Received: 14 February 2020 | Revised: 2 October 2020 | Accepted: 3 October 2020

DOI: 10.1002/spe.2922

RESEARCH ARTICLE

WILEY

Engineering faster sorters for small sets of items

Timo Bingmann | **Jasper Marianczuk** | **Peter Sanders**

Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany

Correspondence
Timo Bingmann, Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany.
Email: bingmann@kit.edu

Funding information
Open access funding enabled and organized by Projekt DEAL

Abstract

Sorting a set of items is a task that can be useful by itself or as a building block for more complex operations. That is why a lot of effort has been put into finding sorting algorithms that sort large sets as efficiently as possible. But the more sophisticated and complex the algorithms become, the less efficient they are for small sets of items due to large constant factors. A relatively simple sorting algorithm that is often used as a base case sorter is insertion sort, because it has small code size and small constant factors influencing its execution time. We aim to determine if there is a faster way to sort small sets of items to provide an efficient base case sorter. We looked at sorting networks, at how they can improve the speed of sorting few elements, and how to implement them in an efficient manner using conditional moves. Since sorting networks need to be implemented explicitly for each set size, providing networks for larger sizes becomes less efficient due to increased code sizes. To also enable the sorting of slightly larger base cases, we adapted sample sort to Register Sample Sort, to break down those larger sets into sizes that can in turn be sorted by sorting networks. From our experiments we found that when sorting only small sets of integers, the sorting networks outperform insertion sort by a factor of at least 1.76 for any array size between six and 16, and by a factor of 2.72 on average across all machines and array sizes. When integrating sorting networks as a base case sorter into Quicksort, we achieved far less performance improvements over using insertion sort, which is probably due to the networks having a larger code size and cluttering the L1 instruction cache. The same effect occurs when including Register Sample Sort as a base case sorter for IPS^{4}o. But for x86 machines that have a larger L1 instruction cache of 64 KiB or more, we obtained speedups of 12.7% when using sorting networks as a base case sorter in std::sort, and of 5%–6% when integrating Register Sample Sort as a base case sorter into IPS^{4}o, each in comparison to using insertion sort as the base case sorter. In conclusion, the desired improvement in speed could only be achieved under special circumstances, but the results clearly show the potential of using conditional moves in the field of sorting algorithms.

KEYWORDS

base case sorting, sample sort, sorter, sorting, sorting algorithm, sorting networks

1 INTRODUCTION

1.1 Motivation

Sorting, that is rearranging elements of an input set into a specific order, is a fundamental algorithmic challenge. At universities around the globe, basic sorting algorithms are taught in introductory computer science courses as examples for theoretical analysis of algorithms. We learn that bubble sort and insertion sort have quadratic asymptotic running time, Quicksort expected $\mathcal{O}(n \log n)$ but worst-case quadratic time, and merge sort always runs in $\mathcal{O}(n \log n)$ time. These algorithms are analyzed by the number of comparisons they require, both asymptotically, up to constant factors, and sometimes also exactly for small input sizes n. But later, practical experience shows that pure theoretical analysis cannot tell the whole story. In real applications and on real hardware factors such as average cases, cache effects, modern CPU features like speculative execution, and of course constant factors actually matter, a lot. Any well-founded choice on which sorting algorithm to use (for a particular use case) should be influenced by all factors.

The usual playing field for developing new sorting algorithms is to sort a *large number* of items as quickly as possible. We will call these *complex* sorting algorithms and many follow the divide-and-conquer paradigm. However, the algorithmic steps for large sets in these complex sorters do not perform well when sorting small sets of items, because they have good asymptotic properties but larger constant factors that become more important for the small sizes. However also sorting small inputs can be relevant for performance. This happens for the *base case* of complex sorting algorithms or when many sorting problems have to be solved. The latter case for example occurs when the adjacency lists of a graph should be sorted by some criterion in order to support some greedy heuristics. For example, the *heavy edge matching* heuristics[1] repeatedly looks for the heaviest unmatched edge of a vertex.

The most common choice for sorting small inputs is insertion sort, which has a worst-case running time of $\mathcal{O}(n^2)$, but small constant factors that make it suitable to use for small n. When the sorter is executed many times, the total running time does add up to a substantial part of the computing time of an application. In this article we therefore attempt to optimize or replace this quadratic sorting algorithm at the heart of most complex sorters and other applications.

Put plainly: *what is the fastest method to sort up to 16 elements on actual modern hardware?*

We investigate two approaches: first to optimize sorting networks and the second to adapt sample sort to small numbers of items. When optimizing sorting networks, the most important aspects are how to execute the conditional swap operations on modern CPUs and which sorting network instances to use: the best/shortest known ones, or recursive locality-aware constructions such as Bose–Nelson's.[2] Optimizing sorting networks has previously been addressed by Codish, Cruz–Filipe, Nebel, and Schneider–Kamp[3] in 2017, but we go much further into the hardware details of the conditional swap operations and use hand-coded assembly employing conditional move instructions. We focused on using "simple" nonvectorized instructions, because they are available on a wider range of architectures. For slightly larger sets of items (e.g., up to 256), we present Register Sample Sort (RSS), which is an adaptation of Super Scalar Sample Sort[4] to use the registers in the CPU for splitters. As items we focus on sorting pairs of an integer as key and an associated reference pointer or value. This enables the sorting of items with complex payload values. Integers keys fit into a register and can easily be compared using a simple instruction. Our results can be applied directly to other keys like float and doubles, but not necessarily to larger objects. However, some of our sorting network implementations are fully generic and can be translated to any data type.

For our experimental evaluation we used four machines: two with Intel CPUs, one with an AMD Ryzen, and a Rockchip RK3399 ARM system on a chip. It turns out that sorting networks with hand-coded conditional move assembly instructions perform much better (a factor 2.4–5.3 faster) and have a much smaller variance in running time than insertion sort. However, when integrating sorting networks into Quicksort we unexpectedly saw only a speedup of 7%–13% depending on the machine and base algorithm. We attribute this to the larger code size of sorting networks and thus L1 instruction cache misses. Register Sample Sort is also faster than insertion sort for 256 items: up to a factor 1.4 over `std::sort`. We then integrated both sorting networks and Register Sample Sort into IPS^{4}o, a fast complex comparison-based sorter by Axtmann et al..[5] Our experiments validate the authors measurements that IPS^{4}o is some 40%–60% faster than `std::sort`, and were able to show that our better base case sorters improve this by another 1.3% on Intel CPUs, 5% on AMD CPUs, and 7% on the ARM machine. The ARM machine has higher variance in running time but less outliers than the Intel and AMD ones. We conclude that the larger code size of sorting networks is a disadvantage, and that Intel and AMD's instruction scheduling and pipelining units are good at accelerating insertion sort. For ARM machines, better algorithms however make a difference because the CPUs are simpler.

This article is based on the bachelor's thesis of Jasper Marianczuk.[6]

1.2 | Overview of the article

Section 2 is dedicated to sorting networks: Section 2.1 starts with the general basics of sorting networks and inline assembly code. After that, we look at different ways of implementing sorting networks efficiently in C++ in Section 2.2.

In Section 3 we regard Super Scalar Sample Sort and develop an efficient modified version for sets with 256 elements or less by holding the splitters in general purpose registers instead of an array. The resulting algorithm is called Register Sample Sort.

Section 4 discusses the results and improvements of using sorting networks we achieved in our experiments, measuring the performance of the sorting networks and Register Sample Sort individually, and also including them as base cases into Quicksort and IPS^{4}o. After that we conclude the results of this article in Section 5.

1.3 | Related work

Sorting is a large and well-studied topic in computer science and the many results fill entire volumes[7,8] of related work. We can focus only on the most relevant here.

Sorting networks are considered in more detail in the following section. The most well-known sorting networks are Bose–Nelson's recursive construction,[2] Batcher's bitonic and odd-even merge sorts,[9] and the AKS (Ajtai, Komlós, and Szemerédi's) sorting network.[10] The publication closest to our work is the paper by Codish et al.,[3] who optimized sorting networks for locality and instruction-level parallelism.

Sorting networks were used by many authors to vectorize base-case sorting using single-instruction multiple-data (SIMD) instructions in wide registers, as part of larger vectorized sorters. Inoue et al. proposed AA-sort[11] first in 2007, which is both multicore parallelized and SIMD vectorized but processes larger blocks of items which are then merged. Furtak, Amaral, and Niewiadomski used code generation and developed three SIMD implementations,[12] for x86-64's SSE2 and G5's AltiVec instruction sets. Xiaochen, Rocki, and Suda proposed two algorithms for AVX-512 SIMD,[13] one sorting 16 items in one register, and one sorting 31 items in two registers, both based on bitonic sorting networks. More recently, Bramas came up with an alternative way to used bitonic sorting networks to sort up to 16 elements with AVX-512 SIMD instructions.[14] Hou, Wang, and Feng developed an entire framework[15] which automatically translates abstract sorting networks into vectorized code for different instruction sets. For example, they support Bose–Nelson, Batcher's bitonic, and Hibbard[16] networks and can output AVX, AVX2, and IMCI code. In 2019, Yin et al. presented a both parallelized and vectorized sorting algorithm,[17] which contains a 16×16 SIMD sorting kernel based on bitonic sorting for blocks of 256 items. Beyond SIMD, sorting networks were also used in various FPGA hardware.[18,19]

Compared with previous work, we consider how to optimize sorting networks without special vectorized instructions or wide registers. We also consider the variance or standard deviation of running time of our implementations, which is interesting for more accurately predicting code execution time. None of the referenced papers consider this interesting aspect of sorting networks and branchless execution.

2 | SORTING NETWORKS

2.1 | Introduction

Sorting algorithms can be classified into two groups: those of which the comparison behavior depends on the input and those of which the behavior is *not* influenced by the particular configuration of the input. Examples of the former are Quicksort,[20] merge sort, insertion sort,[7,21] while the latter are called *data-oblivious*.

One example of data-oblivious sorting algorithms are *sorting networks*. A sorting network operates on a fixed number n of *channels* enumerated from 1 to n, each representing one input variable, and connections between the channels, called *comparators*. When two channels are connected by a comparator then the values are compared and conditionally swapped: if the channel with the lower number holds a value that is greater than the value of the channel with the higher number then the values in the variables are exchanged.

The comparators are given in a fixed order that determines the sequence of executing these *conditional swaps*, such that in the end the channels contain *a permutation* of the original input, and the values held by the channels are in *nondecreasing order*. Sorting networks are data-oblivious because all comparisons are always performed in the same order, no matter which permutation of an input is given.

The two most important metrics to quantify sorting networks are their *size* and *depth*. A network's size refers to the *total number of comparators* it contains, and a network's depth describes the *minimal number of levels* a network can be divided into.

Each individual comparator is located on a singleton *level*. When two comparators do not share a channel and are consecutive, then they can be combined into a common *level*. Inductively, multiple consecutive comparators can be merged into a level, if their channels are not shared with any other comparator in the level. Most importantly for performance, since all the comparators in a level are independent from one another, they can be executed in parallel.

2.1.1 | Networks in practice

There are many methods to come up with sorting networks which correctly order any input.

- **Best known networks:** Sorting networks with proven *optimal sizes* and *optimal depths* are known only for small numbers of input channels. To date, the optimal depth is known only for n up to seventeen,[7,22-25] while the optimal size only for n up to 10.[24] For example, a network for 10 elements with optimal size 29 has depth nine, and one with optimal depth seven has size 31.[7,24] For larger networks individual upper and lower bounds on size or depth are known. These optimal networks were initially optimized by hand and nowadays are searched for with the help of computers and evolutionary algorithms.[26]
- **Recursively generated networks:** Besides elaborate algorithms searching for optimal networks, there are also much simpler methods to generate correct (but nonoptimal) networks. The most commonly used paradigm is recursive divide-and-conquer: split the input into two parts, sort each part recursively, and merge the two parts together in the end. Representatives for this kind of approach are the constructions of Nelson and Bose[2] and the algorithms by Batcher.[9]

  Bose and Nelson split the input sequence into first and second half, while Batcher partitions it into elements with an even index and elements with an odd index. The advantage of these recursive networks over the specially optimized ones is that they can easily be created even for large network sizes. While the generated networks may have more comparators than the best known networks, the number of comparators in a network acquired from either Bose–Nelson or Batcher of size n has an upper bound of $\mathcal{O}(n\,(\log n)^2)$. This was improved by Ajtai, Komlós, and Szemerédi[10] to optimal $\mathcal{O}(\log n)$ depth with the much-cited AKS sorting network, which however have prohibitively high constants and are thus unusable for small n.

Sorting networks are customarily depicted by using horizontal lines for the channels, and undirected vertical connections between these lines for the comparators. A network by Bose and Nelson for six elements is illustrated in this manner in Figure 1, and Figure 2 shows a network with optimal size for 10 elements. The dotted vertical lines indicate the nine levels in the network.

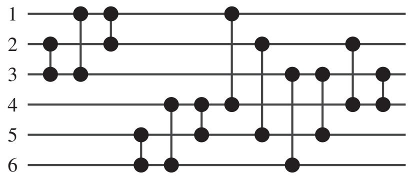

FIGURE 1 Recursively generated sorting network by Bose and Nelson for six elements

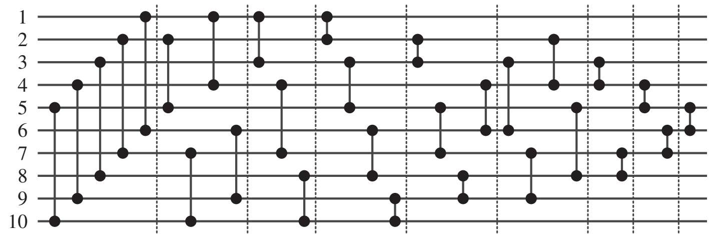

FIGURE 2 Sorting network with optimal size for 10 elements

2.1.2 | Improving the speed of sorting through sorting networks

The central question to our investigation in this section is how sorting networks can improve the sorting speed on a set of elements (on average), if they cannot take any shortcuts for "good" inputs, like an insertion sort that would leverage an already sorted input and do one comparison per element. The answer to this question is avoiding penalties introduced by *branching*. Because the compiler and CPU know in advance which comparisons are going to be executed in which order, the control flow does not contain conditional branches, which in particular gets rid of expensive branch mispredictions and allows instruction level (superscalar) parallelism. On uniformly distributed random inputs, the chances that any number is smaller than another is 50% on average, making branches unpredictable. In the case of insertion sort that means not knowing in advance with how many elements the next one has to be compared until it is inserted into the right place (on average it would be half of them).

Even though with sorting networks the compiler knows in advance when to execute which comparator, implementing the conditional-swap operation in a naive way (as seen in Section 2.1.3) the compiler might still generate branches. In that case, the sorting networks are no faster than insertion sort, or even slower. Hence, we investigated the use of assembly code in this article.

Another interesting use of sorting networks may be in the field of cryptography and security-focused applications. The time it takes to sort with non-data-oblivious algorithms (e.g., Quicksort) may introduce side channels allowing an attacker to infer the order of the input elements.

2.1.3 | Conditional swap

For sorting networks, the basic operation is to compare two values against each other and swap them if they are in the wrong order (the "smaller" element occurs after the "larger" one in the sequence). This *conditional-swap* operation can be implemented straight-forwardly in C++ with an `if` and a `swap`:

```cpp
void ConditionalSwap(Type& left, Type& right) {
  if (right < left) { std::swap(left, right); }
}
```

Here `Type` is a template and can be instantiated with any type that implements the < operator. As suggested by Codish et al.,[3] the same piece of code can be rewritten like this:

```cpp
void ConditionalSwap2(Type& left, Type& right) {
  Type temp = left;
  if (right < temp) { left = right; }
  if (right < temp) { right = temp; }
}
```

At first glance it appears as if there are now two conditional branches. But since the statement executed when the condition is true now only consists of a single assignment each, these can be expressed in x86-architecture with a *conditional move* instruction. In AT&T syntax (see Section 2.1.4), a conditional move (`cmov a,b`) will write the value of register `a` into register `b`, if a condition is met. If the condition is not met, no operation takes place (but still taking the same number of CPU cycles as the move operation would have). Since the address of the next instruction no longer depends on the previous condition, the control flow now does not contain branches. This avoids the large cost of *branch mispredictions* which require the execution pipeline of the CPU to be flushed and many speculative operations to be undone. The only downside of conditional moves is that they may take longer to evaluate than a normal move instruction on certain architectures, and can only be executed when the comparison has performed and its result is available. They can also only operate on register entries.

