sorting networks: standard construction, optimized as in Section 6.1, and using size-optimal networks for $n \leq 13$.

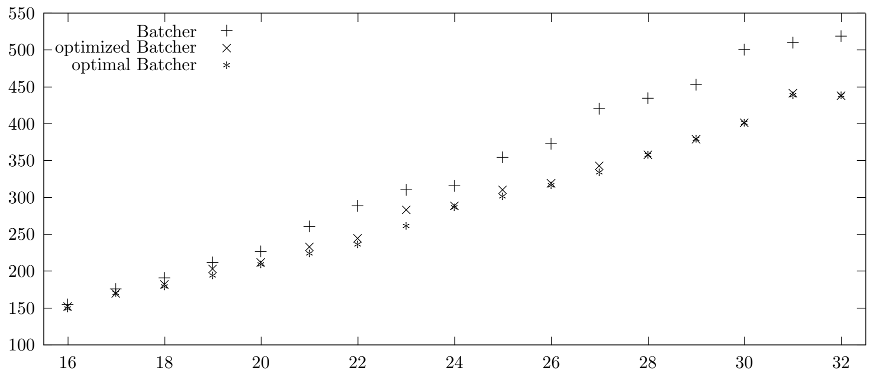

Fig. 22. Number of cycles for Batcher sorting networks: standard construction, optimized as in Section 6.2, and using size-optimal networks for $n \leq 13$.

sort further by up to 5% w.r.t. the results presented in Figure 8. The real-world impact of each of these contributions to the performance of Quicksort is summarized in Figure 23, which depicts average execution times when using for base case: unrolled insertion sort; Batcher sorting networks; Bose–Nelson sorting networks; and the optimized variants of these. Table 2 details individual percentage gains for particular cases.

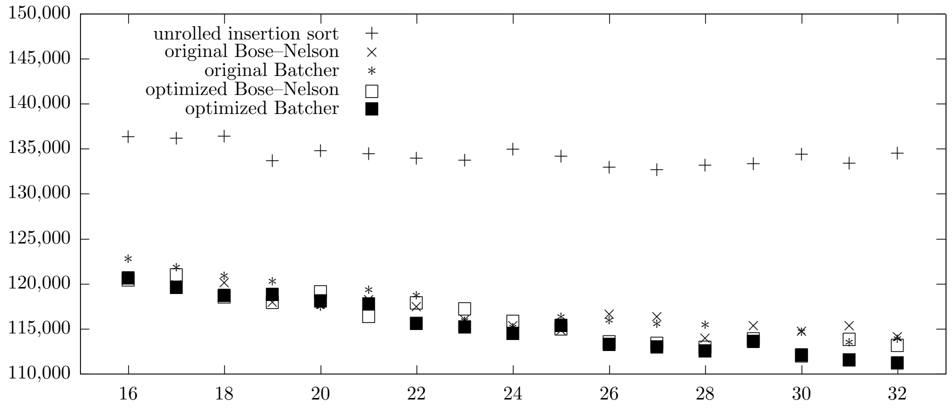

Fig. 23. Performance of Quicksort for different base cases with limit sizes varying from 16 to 32.

limit	Bose–Nelson		Batcher	
size	original	optimized	original	optimized
16	**11.5%**	**11.5%**	10.0%	**11.5%**
17	12.0%	11.0%	10.6%	**12.2%**
18	11.9%	**13.0%**	11.4%	**13.0%**
19	**11.7%**	11.6%	10.0%	11.1%
20	12.6%	11.5%	**12.9%**	12.4%
21	12.0%	**13.3%**	11.3%	12.4%
22	12.3%	11.9%	11.4%	**13.7%**
23	13.3%	12.2%	13.4%	**13.8%**
24	14.5%	14.0%	14.6%	**15.1%**
25	**14.4%**	14.1%	13.3%	14.0%
26	12.3%	14.4%	12.8%	**14.8%**
27	12.3%	14.4%	12.9%	**14.8%**
28	14.4%	15.1%	13.3%	**15.5%**
29	13.5%	14.5%	14.7%	**14.8%**
30	14.6%	16.5%	14.7%	**16.6%**
31	13.5%	14.5%	14.9%	**16.4%**
32	15.1%	15.8%	15.3%	**17.3%**

Table 2. Improvement of using different sorting networks as a base case of Quicksort w.r.t. unrolled insertion sort.