When the elements to be swapped are plain integers, some compilers do generate code with conditional moves for those operations while others default to jump branches. In the experiments in Section 4 we consider pairs of an unsigned 64-bit integer key and an unsigned 64-bit reference value, which could be a pointer to data or an address in an array. To force the usage of conditional moves, we investigated use of *inline assembly*,[27] a feature of gcc and other compilers that allows the programmer to specify small amounts of assembly code to be inserted into the generated machine code. This technique and the notation is further explained in the following Section 2.1.4.

2.1.4 | Inline assembly code

In this section we introduce the reader to a relatively obscure feature in modern C/C++ compilers: inline assembly code. We will use it in the next section to hard-code conditional swap operations and thus bypass the compiler's optimizations for these operations.

The machine instructions executed by the CPU are also called *assembly code*, which can be expressed as the actual op-codes or as human-readable text. There are two competing conventions for the textual representation: the Intel syntax or MASM syntax and the AT&T syntax.

The main differences are the parameter order and operand size. In Intel syntax the destination parameter is written first, then the source of the value: `mov dest,src`, while the size of the operands need not be specified. In AT&T syntax on the other hand, the source parameter is written first, followed by the destination: `movq src,dest`. The size of the operand must be appended to the instruction: "`b`" (byte = 8 bit), "`l`" (long = 32 bit), "`q`" (quad-word = 64 bit). In this article only the *AT&T syntax* will be used, because it is used internally by gcc.

The gcc and clang C++ compilers have a feature that allows the programmer to write assembly instructions in between regular C++ code, called "inline assembly" (`asm`).[27] This inline code consists of a continuous piece of assembly code, together with a specification of how it should interact with the surrounding C++ code. This specification communicates to the compiler what happens inside the `asm` block and consists of a definition for *input* and *output* variables and a list of *clobbered registers*. Gcc does not optimize the given assembly statements, they are added into the generated assembly code verbatim and translated to machine code by the GNU Assembler.

A variable listed as output means that the value will be modified, a clobbered register is one where gcc cannot assume that the value it held before the `asm` block will be the same as after the block. In this article, the clobbered registers will almost always be the conditional-codes registers ("`cc`"), which include the carry flag, zero flag, and the signed flag, which are modified during a compare-instruction. This way of specifying the input, output, and clobbered registers is also called *extended asm*.

Taking the code from Section 2.1.3, and assuming `Type = uint64_t`, the statement

```
uint64_t temp = left;
if (right < temp) {
  left = right;
}
```

can now be written with extended inline x86 assembly as

```
uint64_t temp = left;
__asm__(
  "cmpq %[temp],%[right]\n\t"                       // compare right and temp
  "cmovbq %[right],%[left]\n\t"                     // left = right, if
  : [left] "=&r"(left)                              // output variables
  : "0"(left), [right] "r"(right), [temp] "r"(temp) // input variables
  : "cc"                                            // clobbered registers
);
```

In extended `asm`, one can define C++ variables as input or output operands. For inputs the compiler will assign a register if it has the "`r`" modifier and load the value into it. For outputs, a register is allocated and the value is written back to the given variable after the `asm` block or used immediately in further steps. The names in square brackets are symbolic names only valid in the context of the assembly instructions and independent from the names in the C++ code before. The link between the C++ names and the symbolic names is defined in the input and output declarations, which may or may not be the same.

For conditional moves it is important to properly declare the input and output variables, because they perform a task that is a bit unusual: an output variable may or may not be overwritten. In the case of the output register for `left` used above, two things must apply: if the condition is false, it must hold the value of `left`, and if the condition is true, it must hold the value of `right`.

For optimizations purposes, the compiler might reduce the number of registers used by placing the output of one operation into a register that previously held the input for some other operation. To prevent this, the declaration for the output `[left] "=&r"(left)` has the "`&`" modifier added to it, meaning it is an "early clobber" register and that no other input can be placed in that register. In combination with `"0"(left)` in the input definition, the same register is additionally tied to an input, such that the previous value of `left` is loaded beforehand, in case the conditional move is not executed. Because `left` was already declared as output, instead of giving it a new symbolic name we tie it to the input by referencing its index `"0"` in the output list.

The "`=`" in the output declaration only means that this register will be written to. Any output needs to have the "`=`" modifier. Each assembly instruction is postfixed with "`\n\t`" because the strings are appended into a single long line by the C++ compiler and the line breaks separate instructions for the assembler.

The `cmov` instruction is postfixed with "`b`" in this example, which stands for *below*, such that the move is executed if `right` is below `temp` (unsigned comparison `right < temp`). Apart from *below* we will also see *not equal* ("`ne`") and *carry* ("`c`") as a postfix in further examples. Furthermore, both the `cmp`and the `cmovb` are postfixed with "`q`" (quad-word) to indicate that the operands are 64-bit values.

When a subtraction (`minuend` – `subtrahend`) is performed and `subtrahend` is larger than `minuend` (interpreted as unsigned numbers), the operation causes an underflow which results in the carry flag being set. The carry flag can be used as a condition by itself (postfix "`c`") and it also influences condition checks like *below*. This property of the comparison setting the carry flag will be used in Section 3.1.

2.2 | Implementation of sorting networks

We now consider how to actually implement sorting networks for performance.

2.2.1 | Providing the network frame

We collected the following sorting networks for small inputs: For sizes of up to 16 elements the best known networks were taken from John Gamble's Website[28] and are size-optimal up to 10 elements. The Bose–Nelson networks have been generated using the instructions from their paper.[2] We did not use any Batcher odd-even network because Codish et al.[3] showed that there was no difference between Batcher and Bose–Nelson in practice.

For sizes of eight and below the best and generated networks have the same amount of comparators and levels. For sizes larger than eight the generated networks are at a disadvantage because they have more comparators and/or levels. As a trade-off their recursive structure makes it possible to leverage a different trait: *locality*. Instead of optimizing them to sort in as few levels as possible, we can first sort the first half of the set, then the second half, and then apply the merger. Thus chances are higher that all $\frac{n}{2}$ elements of the first half may fit into the processor's general purpose registers. To determine if there is an achievable speedup, the networks were generated optimizing for (a) *locality* and (b) *parallelism*.

Furthermore, we investigated two implementations of locality-optimizing recursively defined Bose–Nelson networks: one where the entire network is rolled out for each input size individually, and a second were the functions of each input size may call smaller sorters for parts of the input. The first unrolled variant retains the name *locality*, while we call the second one a *recursive* implementation.

Examples of networks for 16 elements can be seen in Figures 3, 4, and 5.

All networks are implemented such that they have an entry method that takes a pointer to an array `A` and an array size `n` as input and delegates the call to the specific method for that number of elements. To measure different implementations for the conditional swaps, the sorting networks are templated with both the swap and the item type.

Our approach differs from most previous work[3] in the type of elements that were sorted. While most experiments measured the sorting of plain `int`s, which are usually 32-bit sized integers, we made the decision to sort elements that

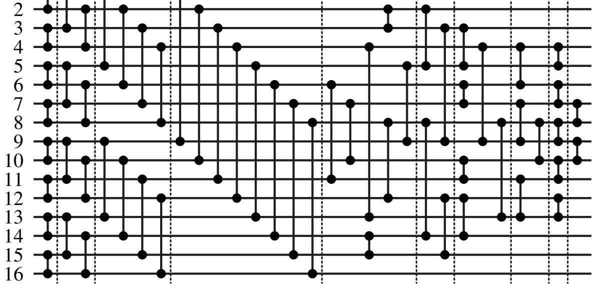

FIGURE 3 Sorting network with optimal size for 16 elements

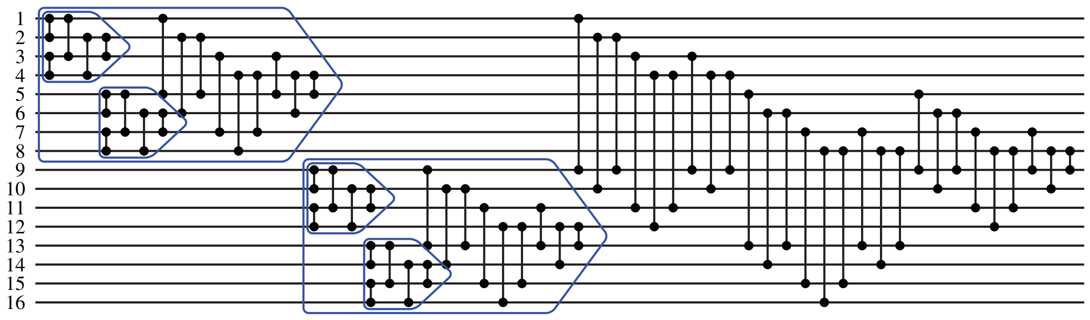

FIGURE 4 Bose–Nelson network for 16 elements optimizing locality (unrolled or recursive implementations). The subnetworks marked blue sort eight and four items, respectively [Color figure can be viewed at wileyonlinelibrary.com]

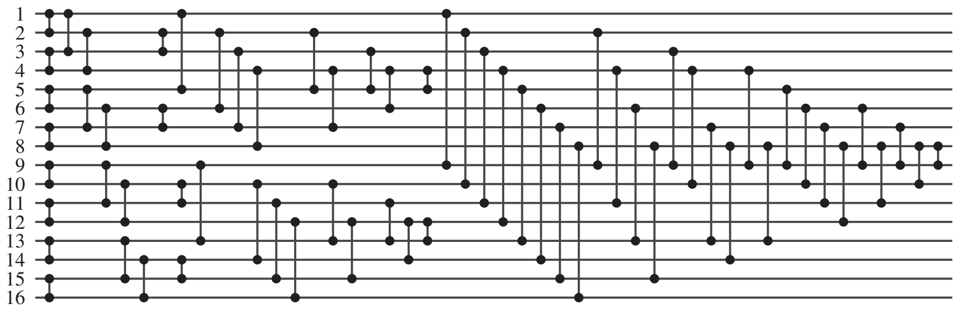

FIGURE 5 Bose–Nelson network for 16 elements optimizing parallelism

consist of a 64-bit integer key and a 64-bit integer reference value. This enables not only sorting of numbers but also of complex payloads, by giving a pointer or an array index as the reference value. This was implemented by creating a `struct` that contain a key and reference value each, having the following structure:

```
struct SortableRef {
  uint64_t key, ref;
}
```

We also defined the operators >, >=, ==, <, <=, and != for usability. Other keys than integers are also possible: floats or doubles also fit into a single register and can be compared using a single CPU instruction.

2.2.2 | Implementing the conditional swap

`ConditionalSwap` is implemented as a templated method like this:

```
template <typename Type>
inline void ConditionalSwap(Type& left, Type& right) {
  // body
}
```

Our goal is thus to find the best instructions to implement this method, either by convincing the compiler to produce good code or by writing inline assembly instructions directly. The following variants will represent the body of one specialization of the template function for a specific struct. Each of them was given a three to four letter abbreviation to name them in the results. We implemented the following approaches:

`ISwp` **(if and std::swap)**:	This is the simplest way of writing the conditional-swap operation, without any inline assembly as a C/C++ `if` statement with a `std::swap` (Figure 6). We already saw this code in Section 2.1.3.
`TCOp` **(ternary conditional operators)**:	This is an alternative portable C/C++ implementation without inline assembly. It uses two *ternary conditional*

operators ("? :") and a temporary variable (Figure 7). The goal of this method is to try to convince the compiler to generate conditional moves, which seems to happen more often for the ternary operator.

`Tie` **(std::tie with std::tuple):** This is another portable C/C++ implementation using an assignment of a pair of variables using *structured bindings*. In the current C++ versions this can be expressed with `std::tuple`and `std::tie` (Figure 8). Again this is portable, and the compiler is tasked to generate efficient code from it.

`JXhg` **(jmp and xchg)**: This is a straight-forward inline assembly version implemented using a comparison, a conditional jump, and two exchange (`xchg`) instructions. If `right_key` is above `left_key`or equal to it, the `xchg` instructions are skipped (**j**ump **a**bove or **e**qual, `jae`). `xchg` swaps the contents of two registers. The "`%=`" directive generates a unique label for each `asm` instance (Figure 9).

`4Cm` **(four cmovs and temp variables)**: This is the shortest assembly implementation with a comparison and four conditional move (`cmov`) operations. Since we need to swap a key-reference pair of 64-bit integers, we need four `cmov`s and two temporary variables (Figure 10). The left key and value are stored in the temporary variables, unconditionally. Then the two keys are compared in assembly and the result is stored in the flags register. If the items need to swapped, then the four `cmov`s first move the right item to the left, and then the temporary item into the right. The issue with this version is that there is an unconditional copy of the left pair into the temporary variables.

`6Cm` **(six cmovs and temp variables)**: In `4Cm` the temporary variables are unconditionally assigned outside the assembly block. This is unnecessary if the condition is false, such that we proposed this variant with six `cmov`s (Figure 11). As before, the keys are compared in assembly, but this time, six `cmov`s are necessary to swap the items: first copy the left pair into temporary variables, then replace the left with the right pair, and then move the temporaries into the right. If the items need not be swapped, then the temporary variable is not changed.

`4CmS` **(four cmovs split and temp variables)**: Since the C++ compiler cannot reorder operations inside inline assembly blocks, we attempted to split these such that the compiler can *interleave* load/store operations from multiple consecutive conditional-swaps to avoid memory stalls. There is obviously a limit to this reordering, which requires the `asm` blocks to be declared `volatile` such that these stay in order among themselves. The `cmov`s are the same as in the `4Cm` version, except they are split into different blocks (Figure 12).

`2CPm` **(move pointers with two cmovs)**: The following two variants are based on applying `cmov` to *pointers* to the `SortableRef`instead of moving a pair of key and value. The idea is based on assembly code generated by the clang compiler for the `ConditionalSwap2` method in Section 2.1.3. First we declare two pointers to the items, and copy the left item into a temporary variable.

After the comparison, the right pointer is copied into the left pointer using a `cmov`, and then the temporary pointer into the right pointer. After the assembly block, the pointers are used to actually swap the key and value pairs using assignments in C/C++, unconditionally. We split the assembly into two blocks and let the compiler optimize and interleave load and store operations (Figure 13).

`2CPp` **(move pointers with two cmovs and predicate)**: Instead of performing the comparison inside the `asm` block, which requires knowledge of the datatype of the key, it can also be done using an indicator predicate. This predicate is simply an integer variable that is passed into the assembly code. Combining predicates with the pointer swapping technique from `2CPm` delivers this variant which can operate on keys with custom comparators and any data payload (Figure 14). The goal is simply to see how much slower the external predicate is.

```
if (right < left)
    std::swap(left, right);
```

FIGURE 6 `ISwp` (if and std::swap) [Color figure can be viewed at wileyonlinelibrary.com]

```
bool r = (right < left);
auto temp = left;
left = r ? right : left;
right = r ? temp : right;
```

FIGURE 7 `TCOp` (ternary conditional operators) [Color figure can be viewed at wileyonlinelibrary.com]

```
std::tie(left, right) = (right < left)
    ? std::make_tuple(right, left)
    : std::make_tuple(left, right);
```

FIGURE 8 `Tie` (std::tie with std::tuple)

```
__asm__(
    "cmpq %[left_key],%[right_key]\n\t"
    "jae %=f\n\t"
    "xchg %[left_key],%[right_key]\n\t"
    "xchg %[left_ref],%[right_ref]\n\t"
    "%=:\n\t"
    : [left_key] "=&r"(left.key),
      [right_key] "=&r"(right.key),
      [left_ref] "=&r"(left.ref),
      [right_ref] "=&r"(right.ref)
    : "0"(left.key), "1"(right.key),
      "2"(left.ref), "3"(right.ref)
    : "cc"
);
```

FIGURE 9 `JXhg` (jmp and xchg) [Color figure can be viewed at wileyonlinelibrary.com]

```
uint64_t tmp = left.key;
uint64_t tmp_ref = left.ref;
__asm__(
    "cmpq %[left_key],%[right_key]\n\t"
    "cmovbq %[right_key],%[left_key]\n\t"
    "cmovbq %[right_ref],%[left_ref]\n\t"
    "cmovbq %[tmp],%[right_key]\n\t"
    "cmovbq %[tmp_ref],%[right_ref]\n\t"
    : [left_key] "=&r"(left.key),
      [right_key] "=&r"(right.key),
      [left_ref] "=&r"(left.ref),
      [right_ref] "=&r"(right.ref)
    : "0"(left.key), "1"(right.key),
      "2"(left.ref), "3"(right.ref),
      [tmp] "r"(tmp), [tmp_ref] "r"(tmp_ref)
    : "cc"
);
```

FIGURE 10 `4Cm` (four cmovs and temp variables) [Color figure can be viewed at wileyonlinelibrary.com]

FIGURE 11 6Cm (six cmovs and temp variables) [Color figure can be viewed at wileyonlinelibrary.com]

```
uint64_t tmp;
uint64_t tmp_ref;
__asm__ (
    "cmpq %[left_key],%[right_key]\n\t"
    "cmovbq %[left_key],%[tmp]\n\t"
    "cmovbq %[left_ref],%[tmp_ref]\n\t"
    "cmovbq %[right_key],%[left_key]\n\t"
    "cmovbq %[right_ref],%[left_ref]\n\t"
    "cmovbq %[tmp],%[right_key]\n\t"
    "cmovbq %[tmp_ref],%[right_ref]\n\t"
    : [left_key] "=&r"(left.key),
      [right_key] "=&r"(right.key),
      [left_ref] "=&r"(left.ref),
      [right_ref] "=&r"(right.ref),
      [tmp] "=&r"(tmp), [tmp_ref] "=&r"(tmp_ref)
    : "0"(left.key), "1"(right.key), "2"(left.ref),
      "3"(right.ref), "4"(tmp), "5"(tmp_ref)
    : "cc"
);
```

FIGURE 12 4CmS (four cmovs split and temp variables) [Color figure can be viewed at wileyonlinelibrary.com]

```
uint64_t tmp = left.key;
uint64_t tmp_ref = left.ref;
__asm__ volatile (
    "cmpq %[left_key],%[right_key]\n\t"
    :
    : [left_key] "r"(left.key),
      [right_key] "r"(right.key)
    : "cc"
);
__asm__ volatile (
    "cmovbq %[right_key],%[left_key]\n\t"
    : [left_key] "=&r"(left.key)
    : "0"(left.key), [right_key] "r"(right.key)
    :
);
__asm__ volatile (
    "cmovbq %[right_ref],%[left_ref]\n\t"
    : [left_ref] "=&r"(left.ref)
    : "0"(left.ref), [right_ref] "r"(right.ref)
    :
);
__asm__ volatile (
    "cmovbq %[tmp],%[right_key]\n\t"
    : [right_key] "=&r"(right.key)
    : "0"(right.key), [tmp] "r"(tmp)
    :
);
__asm__ volatile (
    "cmovbq %[tmp_ref],%[right_ref]\n\t"
    : [right_ref] "=&r"(right.ref)
    : "0"(right.ref), [tmp_ref] "r"(tmp_ref)
    :
);
```

In Sections 4.2 and 4.3 we report on our experiments with these conditional swap operations.

We also tried a variant with XOR (see also Codish et al.[3]), where the left item is first stored in a temporary variable and then conditionally overwritten with the right in case the values have to be swapped. The right item is then calculated using an XOR of the right, the original left, and current left (which was possibly replaced with the right). In case no swap occurs, the original left and current left are the same and cancel out. In case a swap occurs, the current left and right are the same and cancel out, leaving the original left as the new value of right. We experimented with this variant, but it turned

```
Type* left_pointer = &left;
Type* right_pointer = &right;
Type temp = left;
__asm__ volatile(
    "cmpq %[tmp_key],%[right_key]\n\t"
    "cmovbq %[right_pointer],%[left_pointer]\n\t"
    : [left_pointer] "=&r"(left_pointer)
    : "0"(left_pointer),
      [right_pointer] "r"(right_pointer),
      [tmp_key] "r"(temp.key),
      [right_key] "r"(right.key)
    : "cc"
);
left = *left_pointer;
left_pointer = &temp;
__asm__ volatile(
    "cmovbq %[left_pointer],%[right_pointer]\n\t"
    : [right_pointer] "=&r"(right_pointer)
    : "0"(right_pointer),
      [left_pointer] "r"(left_pointer)
    :
);
right = *rightPointer;
```

FIGURE 13 2CPm (move pointers with two cmovs) [Color figure can be viewed at wileyonlinelibrary.com]

```
Type* left_pointer = &left;
Type* right_pointer = &right;
Type temp = left;
int cmp_result = (int)(right < temp);
__asm__ volatile(
    "cmp $0,%[cmp_result]\n\t"
    "cmovneq %[right_pointer],%[left_pointer]\n\t"
    : [left_pointer] "=&r"(left_pointer)
    : "0"(left_pointer),
      [right_pointer] "r"(right_pointer),
      [cmp_result] "r"(cmp_result)
    : "cc"
);
left = *left_pointer;
left_pointer = &temp;
__asm__ volatile(
    "cmovneq %[left_pointer],%[right_pointer]\n\t"
    : [right_pointer] "=&r"(right_pointer)
    : "0"(right_pointer),
      [left_pointer] "r"(left_pointer)
    :
);
right = *rightPointer;
```

FIGURE 14 2CPp (move pointers with two cmovs and predicate) [Color figure can be viewed at wileyonlinelibrary.com]

out to be much slower (by a factor of 2–3) in preliminary results. Another variant with XOR would be to use $x = x \oplus y$, $y = x \oplus y$, $x = x \oplus y$. This however requires a conditional branch and is thus only a more complicated replacement for a swap.

3 | REGISTER SAMPLE SORT

After initial positive experimental results of our investigation of sorting networks for small inputs, we decided to turn to *sample sort* for slightly larger inputs. Our preliminary experiments showed that sorting networks were indeed faster, but they also required a lot of instructions, leading to large instruction decoding times and L1 cache misses. Another motivation was that the base cases issued by *In-Place Parallel Super Scalar Samplesort*(IPS^{4}o)[5] were considerably larger than 16 items. Hence, instead of extending sorting networks beyond 16 elements, we close this gap by providing a completely different basic algorithm for small to medium size inputs: *Register Sample Sort* (RSS). Our new algorithm is based on Super Scalar Sample Sort (S^4)[4] and can reduce large base case sizes down to blocks of 16 or less in an efficient manner. The central idea is to place the splitters not into an array, as described in the original S^4, but to hold them in general purpose registers for the whole duration of the element classification.

3.1 | Basic sequential sample sort

Sample sort[29] is a sorting algorithm that follows the divide-and-conquer principle. The input is split into k disjoint intervals of the total ordering defined by $k+1$ splitters $s_0, \dots, s_k$. These are chosen by first selecting a sample subset S of $a \cdot k$ items with oversampling factor a and sorting the sample S. Afterward the splitters $\{s_0, s_1, \dots, s_{k-1}, s_k\} = \{-\infty, S_a, S_{2a}, \dots, S_{(k-1)a}, \infty\}$ are taken equidistant from S. Oversampling is used to get better splitters to achieve more evenly sized partitions, trading balance for the additional time to select and sort the larger sample.

Given the splitters, all elements e_i are then *classified* by placing them into buckets b_j, where $j \in \{1, \dots, k\}$ and $s_{j-1} < e_i \le s_j$. If k is a power of 2, this placement can be achieved by viewing the splitters as a perfect binary tree, with $s_{k/2}$ being the root, all s_l with $l < k/2$ representing the left subtree and those with $l > k/2$ the right one. To classify an element, one must only traverse this binary tree in logarithmic time, resulting in a binary search on $k+1$ elements instead of a linear one.[4]

Quicksort[20] can be seen as a specialization of sample sort with fixed parameter $k = 2$. Sample sort is very popular for sorting large amounts of items on distributed systems,[30-32] on GPUs,[33] and also for strings.[34,35]

3.2 | Implementing register sample sort for medium-sized sets

In this section we explain how to implement sample sort using registers in a CPU. The main issue is that, other than memory, registers *cannot be accessed using an index* on most popular architectures.

3.2.1 | Traversing A tree in registers

In implementations of S^4 splitters are organized in memory as a perfect binary search tree $t := [s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \dots]$. Traversal of the splitter tree is then performed using an index j: the children of splitter j are at positions $2j$ and $2j+1$. If an element is smaller than t_j, it must be compared with t_{2j}, otherwise to t_{2j+1}, in the next step. This allows an easy branchless traversal of the tree by multiplying j with two and conditionally incrementing it. But this way of accessing the splitters does not work when they are placed in registers, because we cannot access registers by index.

Our solution is to create *an unconditional complete copy* of the left subtree, and to use `cmov` operations to *conditionally overwrite it* with the complete right subtree should the element be greater than the root node. The next comparison is then performed against the root of the copied tree that now contains the correct splitters. This way we traverse the tree until a leaf node is reached. The copying of subtrees is obviously expensive and only viable for small trees. For three splitters this requires one conditional move, and for seven splitters it requires three conditional moves after the first comparison and one additional after the second comparison, per element.

3.2.2 | Calculating the final bucket index

However, after finding the correct splitters to compare to, we are left with yet another problem: how to determine the index of the bucket the element is to be placed into. In S^4[4] this bucket index is calculated directly from the index of the last splitter. For Register Sample Sort, we choose an approach similar to creating this index using the correlation between binary numbers and the tree-like structure of the splitters. We view the splitters not as a binary tree but just as a list where the middle of the list represents the root node of the tree, its children being the middle element of the left and the middle element of the right list.

If an element e_i is larger than the first splitter $s_{k/2}$ (with $k-1$ being the number of nonsentinel splitters, s_0 and s_k are handled implicitly), it must be placed in a bucket b_j with $j \geq \frac{k}{2}$ (assuming 0-based indexing for b). This also means that the index of that bucket, represented as a binary number, must have its bit at position $l := \log \frac{k}{2}$ set to 1. Hence the result of the comparison $(e_i > s_{k/2})$ can be interpreted as an integer (1 for true, 0 for false) and added to j. If it was not the final comparison, j is then multiplied by 2 (meaning its bits are shifted left by one position). This means the bit from the first comparison makes its way “left” in the binary representation while the comparison traverses down the tree, and so forth with the other comparisons. After traversing the splitter tree to the end, e_i will have been compared with the correct splitters and j will hold the index of the bucket that e_i belongs into. A similar method is used in S^4 when calculating the index during tree traversal. These operations can be implemented without branches by making use of the way the comparisons are performed:

At the end of Section 2.1.4 we explained that when comparing (unsigned) numbers (which is nothing but a subtraction), and the `subtrahend` being greater than the `minuend`, the operation causes an underflow and the carry flag is set. We also notice that when converting the result of the predicate $(e_i > s_{k/2})$ to an integer value, the integer will be 1 for `true` and 0 for `false`. So in assembly code, we can compare the result from evaluating the predicate to the value 0: `cmp %[result],%[zero]` where `zero` is just a register that holds the value 0. This trick is needed because the `cmp` instruction needs the second operand to be a register. This will execute 0 − `result`, which underflows for the predicate returning true. This way we can postfix the `cmov` needed for moving the next splitters with “`c`” checking for a set carry flag. The second instruction we make use of is the *rotate carry left* (`rcl`) instruction, which performs a *rotate left* instruction on j, but includes the carry flag as an additional bit after the least significant bit of the integer. This exactly takes the predicate result and puts it at the bottom of j, with the previous content being shifted one to the left beforehand. That means it performs two necessary operations at once.

3.2.3 | Putting things together: Determining parameters and pseudocode

With the previous two challenges of classification solved, we can now design the parameters for our Register Sample Sort.

As with S^4, when processing items from the input we can *interleave classification* of multiple elements, allowing for all the registers in the machine to be used. This additional parameter is called *block size*.

The main constraint on the parameters of Register Sample Sort is the *number of registers* in the CPU. The keys of the splitters (since we only need a splitter's key for classifying an element) must be small enough to fit into a general purpose register. Needing more than one register per key would mean running out of registers more quickly and also spending extra time to conditionally move the splitter keys around. For three splitters the number of registers needed for block sizes 1 to 5 are shown in Table 1. We can see that the trade-off for classifying multiple elements at the same time is the amount of registers needed.

TABLE 1 Number of registers required by register sample sort with three or seven splitters

	Three splitters block size					**Seven splitters block size**				
	1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Splitters	3	3	3	3	3	7	7	7	7	7
Buckets pointer	1	1	1	1	1	1	1	1	1	1
current element index	1	1	1	1	1	1	1	1	1	1
Element count	1	1	1	1	1	1	1	1	1	1
Index	1	2	3	4	5	1	2	3	4	5
Predicate result	1	2	3	4	5	1	2	3	4	5
`splitterX`	1	2	3	4	5	3	6	9	12	15
Sum	9	12	15	18	21	15	20	25	30	35

If we were to use seven splitters instead of three, the number of registers required for classifying just one element at a time would go up to 15. Furthermore, if we get recursive subproblems with just slightly over 16 items, classifying into eight buckets would be greatly inefficient, resulting in many empty buckets. This is why we decided to only use *exactly three splitters* for this particular sorter.

Instead of inline assembly from our implementation, we present pseudocode of the classification for `block size = 1` in Algorithm 1. The index j is here called `index`, and the temporary subtree consists in this case of one splitter, which we gave the name `splitterX`. For our branchless implementation we used `cmovne` together with a `test` instruction to move the right hand splitter into `splitterX` and just assigned the integer result of the comparison to index in the first step (lines 9 and 11). At the second and last level of classification no more movement of splitters is required, so instead of performing a comparison against the predicate's result and using `rcl`, we can just shift `index` left by one position and add the predicate's result to it (line 13). Alternatively we could use a bit-wise `OR` or `XOR` after the shift, which would have the same result. But we decided that adding the predicate result was more readable.

For sorting the splitter sample, the same sorting method can be used as for the base case, which in our case is to use a sorting network from Section 2.

Algorithm 1. Register Sample Sort classification pseudocode for three splitters

```
1  void RegisterSampleSortClassification(Type* array, unsigned array_size) {
2    Type [splitter0, splitter1, splitter2] = determineSplitters();
3    Type* [$b_0$,$b_1$,$b_2$,$b_3$] = allocateBuckets(array_size);
4    for (int i = 0; i < array_size; ++i) {
5     int index = 0;
6     int cmp_result = (int)(splitter1 < array[i]);
7     splitterx = splitter0;          // copy left tree into splitterx
8     if (cmp_result) {              // first and only decision in a tree with three splitters
9      splitterx = splitter2;         // overwrite using a cmovne
10    }
11    index = cmp_{r}esult;
12    cmp_{r}esult = (int)(splitterx < array[i]);
13    index = (index << 1) + cmp_{r}esult;
14    $b_{index}$.push_{b}ack(array[i]);
15   }
16 }
```

4 | EXPERIMENTAL RESULTS

In this section we report on four sets of experiments with sorting networks and Register Sample Sort. The first are pure performance measurements of sorting networks, either sorting the same array repeatedly or sorting a larger array containing many independent small problems. The second experiments are on the performance of Quicksort with sorting networks as base case, the third on finding good parameters for Register Sample Sort, and the last on integrating Register Sample Sort and sorting networks into IPS^{4}o.

4.1 | Parameters, machines, inputs, methodology

4.1.1 | Machines and compiler

We used four different machines to perform the measurements, including Intel, AMD, and ARM CPUs. Their labels and hardware properties can be seen in Table 2. In the table "I" and "D" refer to dedicated L1 instruction and data caches. While the AMD Ryzen's L3 cache has a total size of 16 MiB, it is divided into two 8 MiB caches that are exclusive to four

TABLE 2 Hardware properties of the machines used in our experiments

Machine name	Intel-2650	Intel-2670	Ryzen-1800X	RK3399
CPU	2 x Intel XeonE5-2650 v2	2 x Intel XeonE5-2670 v3	AMD Ryzen 1800X	Rockchip RK3399
	8-core, 2.6 GHz	12-core, 2.3 GHz	8-core, 3.6 GHz	ARM Cortex-A53 | -A72
				4×1.5 GHz | 2×2.0 GHz
RAM	128 GiB DDR3-1600	128 GiB DDR4-2133	32 GiB DDR4-2133	4 GiB LPDDR4
L1 cache (KiB per core)	32 I + 32 D (8-way)	32 I + 32 D (8-way)	64 I + 32 D (4/8-way)	32 I + 32 D | 48 I + 32 D
L2 cache (KiB per core)	256 (8-way)	256 (8-way)	512 (8-way)	512 | 1024
L3 cache (MiB total)	20	30 (20-way)	16 [8] (8-way)	–
Linux Distribution	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04	Armbian (Debian) Buster
Compiler	gcc 7.3.0	gcc 7.3.0	gcc 7.3.0	gcc 8.3.0

cores each. Since all measurements were done on a single core, the L3 cache size in brackets is the one available to the program.