We observe that, as the limit size increases, the optimized Batcher sorting networks systematically outperform the other constructions. This tendency becomes more pronounced for limit sizes above 26, which is to be expected as we effectively have 13 available general-purpose registers. Choosing a limit size of 32 and applying all the techniques developed in this work, we see an effective real-world improvement of 17.3% in the performance of Quicksort with insertion sort as a base case.

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 [LCC14]. A further empirical comparison between our implementation and the one described in [FAN07] (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 [FAN07]. Furthermore, we can easily incorporate characteristics of the hardware as parameters to the recursive constructions we use to build sorting networks whose performance is optimized for that architecture. 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, we showed that the relevant measure is actually a combination of small size, enough parallelism and a block structure of the network that minimizes register spilling.

Acknowledgements. This work was supported by the Israel Science Foundation, grant 182/13, and by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247. The authors also wish to thank the anonymous referees of the original paper [CCFNSK15] for their suggestions and comments.

References

[Bat68] K.E. Batcher. Sorting networks and their applications. In *AFIPS Conference Proceedings*, volume 32, pages 307–314. Thomson Book Company, 1968.

[BB11] S.W.A.-H. Baddar and K.E. Batcher. *Designing Sorting Networks: A New Paradigm.* Springer, 2011.

[BN62] R.C. Bose and R.J. Nelson. A sorting problem. *J. ACM*, 9(2):282–296, 1962.

[BZ14] 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.

[CCFFSK14] 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.

[CCFFSK16] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *Journal of Computer and System Sciences*, 82(3):551–563, 2016.

[CCFNSK15] Michael Codish, Luís Cruz-Filipe, Markus Nebel, and Peter Schneider-Kamp. Applying sorting networks to synthesize optimized sorting libraries. In Moreno Falaschi, editor, *LOPSTR*, volume 9527 of *LNCS*, pages 127–142. Springer, 2015.

[CCFSK15a] 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.

[CCFSK15b] 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.

[EGT10] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS '10*, pages 13–22. ACM, 2010.

[EM15] 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.

[FAN07] T. Furtak, J.N. Amaral, and R. Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *SPAA*, pages 348–357. ACM, 2007.

[FFY05] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools.* Morgan Kaufman, 2005.

[Gam11] J. Gamble. Algorithm::networksort 1.30, 2011. Available from `http://cpansearch.perl.org/src/JGAMBLE/Algorithm-Networksort-1.30/lib/Algorithm/Networksort.pm`.

[GZ06] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: optimal parallel sorting on stream architectures. In *IPDPS*. IEEE, 2006.

[Hib63] T.N. Hibbard. A simple sorting algorithm. *J. ACM*, 10(2):142–150, 1963.

[Hoa62] C.A.R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[LCC14] 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.

[Pao10] G. Paoloni. How to benchmark code execution times on intel® IA-32 and IA-64 instruction set architectures. White paper 324264-001, Intel Corporation, September 2010.

[Par91] I. Parberry. A computer-assisted optimal depth lower bound for nine-input sorting networks. *Mathematical Systems Theory*, 24(2):101–116, 1991.

[Sed77] R. Sedgewick. The analysis of quicksort programs. *Acta Inf.*, 7:327–355, 1977.

[SF96] R. Sedgewick and P. Flajolet. *An introduction to the analysis of algorithms.* Addison-Wesley-Longman, 1996.

[SRU99] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond.* Springer, 1999.

[SW11] R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, 2011. 4th Edition.