For compiling our experiment code we used the gcc C++ compiler in version 7.3.0 or 8.3.0 with `-O3` and `-march=native` flags. We did not experiment with LLVM or other compilers because this increases the parameter space by another dimension.

The measurements were done with only essential processes running on the machine apart from the measurement. To prevent the process from being swapped to another core during execution it was run with the command "`taskset 0x1`" as prefix, which pins it to the first core.

4.1.2 | Inputs: Random numbers

In order to measure the time needed to sort some data, we first have to generate data. For all experiments the data type consisted of a pair of one 64-bit unsigned integer key and one 64-bit unsigned integer reference value. Items were generated as uniformly distributed random numbers by a lightweight implementation of the `std::minstd_rand` generator from the C++ `<random>` library that works as follows: First a `seed` is set, taken, for example, from the current time. When a new random number is requested, the generator calculates $\texttt{seed} = (\texttt{seed} \cdot 48271) \bmod 2147483647$ and returns the current `seed`. The numbers generated by this generator does not use all 64 bits available, which however has no effect on the experiment results.

For each measurement i, a new $\texttt{seed}_i$ is taken from the current time. The same $\texttt{seed}_i$ is then set before the execution of each sorter, to provide all sorters with the same random inputs.

We only experimented with random inputs. In future work, "easy" inputs such as sorted and reverse sorted inputs, and others should also be included.

4.1.3 | Measurement of cycles with perf_event

The actual measurement was done via linux's `perf_event` interface that allows to do fine-grained measurements using hardware counters. We measured the number of CPU *cycles* spent on sorting. This also means that our results do not depend on clock speeds (e.g., when overclocking), but only on the CPU's architecture. On the ARM machine RK3399 we resorted to simply measuring time, because the CPU cycles event interface was not available.

4.1.4 | Checking results: Permutation check

For compilation, the optimization flag `-O3` was used to achieve high optimization and speed. This also meant that, without using the sorted data in some way, the compiler would deem the result unimportant and skip the sorting altogether. That is why after each sort, to generate a side-effect, the set is checked for two properties: That it is sorted, and that it is a permutation of the input set. The first can easily be done by checking for each value that it is not greater than the value before it.

The permutation check is done probabilistically: At design time, a (preferably large) prime number p is chosen. Before sorting, $v = \prod_{i=1}^{n}(z - a_i) \mod p$ is calculated for an arbitrary number z and values $a = \{a_1, \dots, a_n\}$. To check the permutation after sorting and obtaining $a' = \{a1', \dots, an'\}$, $w = \prod_{i=1}^{n}(z - a_i') \mod p$ is calculated. If $v \neq w$, a' cannot be a permutation of a. If $v = w$, we claim that a' is a permutation of a.

To minimize the chances of a' not being a permutation of a, but v being equal to w, $v = 0$ was disallowed in the first step. If v is zero, z is incremented by one and the product calculated again, until $v \neq 0$.

We do this kind of check because in each iteration the input array is overwritten with new random numbers, and using an easier algorithm to sort the array afterward and then checking for equality would include copying the unsorted array, sorting, and checking, which adds extra time and variation to a measurement consisting of many of these iterations. The permutation check on the other hand is branch-free, and the sorted check is effectively branch-free for any sorted array, because the compiler is told to expect the sorted condition to be true, and will speculatively execute the branch in which it is, not leading to any flushing of the pipeline. This way we can run the sorted and permutation check in a second measurement with the same number of iterations, and subtract that time from the original measurement, giving only the time needed for sorting.

4.1.5 | Multiple source file compilation

Initially, the experiments were a single source file (`.cpp`) with an increasing amount of headers that were all included in that single file. This was mostly due to the fact that templated methods cannot be placed in source files because they need to be visible to all including files at compile time. The increasing amount of code and the many different templates, however, brought the compiler to a point where it took over a minute to compile the program. The problem we encountered was that the compiler apparently allots less time to optimization the longer the compilation runs on a source file. Hence, once our experiment program became reasonably large, the optimizations became poor. We saw measurements being slower for no apparent reason. To solve that problem, we used code generation to create source files that contain a smaller amount of methods that initiate part of a measurement in a wrapper method. From the main source file we thus only need to call the correct wrapper methods to perform the measurements, and this way we were able to achieve results that were more stable and reproducible. In our case that means one file for the OneArrayRepeat experiment (Section 4.2), one for ArrayInRow (Section 4.3), one for QuickSort (Section 4.4), one for SampleSort (Section 4.5), and an individual file for each configuration of the IPS^{4}o measurement (Section 4.6).

4.1.6 | Measurement loops and warm-up

As the sorting algorithm on a small number of items is very fast, we measured many iterations and divided by the number of repetitions. Since in this scenario we have to generate new random inputs for each iteration, we decided to first measure the three steps (a) data generation, (b) sorting, and (c) checking, and then measure only (a) data generation and (c) checking on the same inputs. By subtracting the two running times we receive the pure sorting time. We call this measurement loop *OneArrayRepeat*.

More details on the method are now discussed: At the beginning a random seed is selected and the generator initialized. To reduce the chance of cache misses at the beginning of the measurement, one warm-up run of random generation, sorting, and checking is performed before starting the clock. After the warm-up round, the array is then filled, sorted, and checked `numberOfIterations` times. The random generator is not reseeded each round. After the main measurement, a second phase is run with the same data. But this time only the generation of the random numbers and the checking is measured to later subtract the time from the previously measured one, resulting in the time needed for the sorting alone. To generate and check the same input again, the random generator is reseeded with the previously selected seed. Obviously, the checking of the unsorted data (usually) fails but it has to performed to measure the time. Hence, we devised a simulated checking method, which does the exact same comparisons and permutation calculations, but ignores the result.

Nevertheless there are random nondeterministic fluctuations in running time even on the same code. And since both measurement parts are subject to their own deviation, it can occasionally happen that the second measurement takes longer than the first, leading to negative running times for sorting. We received negative values more often for the sorters with small array sizes, where the sorting itself takes relatively little time compared with the random generation and sorted checking. The negative times show up as outliers in the results.

The measurement loop itself is repeated `numberOfMeasures` times for each `arraySize` that is sorted.

For the measurements shown in Section 4.3 the method was slightly modified. The goal was to better highlight cache- and memory-effects by creating one longer array that does not fit into the CPU's L3-cache and then sorting disjoint short subsequences of size `arraySize` in order. We call this modified measurement loop *ArrayInRow*.

Because we can create the whole array at the beginning, we can generate the numbers before and check for correct sorting after measuring, hence there is no need to do a second measurement like in the OneArrayRepeat benchmark. As in the prior benchmark, one warm-up round is performed prior to running the measured loop.

For ArrayInRow, instead of giving a `numberOfIterations` parameter to indicate how often the sorting is to be repeated, we provide a `numberOfArrays` value that prescribes how many arrays of size `arraySize` are to be created contiguously. This parameter is chosen for each `arraySize` in a way that (`numberOfArrays` · `arraySize`) does not fit into the L3 cache of the machine the measurement is performed on.

4.1.7 | Generating plots

Due to the high number of dimensions in the measurements (machine the measurement is run on, type of sorting network, conditional swap implementation, array size) the results could not always be plotted two-dimensionally. We used box plots where applicable to show more than just an average value for a measurement. The box encloses all values between the first quartile q_1 and third quartile q_3. The line in the middle shows the median. Further the inter-quartile-range $\overline{q}$ is calculated as the distance between first and third quartile. The lines (called whiskers) left and right of the boxes extend to the smallest value greater than $q_1 - 1.5\overline{q}$ and the greatest value smaller than $q_3 + 1.5\overline{q}$, respectively. Values below these ranges are called *outlier* and shown as individual dots.

4.2 | Sorting sets of 2–16 items

In this and the following subsection we report on experiments comparing sorting algorithms and conditional-swap implementations. For the details about the different sorters and swaps refer to Section 2.2.

The algorithm variants in the tables and figures are labeled in an abbreviatory way such as "`SN BN-R 4CmS`". The abbreviations are composed from the following three parts:

1. First, `IS` or `SN` indicate if the algorithm is insertion sort or a sorting network.
   For `IS` we evaluated four variants: `Def` is a textbook implementation with array indices, `POp`uses pointers as iterators, `STL` is copied from gcc's STL implementation, and `AIF` is `Def` with an additional check if the next item is smaller than the first.
2. In case of sorting networks, the algorithms are labeled as `Best` networks or Bose–Nelson networks (`BN`). The Bose–Nelson sorters are further available optimized for *locality* (`BN-L`), *parallelism* (`BN-P`), or written as *recursively* called sorting functions (`BN-R`) (see Section 2.2.1).
3. And as last component, the name of the conditional swap implementation is appended for sorting networks (see Section 2.2.2 for their abbreviations).

As an example consider "`SN BN-R 4CmS`". This implementation is a Bose–Nelson sorting network generated using recursively constructed functions which perform the conditional-swap using the `4CmS` variant.

Table 3 shows the binary x86 code size of each insertion sort and sorting network variant (size 2–16). We determined their binary x86 code size by compiling the code (with `-O3` optimization) and then disassembling the object file and identifying which functions therein belong to the algorithm. In Table 3 we also included the code size of `std::sort`, Register Sample Sort (RSS), and IPS^{4}o, each excluding any subsorters.

To determine the fastest algorithm variant, we ran the OneArrayRepeat experiment with the following parameters on all machines and algorithms: `numberOfIterations` = 100, `numberOfMeasures` = 500, and `arraySize` ∈ {2, … , 16}.

TABLE 3 Size in bytes of binary x86 assembly code of sorters generated by gcc (with optimization)

Algorithm	Size	Algorithm	Size	Algorithm	Size	Algorithm	Size	Algorithm	Size
`IS Def`	91	`SN Best ISwp`	17,936	`SN BN-L ISwp`	19,264	`SN BN-P ISwp`	19,216	`SN BN-R ISwp`	12,880
`IS POp`	122	`SN Best Tie`	33,328	`SN BN-L Tie`	34,400	`SN BN-P Tie`	35,920	`SN BN-R Tie`	19,072
`IS STL`	188	`SN Best JXhg`	12,080	`SN BN-L JXhg`	10,160	`SN BN-P JXhg`	12,880	`SN BN-R JXhg`	6144
`IS AIF`	200	`SN Best 4Cm`	19,008	`SN BN-L 4Cm`	17,456	`SN BN-P 4Cm`	20,640	`SN BN-R 4Cm`	9584
`std::sort`	1283	`SN Best 4CmS`	19,472	`SN BN-L 4CmS`	17,312	`SN BN-P 4CmS`	20,704	`SN BN-R 4CmS`	10,720
RSS `332`	1280	`SN Best 2CPm`	24,064	`SN BN-L 2CPm`	25,360	`SN BN-P 2CPm`	25,696	`SN BN-R 2CPm`	14,096
IPS^{4}o	42,689	`SN Best 2CPp`	26,912	`SN BN-L 2CPp`	28,416	`SN BN-P 2CPp`	28,736	`SN BN-R 2CPp`	14,640

Table 4 shows our results as a ranking of the algorithms by geometric mean, individually and across all machines, of the slowdown relative to the best on the particular machine. Table 5 summarizes the fastest sorting network over the fastest insertion sort implementation. Further results are shown in the Appendix Tables A1, A2, A3, and A4; there we show the name of the sorter and the average number of cycles per iteration, over the total of all measurements, for all machines. The algorithm that performed best in a column is marked in bold font, and for each column the slowdown relative to the best in that column was calculated. For each row the geometric mean "GeoM" is shown over these relative slowdown values and the mean is used to rank the algorithms.

The tables show that the implementations without conditional branches (`4Cm`, `4CmS`, `2CPm`, `2CPp`) and those with (`ISwp`, `Tie`, `JXhg`) are clearly separated by rank in the overall result, the former occupy the lower share of the ranks, while the latter all the higher ranks. To improve readability, the variants `TCOp`and `6Cm` are omitted. Variant `6Cm` was similar to the other variants without conditional branches, and `TCOp` was close to the results of `JXhg`. We omitted them because they are probably executed very similarly by the processors.

Comparing the machines in Table 4, we see that our hypothesis that the `4CmS` conditional swap is better than the `4Cm` was shown to be true for machines Intel-2650 and Intel-2670, but not for machines Ryzen-1800X and RK3399. We believe this is due to the first machines having better support for interleaving load and store operations. We also see in Table 4 that the first five ranks have very similar geometric means, which implies the Bose—Nelson networks (`BN-L` and `BN-R`) can compete (due to their locality) with the optimized networks (`Best`) that have fewer comparators.

Figures 15, 16, 17, and 18 show box plots of the CPU cycle measurement for array size 8 on each machine. These plots highlight that the best sorting network implementations are not only faster on average, but that their distribution is almost entirely faster than any of the insertion sort implementations, together with a lower variance. As in the tables, the variants `TCOp` and `6Cm` are omitted to improve readability. Furthermore, one outlier was removed from data set of machine Intel-2670 for the `SN BN-P 4CmS` sorter with value −228.55 such that the plot has a scale similar to those of the other two machines, to improve comparability. It is remarkable that on the RK3399 machine the variance of the sorting networks is much higher than on the other platforms. However, this may be an effect due to measuring nanoseconds wallclock time on the machine (due to lack of performance counters or support for them) versus CPU cycles on the others.

To see a trend in increasing array size, we chose a few conditional-swap implementations that do best for more than one network and array size on all machines. These series increase as expected from the $\mathcal{O}(n\log^2 n)$ asymptotic growth rate of the sorting networks. Their average sorting times with OneArrayRepeat can be seen in Figure 19. For better readability, we omitted the Bose–Nelson parallel (`BN-P`) networks in these plots. The figures underline the results already shown in the tables: the `4Cm` and `4CmS` implementations have good performance and are almost always faster on average than insertion sort (apart from `arraySize = 2` on machine RK3399).

These results indicate that there is potential in using sorting networks, showing that the best insertion sort is slower by a factor of 1.79 on average on machine RK3399, up to a factor of 4.47 on average on machine Ryzen-1800X (see Table 5 for details). The issue with the OneArrayRepeat experiment is that the same memory area is sorted over and over again, which is rarely a use case when sorting a base case. Because of this, the results probably reflect unrealistic conditions regarding cache accesses and cache misses. To get closer to realistic base case sorting, the next section regards the ArrayInRow pattern.

TABLE 4 Ranking of sorting algorithms by geometric mean of relative slowdowns for the OneArrayRepeat experiment across all machines

Sorter	Overall		Intel-2650		Intel-2670		Ryzen-1800X		RK3399	
	Rank	GeoM	Rank	GeoM	Rank	GeoM	Rank	GeoM	Rank	GeoM
SN BN-L 4Cm	1	1.09	4	1.12	1	1.08	1	1.08	3	1.07
SN Best 4Cm	2	1.12	6	1.14	5	1.10	5	1.19	2	1.04
SN BN-P 4Cm	3	1.13	8	1.17	8	1.14	4	1.18	1	1.03
SN BN-R 4Cm	4	1.15	3	1.11	4	1.09	3	1.15	4	1.25
SN BN-L 4CmS	5	1.18	2	1.09	3	1.09	2	1.14	5	1.45
SN Best 4CmS	6	1.23	1	1.09	2	1.09	6	1.23	15	1.57
SN BN-P 4CmS	7	1.27	5	1.13	7	1.14	7	1.24	18	1.64
SN BN-R 4CmS	8	1.33	7	1.15	6	1.12	8	1.29	21	1.89
SN Best 2CPp	9	1.53	12	1.35	13	1.43	12	1.86	6	1.50
SN BN-P 2CPp	10	1.55	13	1.38	14	1.45	13	1.87	7	1.54
SN Best 2CPm	11	1.55	9	1.19	9	1.18	10	1.62	23	2.54
SN BN-P 2CPm	12	1.58	10	1.21	10	1.22	9	1.61	27	2.62
SN BN-L 2CPm	13	1.68	11	1.32	11	1.31	11	1.82	24	2.56
SN BN-L 2CPp	14	1.68	15	1.54	15	1.61	15	2.10	11	1.55
SN BN-R 2CPp	15	1.81	16	1.67	16	1.73	16	2.38	16	1.57
SN BN-R 2CPm	16	1.82	14	1.43	12	1.41	14	2.06	28	2.66
SN BN-P ISwp	17	2.32	17	2.03	18	2.13	26	4.32	8	1.54
SN Best ISwp	18	2.33	19	2.05	17	2.10	30	4.41	12	1.55
SN BN-L ISwp	19	2.37	23	2.15	21	2.19	27	4.33	10	1.54
SN BN-R ISwp	20	2.40	24	2.17	25	2.25	29	4.37	13	1.56
SN BN-L JXhg	21	2.41	28	2.29	27	2.38	17	3.92	14	1.57
SN Best JXhg	22	2.45	27	2.29	28	2.41	22	4.24	9	1.54
IS Def	23	2.45	22	2.13	22	2.21	19	4.14	20	1.84
SN BN-P JXhg	24	2.49	29	2.32	29	2.45	20	4.23	17	1.60
SN BN-R JXhg	25	2.53	30	2.38	30	2.50	18	4.10	19	1.68
IS POp	26	2.62	26	2.24	26	2.32	21	4.23	22	2.13
SN Best Tie	27	2.83	18	2.05	19	2.16	28	4.34	30	3.32
SN BN-R Tie	28	2.86	25	2.20	23	2.22	25	4.31	29	3.17
SN BN-L Tie	29	2.86	21	2.09	24	2.22	23	4.24	31	3.40
SN BN-P Tie	30	2.93	20	2.08	20	2.17	24	4.25	32	3.85
IS STL	31	3.02	31	2.46	31	2.59	31	5.09	26	2.59
IS AIF	32	3.08	32	2.48	32	2.69	32	5.27	25	2.57

4.3 | Sorting many continuous sets of 2–16 items

In this section we consider the ArrayInRow benchmark: instead of sorting a single array multiple times, multiple arrays are created adjacent to each other and sorted in series. The number of arrays used is chosen in a way that their concatenation does not fit into the CPU's L3 cache. Before the actual measurement, the entire array is copied into a reference array and each subarray is sorted. Since all items are touched in this reference sweep, the original array should not be present in the cache at the start of the actual sorting run. After the timed run, the results are compared against the reference array for correctness.

TABLE 5 Speedup factor of the fastest sorting network over the fastest insertion sort implementation for the OneArrayRepeat experiment and array sizes 2–16

Machine	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Avg
Intel-2650	2.37	1.91	2.16	2.79	2.54	2.62	2.44	2.16	2.20	1.95	2.08	1.84	1.76	1.77	1.78	2.16
Intel-2760	–	3.15	3.16	3.02	3.23	2.84	2.77	2.47	2.27	2.04	2.06	1.97	1.86	1.79	1.80	2.46
Ryzen-1800X	4.51	5.28	5.29	5.17	5.61	7.11	4.86	4.26	4.03	3.72	3.60	3.35	3.70	3.25	3.25	4.47
RK3399	1.08	1.16	1.39	1.50	1.85	1.92	1.92	2.00	2.05	1.91	1.94	2.05	2.00	2.03	2.05	1.79
Average	2.65	2.88	3.00	3.12	3.31	3.62	3.00	2.72	2.64	2.40	2.42	2.30	2.33	2.21	2.22	2.72

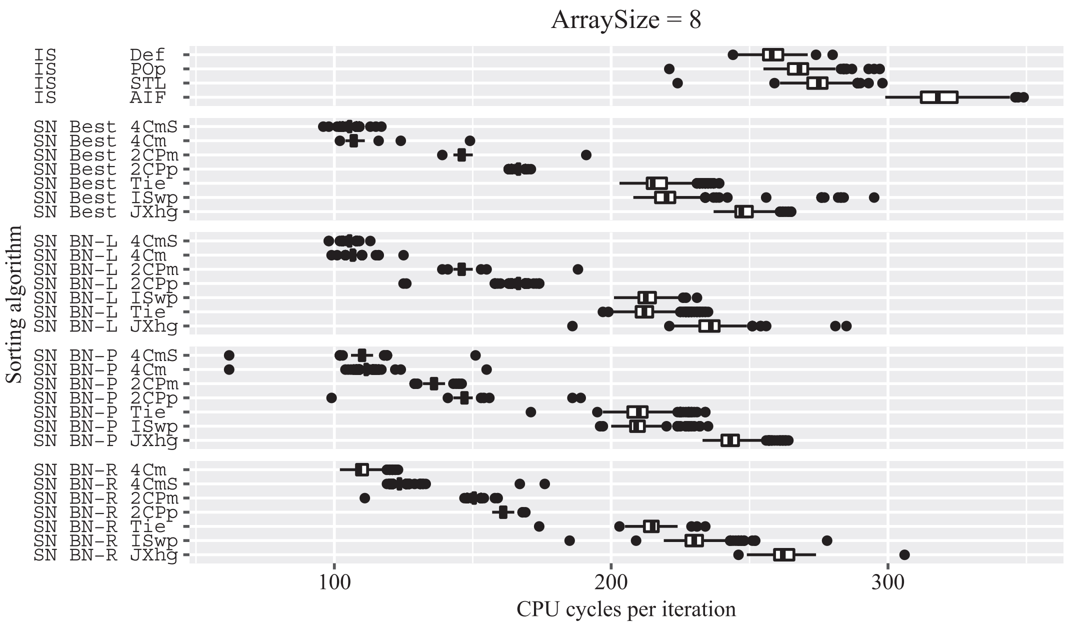

FIGURE 15 OneArrayRepeat experiment with array size = 8 on machine Intel-2650

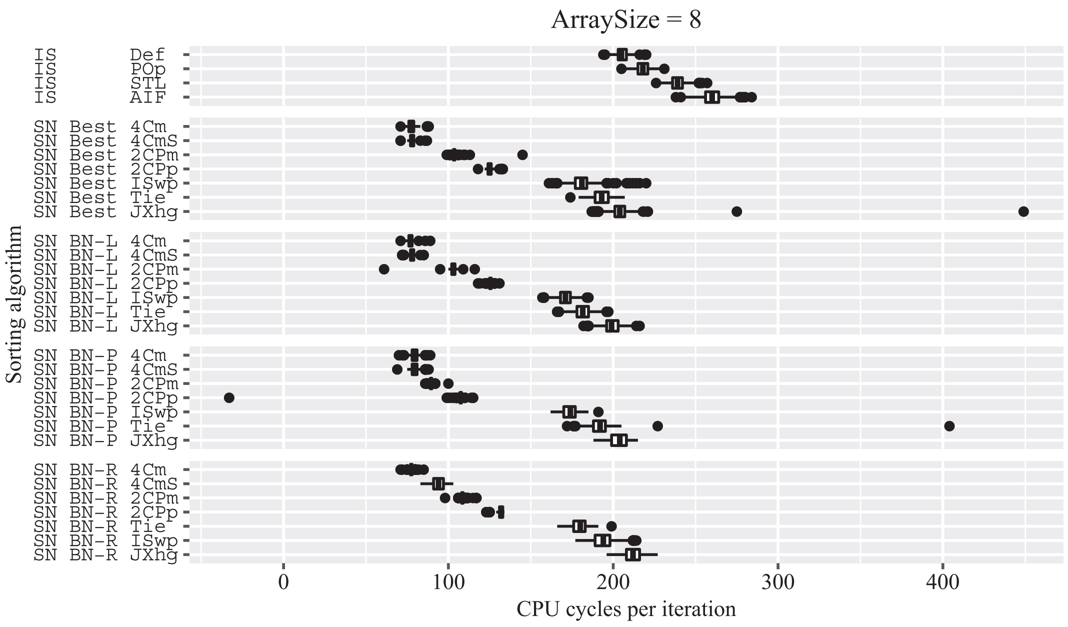

FIGURE 16 OneArrayRepeat experiment with array size = 8 on machine Intel-2670

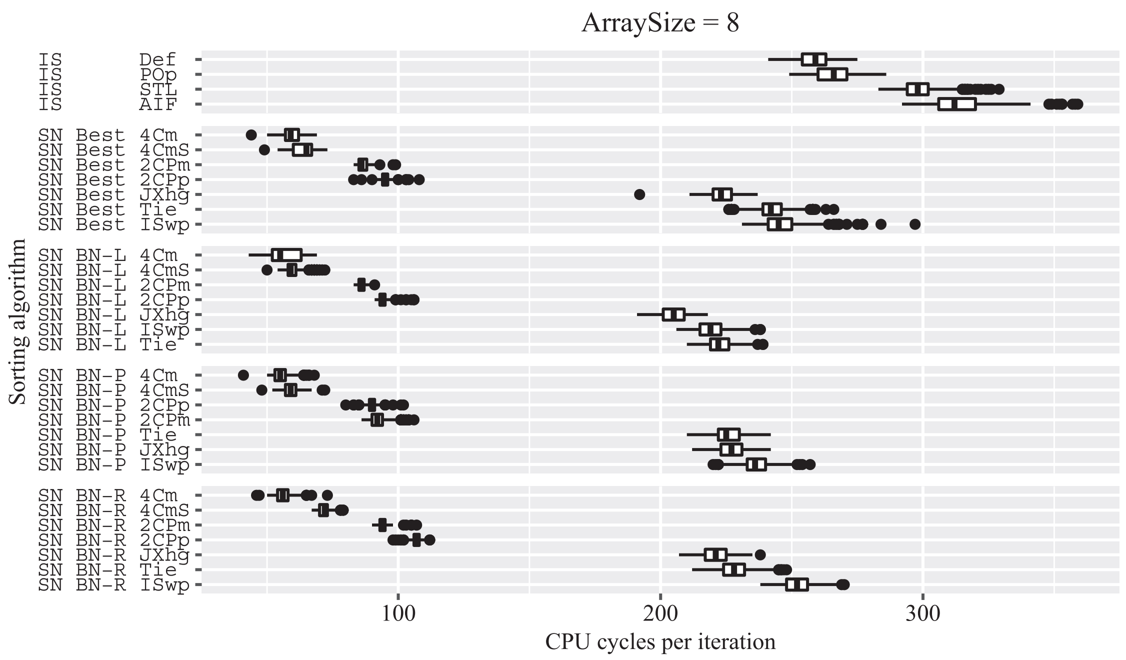

FIGURE 17 OneArrayRepeat experiment with array size = 8 on machine Ryzen-1800X

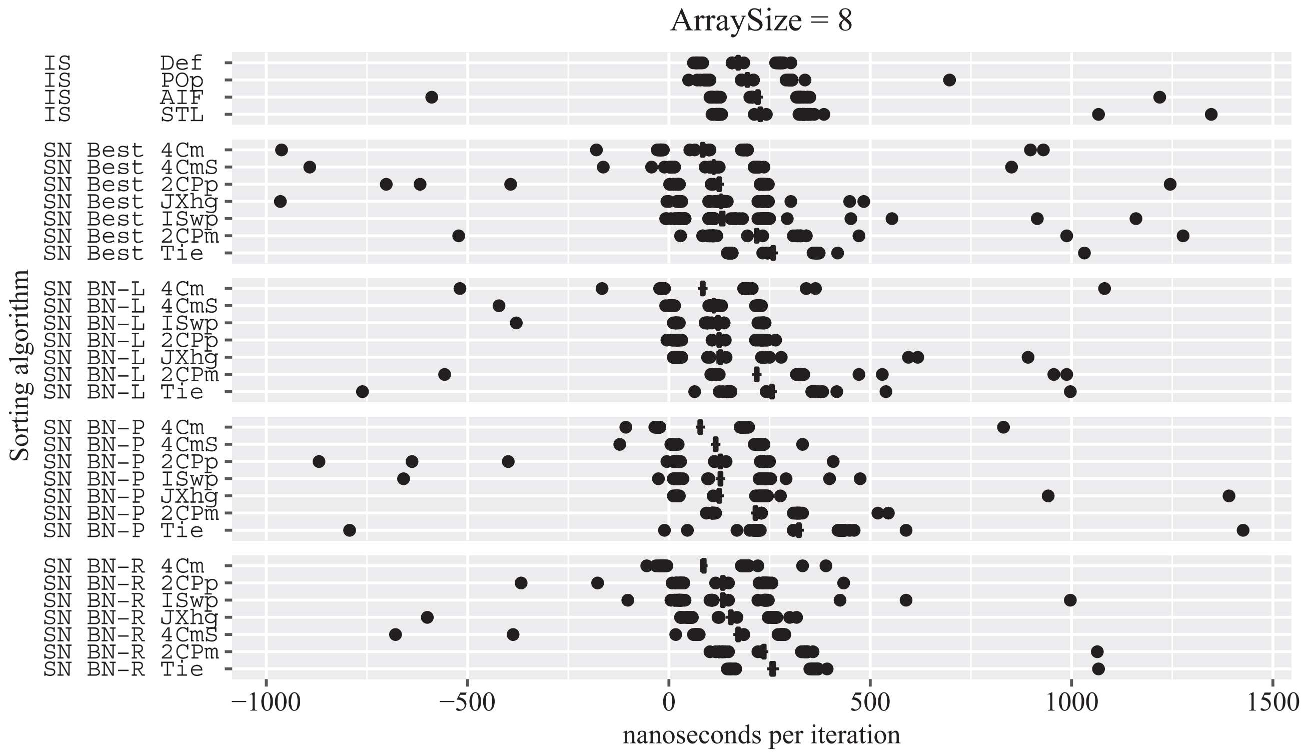

FIGURE 18 OneArrayRepeat experiment with array size = 8 on machine RK3399

Overall the results of ArrayInRow shown in Figure 20 are similar to the previous ones in Figure 19. Table 6 shows the algorithms ranked by geometric mean of the relative slow across all machines, and we can see that the order has only changed very slightly. Due to the input not being in cache there is a performance penalty for all sorters in ArrayInRow, but the relative performances do not change much. Table 7 also shows that insertion sort is slower by a factor of 2.26 on average across all machines.

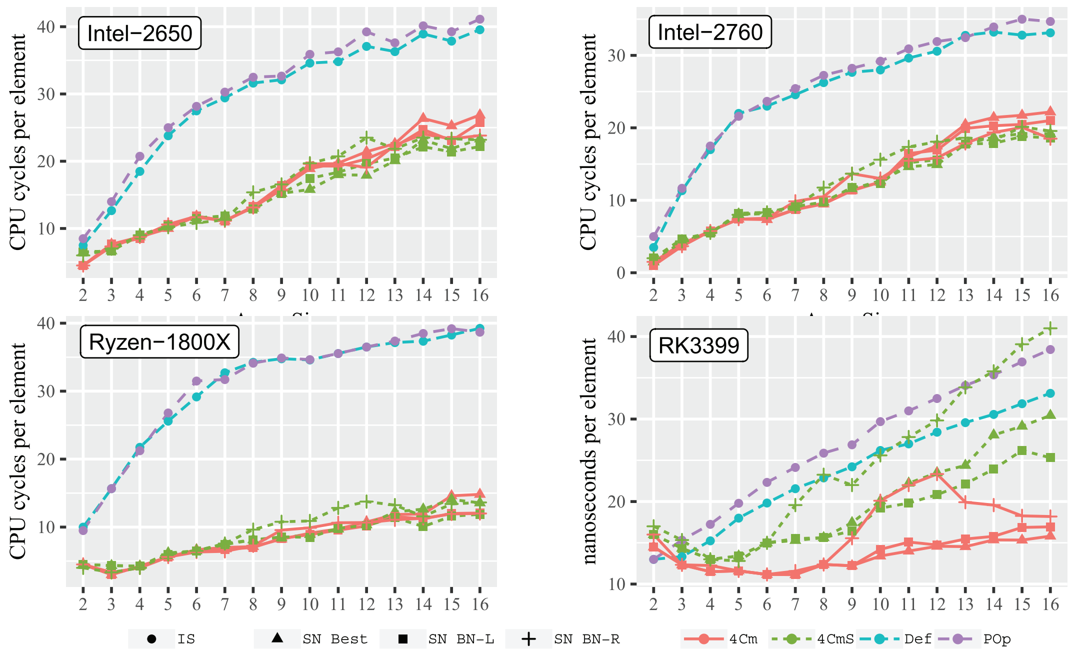

FIGURE 19 OneArrayRepeat experiment with array size 2–16 on all machines [Color figure can be viewed at wileyonlinelibrary.com]

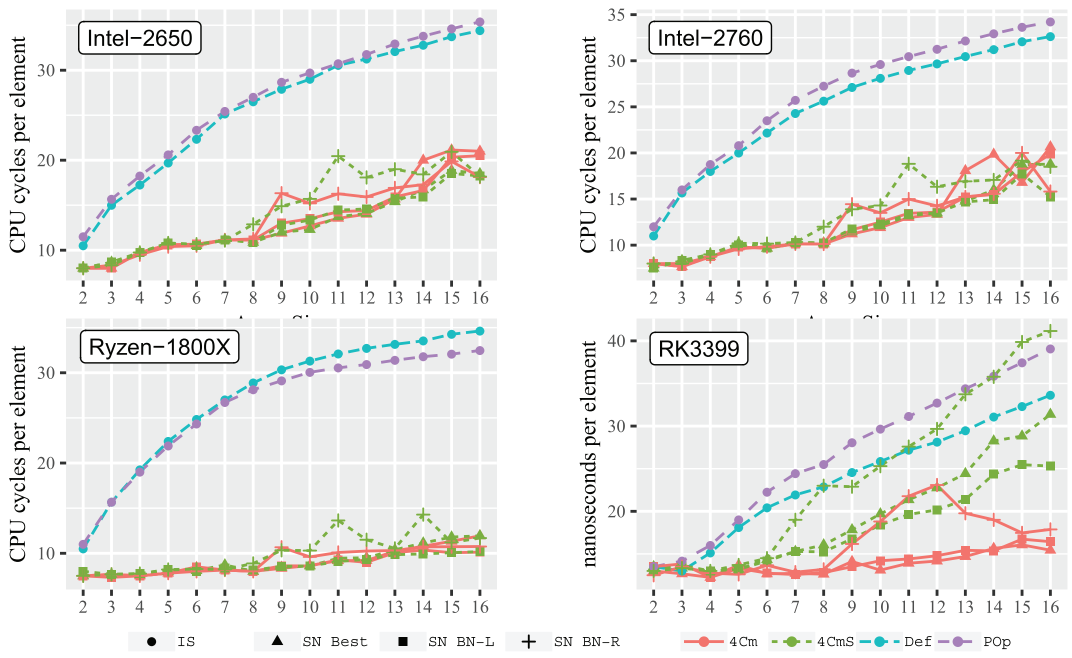

FIGURE 20 ArrayInRow experiment with array sizes 2–16 across all machines [Color figure can be viewed at wileyonlinelibrary.com]

4.4 | Sorting a large set of items with Quicksort

After the encouraging performance of the sorting networks, we were interested in how they perform as base case sorters inside a sorting algorithm for larger sets. To study this we modified Introsort,[36] the Quicksort implementation used in the current gcc's STL library. In this particular implementation Introsort calls insertion sort only once at the end on the entire array. Since this is not possible with our sorting networks, we modified the implementation to called the networks directly when the partitioning resulted in a set of 16 elements or less. Also we determined the pivot using a three-element Bose–Nelson network instead of using `if-else` and `std::swap`.

TABLE 6 Ranking of sorting algorithms by geometric mean of relative slowdowns for the ArrayInRow experiment across all machines

Sorter	GeoM	Sorter	GeoM	Sorter	GeoM	Sorter	GeoM
`SN BN-L 4CmS`	1.030	`SN Best 2CPm`	1.277	`SN Best Tie`	2.239	`IS POp`	2.330
`SN Best 4CmS`	1.041	`SN BN-P 2CPm`	1.308	`SN Best ISwp`	2.244	`SN BN-R ISwp`	2.338
`SN BN-L 4Cm`	1.041	`SN Best 2CPp`	1.443	`SN BN-P ISwp`	2.244	`SN BN-L JXhg`	2.415
`SN Best 4Cm`	1.053	`SN BN-L 2CPm`	1.451	`SN BN-L Tie`	2.244	`SN Best JXhg`	2.449
`SN BN-R 4Cm`	1.076	`SN BN-P 2CPp`	1.483	`SN BN-P Tie`	2.248	`SN BN-P JXhg`	2.457
`SN BN-P 4CmS`	1.080	`SN BN-R 2CPm`	1.537	`IS Def`	2.300	`SN BN-R JXhg`	2.460
`SN BN-P 4Cm`	1.082	`SN BN-L 2CPp`	1.685	`SN BN-R Tie`	2.302	`IS STL`	2.487
`SN BN-R 4CmS`	1.145	`SN BN-R 2CPp`	1.773	`SN BN-L ISwp`	2.329	`IS AIF`	2.514

TABLE 7 Speedup factor of the fastest sorting network over the fastest insertion sort implementation for the ArrayInRow experiment and array sizes 2–16

Machine	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Avg
Intel-2650	1.50	1.88	1.82	2.01	2.29	2.47	2.52	2.36	2.35	2.25	2.23	2.08	2.06	1.96	1.90	2.11
Intel-2760	1.69	2.14	2.06	2.14	2.46	2.54	2.53	2.41	2.36	2.23	2.23	2.07	2.09	1.91	2.14	2.20
Ryzen-1800X	1.44	2.11	2.52	2.84	3.09	3.37	3.53	3.48	3.50	3.36	3.44	3.19	3.20	3.18	3.21	3.03
RK3399	1.03	1.07	1.23	1.46	1.61	1.70	1.77	1.78	1.95	1.94	2.04	1.99	1.97	2.03	2.12	1.71
Average	1.42	1.80	1.91	2.11	2.36	2.52	2.59	2.51	2.54	2.45	2.49	2.34	2.33	2.27	2.34	2.26

In the results we labeled the base case sorter variants with the same schema as described in Section 4.2, but note again that this time the label describes the *base case sorter* integrated into Introsort. To validate the results, we also include two unmodified Introsort implementations as references: `QSort` is a direct source code copy of gcc's Introsort doing a final insertion sort at the end, and `StdSort` is a simple call to `std::sort`. Theoretically these should perform identically but in practice there are differences due to the way gcc optimizes library and nonlibrary code.

The sorters were measured using the OneArrayRepeat experiment loop with parameters `numberOfIterations` = 50, `numberOfMeasures` = 200, `arraySize` = $2^{14} = 16,384$. The running times are shown as box plots in Figures 21, 22, 23, and 24. The speedups of the best modified Introsort variants with sorting networks are compared against the `std::sort` reference implementation and variants with insertion sort as base case in Table 8.

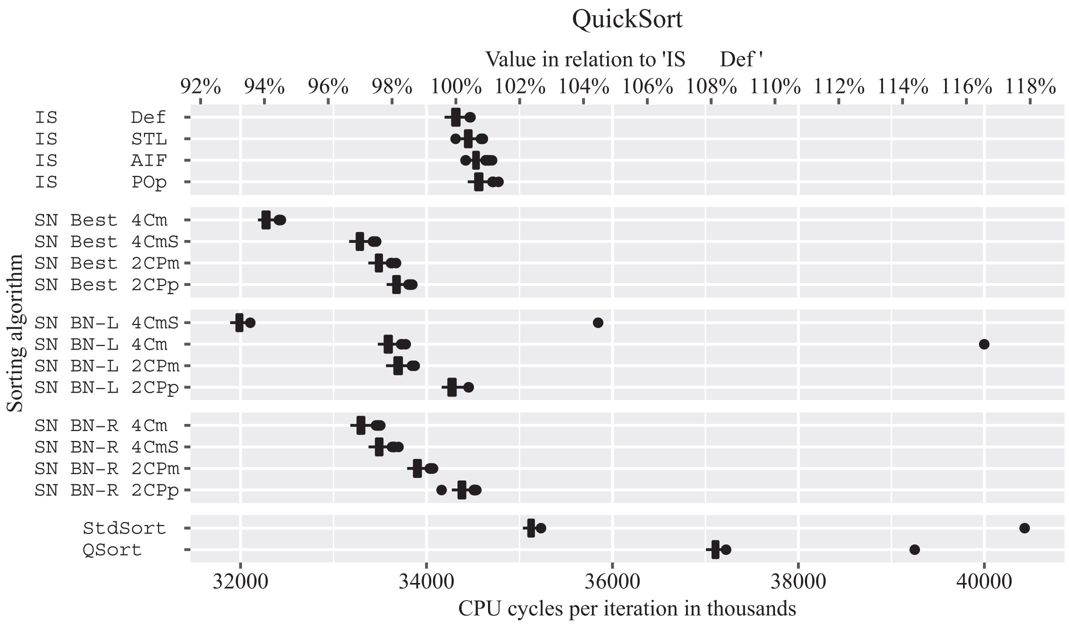

FIGURE 21 Running time of Quicksort with different base cases on machine Intel-2650

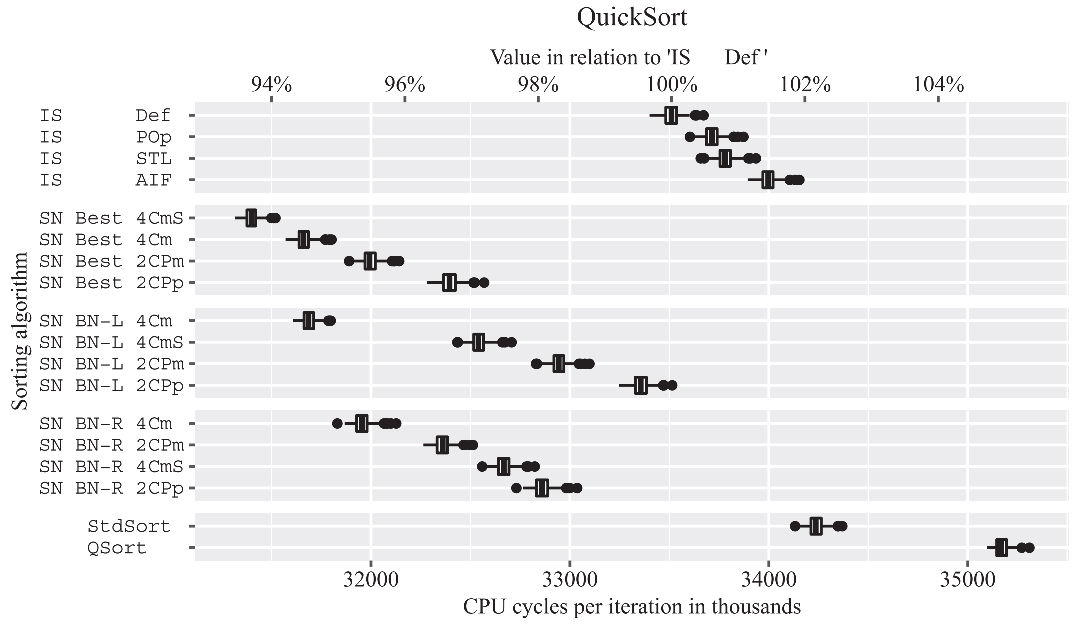

FIGURE 22 Running time of Quicksort with different base cases on machine Intel-2670

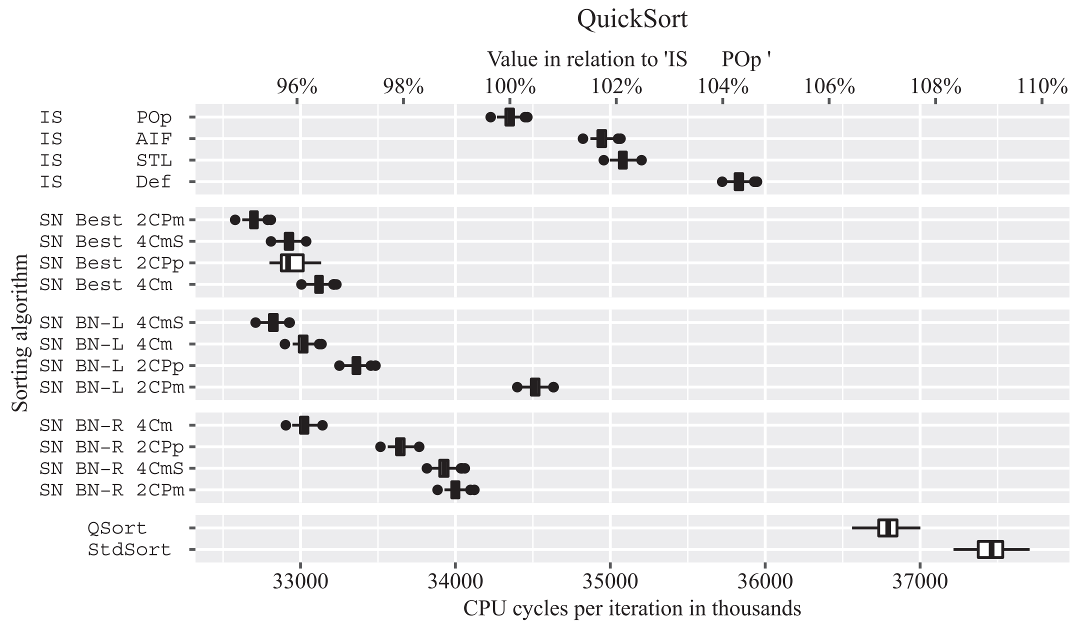

FIGURE 23 Running time of Quicksort with different base cases on machine Ryzen-1800X

Our first notable observation is that the variants with immediate insertion sort at the base case are faster than the one with the delayed final insertion sort, which probably comes from the fact that the elements are still present in the first- or second-level caches. This also explains why the `2CPm` conditional swap performs the best with Quicksort, while we saw in the last section that this is not necessarily the case when we have a cache miss for loading the items.

Recalling the results from the previous sections, we expected large improvements by reducing the time needed for sorting sets of 2–16 items due to the branchless sorting networks. However, the results with Quicksort highlight the networks' main weakness: the larger code size (see again Table 3).

By integrating the sorting networks into Quicksort for sorting the base cases, every time a partition has 16 elements or less, the executions switches from the code for Quicksort to the code for the sorting network. Considering the code sizes

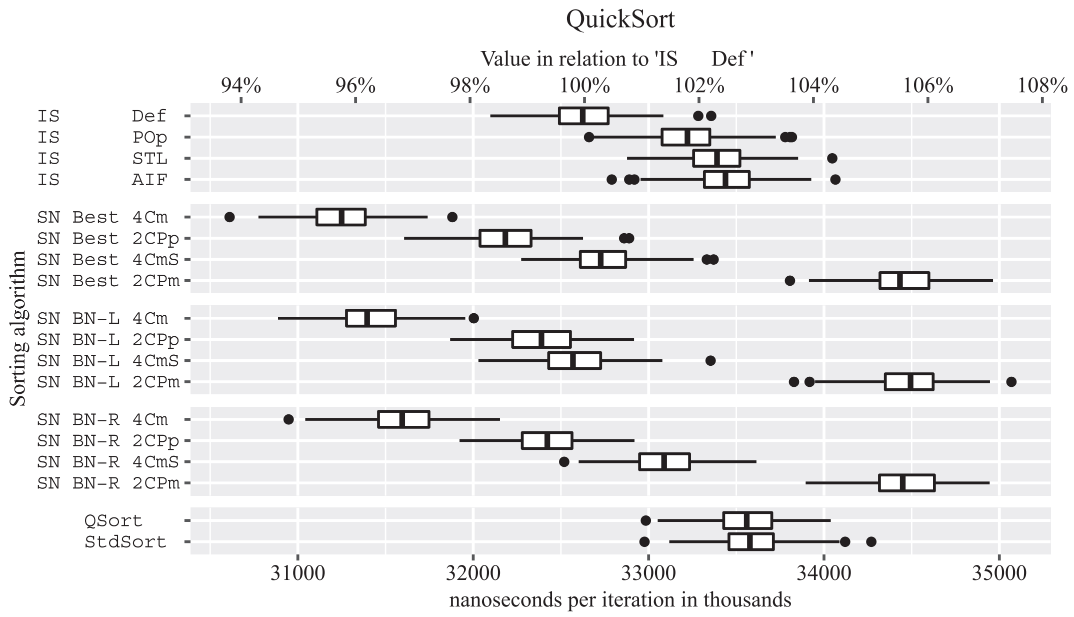

FIGURE 24 Running time of Quicksort with different base cases on machine RK3399

TABLE 8 Average speedups of the fastest sorting network over the fastest insertion sort as base case in Quicksort and unmodified `std::sort`

	Intel-2650:	Intel-2670:	Ryzen-1800X:	RK3399:
	`SN BN-L 4CmS`	`SN Best 4CmS`	`N Best 2CPm`	`SN Best 4Cm`
`IS Def`	6.7%	6.3%	8.7%	4.26%
`IS POp`	7.4%	6.8%	4.8%	5.9%
`std::sort`	8.96%	8.3%	12.7%	6.9%

shown in Table 3, we can conclude that Quicksort with insertion sort is around 1500 bytes, while Quicksort with sorting networks ranges from 12 to 37 KiB. We believe the code for Quicksort is partly removed from the L1 instruction cache and replaced with the code for the sorting network. Because the network's code is a branchless sequence of conditional swaps, each line of code is accessed exactly once per base case sort. This causes a lot of Quicksort's code to be removed from the instruction cache, counteracting the speedup of the sorting network which is then in the cache and will be partially removed again when Quicksort is handed back the flow of control.

This effect is more pronounced for machines Intel-2650, Intel-2670, and RK3399 which have 32 or 48 KiB of L1 instruction cache, where the speedup is 4.3%–6.7% for the best network base case over the best insertion sort base case, and 6.9%–9% over `std::sort`. On machine Ryzen-1800X a larger improvement was achieved for `std::sort`, probably due to the 64 KiB L1 instruction cache. Here we achieved a speedup of 12.7% over `std::sort` when making use of the best networks.

Furthermore, it comes as no surprise that we do not see improvements as large as those in Section 4.2 or 4.3 because the partitioning steps of Quicksort take the same amount of time regardless of the base case sorter. Some simple measurements showed that only 13%–20% of the time of Quicksort is spent in the base cases. Hence, these are a considerable fixed part of the variants that is not optimized using sorting networks.

4.5 | Sorting a medium-sized set of items with sample sort

In this section we focus on evaluating Register Sample Sort. The measurements were done with two different goals in mind: First, to determine which parameters work best for the machines and the array size set. And second to see if the results from the preceding three Sections 4.2, 4.3, and 4.4 would relate to the results from sample sort with the sorting networks as base cases.

Register Sample Sort was measured using the OneArrayRepeat experiment loop with parameters: `numberOfIterations` = 50, `numberOfMeasures` = 200, and `arraySize` = 256. In the experimental results the parameters of Register Sample Sort are labeled with "*xyz*" where x = `numberOfSplitters`, y = `oversamplingFactor`, and z = `blockSize`.

Figures 25, 26, 27, and 28 show box plots of running times of Register Sample Sort with `numberOfSplitters` = 3 and locality-optimized Bose–Nelson networks as base case on 256 items. To be able to compare the results on the different machines, the configurations in all plots were ordered based on their speed on machine Intel-2650. We measured larger variances and got a lot more outliers, so choosing a "best" configuration was not so easy.

An oversampling factor of 3 performed best with respect to the median on machine Intel-2650, Intel-2670, and RK3399, while 4 was best on machine Ryzen-1800X. The best results with respect to `blockSize` are also interesting, because

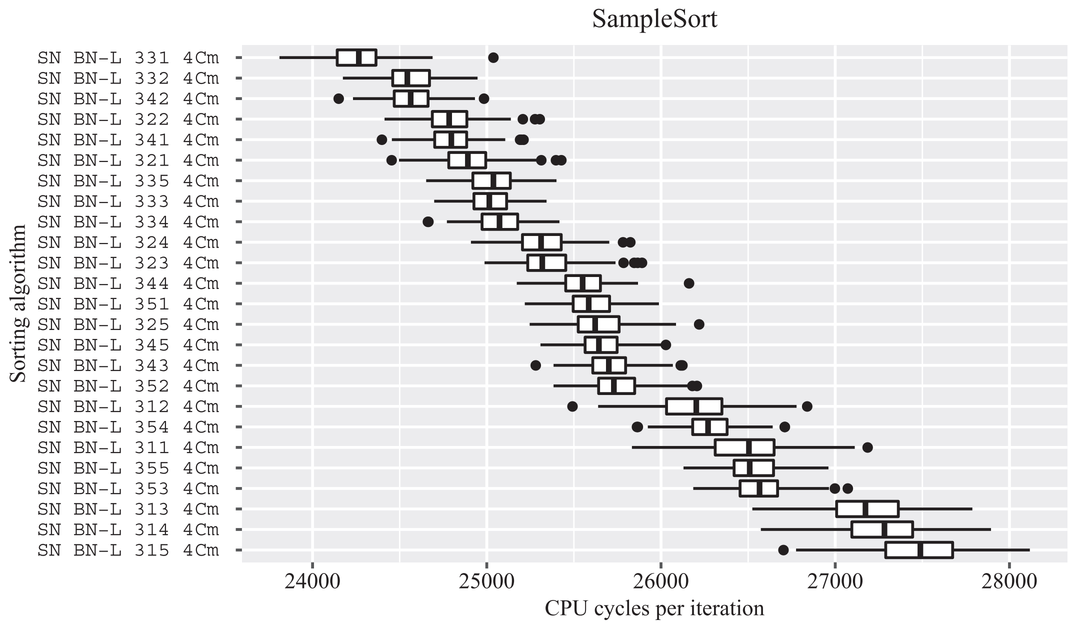

FIGURE 25 Register sample sort on machine Intel-2650 with 256 items and different configurations

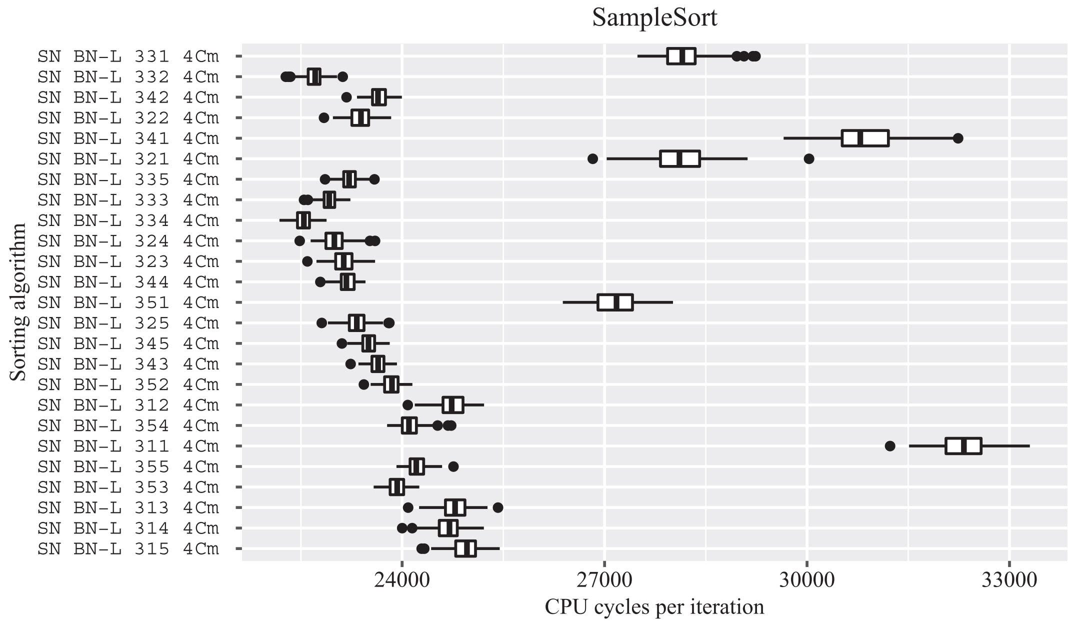

FIGURE 26 Register sample sort on machine Intel-2670 with 256 items and different configurations

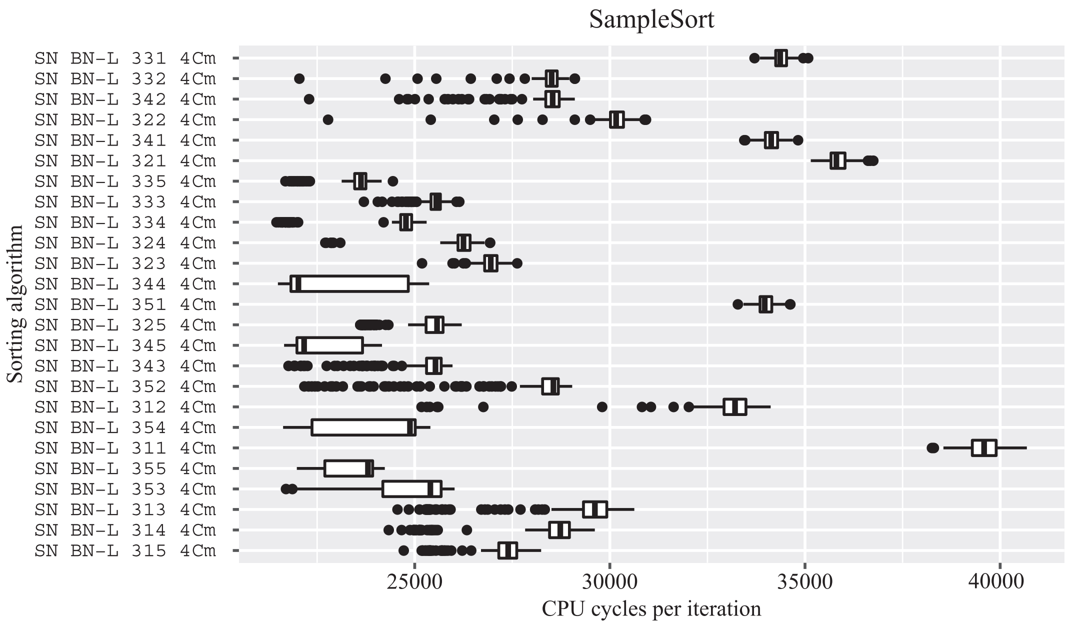

FIGURE 27 Register sample sort on machine Ryzen-1800X with 256 items and different configurations

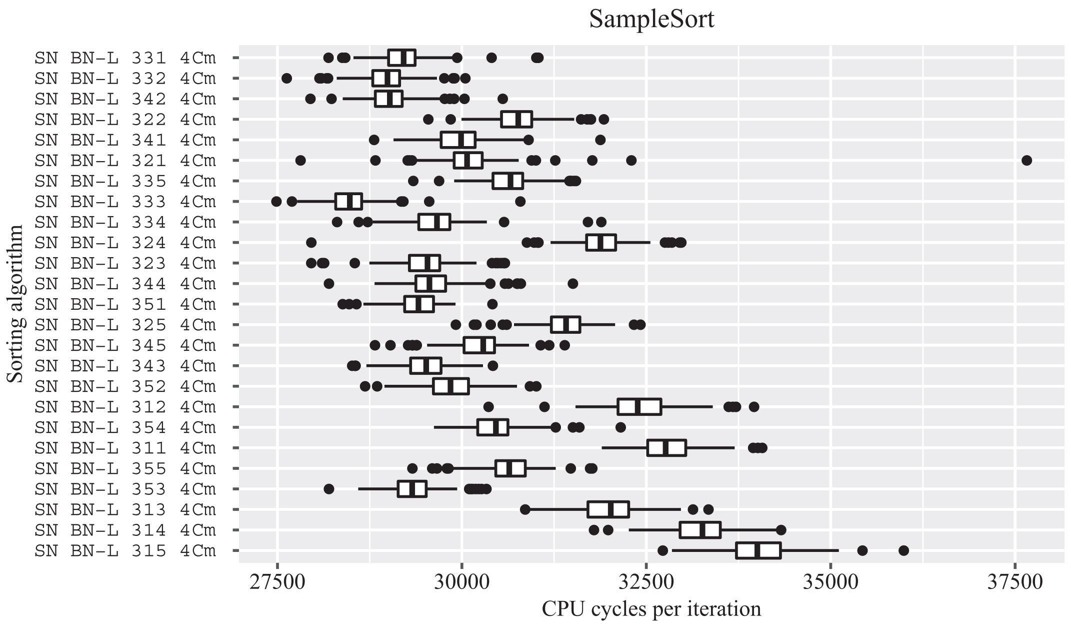

FIGURE 28 Register sample sort on machine RK3399 with 256 items and different configurations

these depend on the number of general purpose registers available. On machine Intel-2650 `blockSize` = 1 was best with oversampling factor 3, while machine Intel-2670 allowed a `blockSize` = 4, with 3 and 2 close behind. On machine Ryzen-1800X block sizes larger than 2 performed better (on average) along with an oversampling factors of 3 or greater. When looking at the other networks and insertion sort as base case, consistently well performing parameters are an oversampling factor of 3 and a block size of 4, but with very little lead over other configurations.

That is interesting to see because all three machines run x86_64 assembly instructions and have the same number of publicly visible general purpose registers. What comes into play here are hidden virtual registers and the size of the instruction cache: Machine Ryzen-1800X has double the amount of L1 instruction cache of what machines Intel-2650 and Intel-2670 have. We can only assume that the instructions for classifying three elements need more space than the smaller 32 KiB instruction caches can provide, while the 64 KiB instruction cache in machine Ryzen-1800X can fit the instructions for classifying four and/or almost five elements at once, considering that block size 5 also performs well.

Machine RK3399 however is an ARM chip, which shows a much larger variance in the results of Register Sample Sort, but is also more robust against the parameters. On machine RK3399 oversampling factor 3 with `blockSize = 3` was best, which indicates a similar number of general purpose registers as in the x86_64 machines. However, the larger robustness can actually be interpreted as that the ARM chip incorporates *fewer* unpredictable running time optimizations such as speculative and out-of-order execution of instructions.

The second goal was to see if the results from Sections 4.2, 4.3, and 4.4 would relate to using sample sort with the sorting networks as base cases. These results can be seen in Figures 29, 30, 31, and 32 for the `332` configuration. All measurements were made with a base case limit of 16.

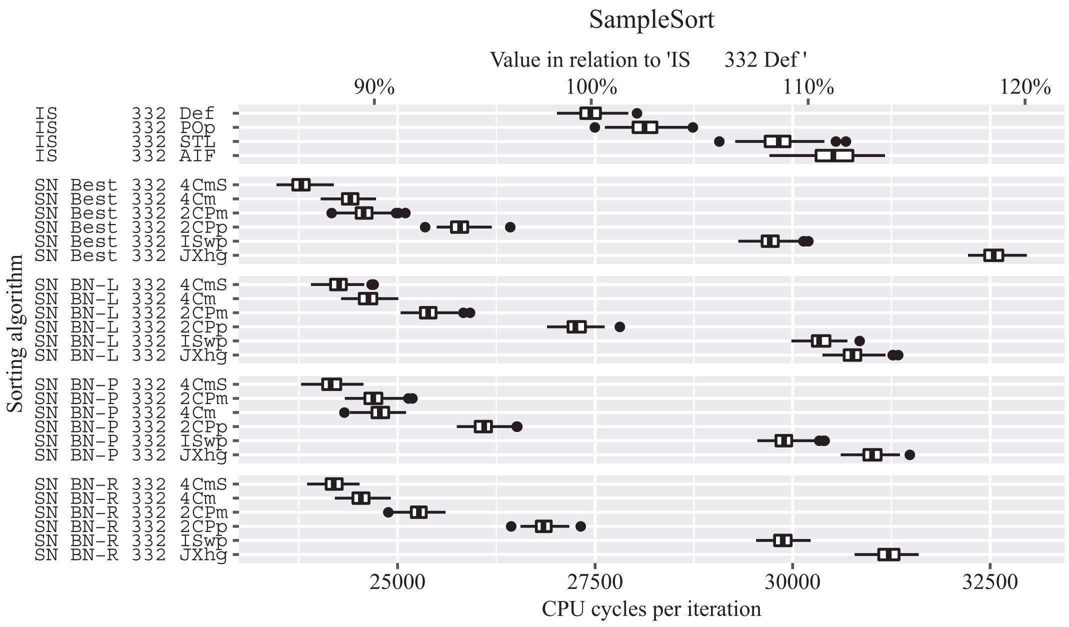

FIGURE 29 Register sample sort `332` with different base cases on machine Intel-2650

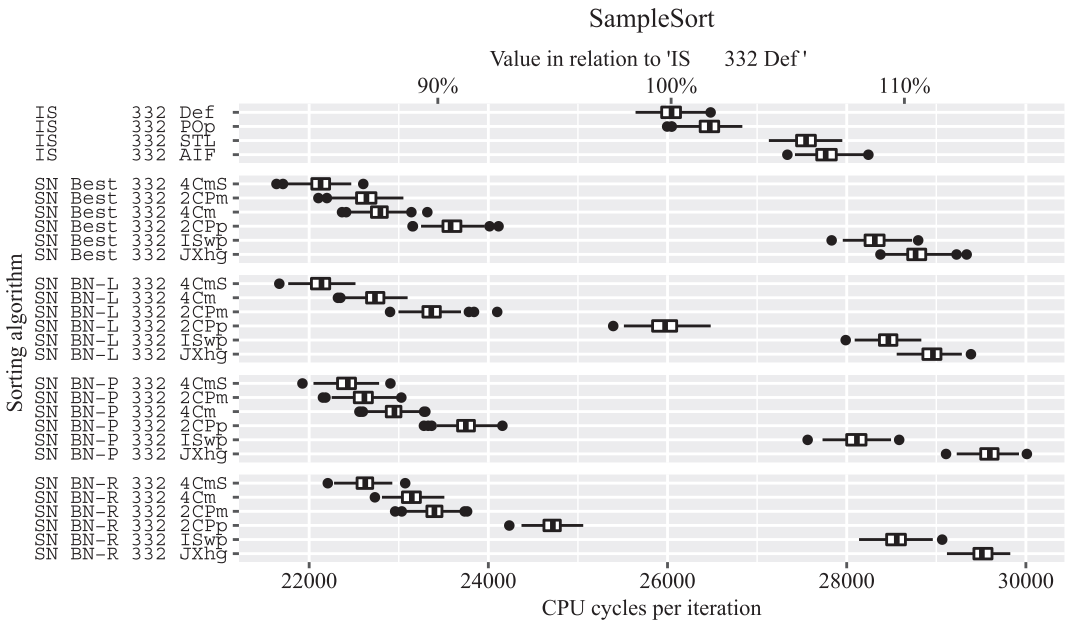

FIGURE 30 Register sample sort `332` with different base cases on machine Intel-2670

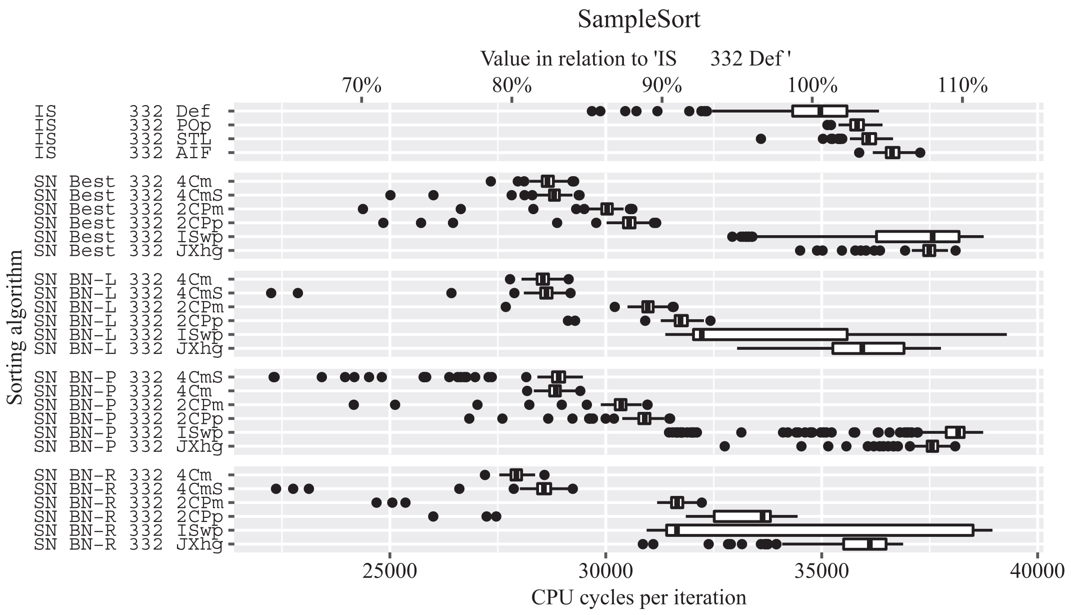

FIGURE 31 Register sample sort `332` with different base cases on machine Ryzen-1800X

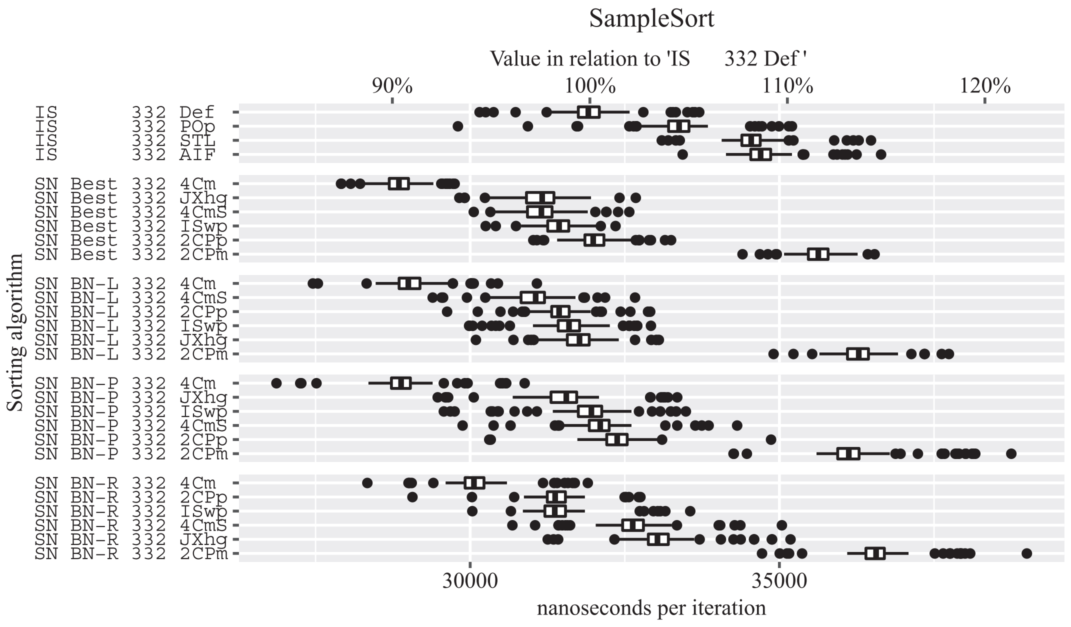

FIGURE 32 Register sample sort `332` with different base cases on machine RK3399

The achieved speedups of using the sorting networks are summarized in Table 9. On the left we see Register Sample Sort with insertion sort as base case `I Def` and unmodified `std::sort` that was also measured sorting 256 elements. On the top we see the best performing network "`SN Best 332 4CmS`" as a base case for Register Sample Sort on all three machines. The number indicates the speedup of Register Sample Sort *with* the sorting network over Register Sample Sort with insertion sort and over std::sort.

Again we see that due to machine Ryzen-1800X having a larger L1 instruction cache the performance gain is greater than for the other machines. And the results from ARM machine RK3399 again have a larger variance than the others with conditional swap `4Cm` performing better than `4CmS`. Unlike in the previous section though we achieved much greater speedups as a result of using the sorting networks as a base case. That comes from the fact that Register Sample Sort has no unpredictable branches classifying the elements, as opposed to Quicksort having to deal with conditional branches

TABLE 9 Average speedups of the fastest sorting network over the fastest insertion sort as base case in register sample sort and unmodified `std::sort`

	Intel-2650:	Intel-2670:	Ryzen-1800X:	RK3399:
	`SN Best 4CmS`	`SN Best 4CmS`	`SN BN-R 4Cm`	`SN Best 4Cm`
`I Def`	13.3%	15%	19.7%	9.6%
`std::sort`	28.7%	32.5%	21.5%	2.8%

during the partitioning, while both need to invest the same time to sort all the base cases. So with Register Sample Sort, the base case sorting takes up a larger time portion of the whole execution than it does with Quicksort. From further results we also see that we can get up to a factor of 1.4 faster than `std::sort` for sets of 256 items with very few conditional branches.

4.6 | Sorting a large set of items with IPS^{4}o

With our efficient implementation of Register Sample Sort for medium-sized sets we can now incorporate our new base case sorters into a complex sorting algorithm. We evaluated them in Axtmann et al.'s *In-Place Parallel Super Scalar Samplesort*(IPS^{4}o)[5] without additionally introducing parallelism into our experiments. The IPS^{4}o algorithm has many parameters that can be adjusted. For our evaluation the most important parameter was `IPS4oBaseCaseSize`, which specifies which base case size to *aim* for. Even though this number is the goal, IPS^{4}o may output larger base cases because it is a large-size sorter. This was the reason we developed Register Sample Sort to break these medium-sized sets down into sizes that can be sorted using the sorting networks.

We initially started the measurement using the best combination of Register Sample Sort from Section 4.5 as a base case for IPS^{4}o, together with the default `IPS4oBaseCaseSize` = 16. However, this combination turned out to perform worse than just insertion sort.

In preliminary measurements we found that 51% of base cases are at most 16 items, and 78% are at most 32 items with `IPS4oBaseCaseSize` = 16. For `IPS4oBaseCaseSize` = 32, 23% of base cases are at most 16 items, and 47% are at most 32 items. From that it was evident that in most of the instances with parameter `IPS4oBaseCaseSize` = 16 the base case sorter was being invoked on sets smaller than even 32 elements. That also meant that Register Sample Sort had to deal with inputs just slightly larger than 16, which incurs a larger overhead than plain insertion sort and is not justified by the larger amount of items.

In addition to that the size of the instruction cache that had already had a great influence on the measurements of Quicksort seemed to be another factor for the bad performance of Register Sample Sort as a base case.

That is why we decided to measure the following setups:

(a) pure insertion sort as base case (`IS`) with `IPS4oBaseCaseSize` ∈ {16, 32},
(b) Register Sample Sort as base case (`SS+SN`) with `IPS4oBaseCaseSize` ∈ {16, 32, 64}, tree configurations "`331`" and "`332`", and `Best` networks and Bose–Nelson networks optimizing locality (`BN-L`) and recursion (`BN-R`) for base case size 16 of Register Sample Sort with conditional swap `4CmS`, and
(c) a combination of the sorting networks and insertion sort (`IS+SN`) without Register Sample Sort.

Variant (c) was introduced because the base case sizes were often smaller than 16, and we wanted to make use of that by using the sorting networks, while not having to rely on Register Sample Sort with its larger overhead for the slightly larger base cases, hence `IS+SN` uses the Bose–Nelson networks optimizing locality if the set had 16 elements or less, and insertion sort otherwise.

Figures 33, 34, 35, and 36 display the results from the measurements with the four variants above. The `IPS4oBaseCaseSize` ∈ {16, 32} was added to the variant's label along with an underscore followed by the Register Sample Sort configuration. The OneArrayRepeat experiment loop was used with parameters `numberOfIterations` = 50, `numberOfMeasures` = 200, and `arraySize` = 2^{18} = 262, 144. On machine Intel-2650 one outlier for "`SS+SN BN-R 32_331 4CmS`" with the value 38,454,084 cycles was removed from the plot for better readability.

As already seen in Axtmann et al.'s publication,[5] we measured an average speedup factor of 2.3 over `std::sort` with unchanged IPS^{4}o across all machines. However, on machine Intel-2650, none of our variants (`IS`, `SS+SN`, `IS+SN`) led to an improvement in sorting speed over the default use of insertion sort with `IPS4oBaseCaseSize` = 32. For machine

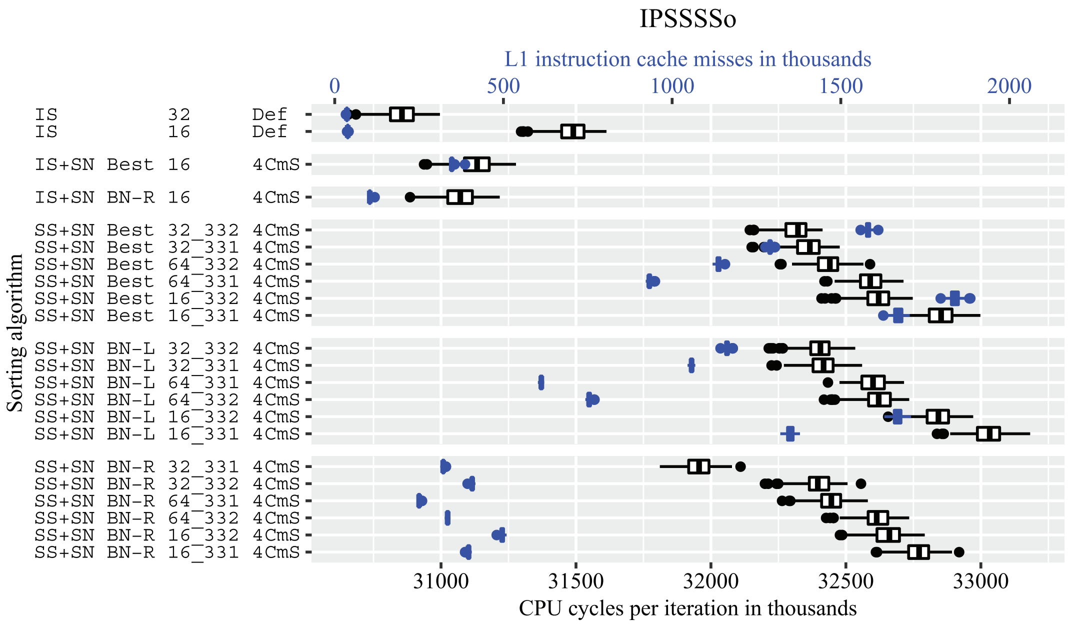

FIGURE 33 Sorting times for IPS^{4}o on machine Intel-2650 with different base cases and base case sizes [Color figure can be viewed at wileyonlinelibrary.com]

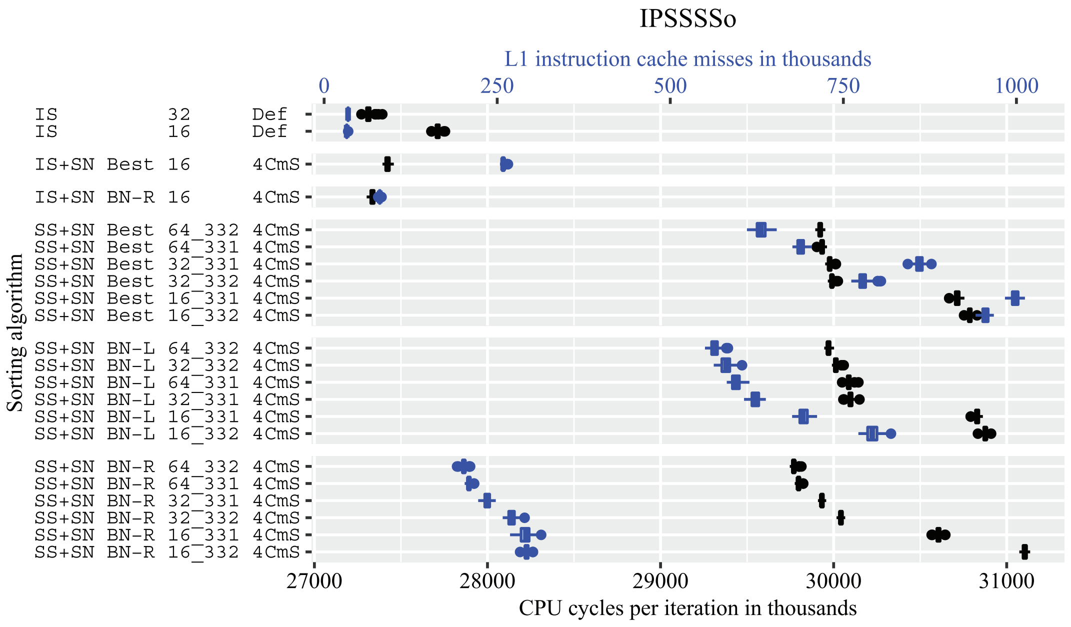

FIGURE 34 Sorting times for IPS^{4}o on machine Intel-2670 with different base cases and base case sizes [Color figure can be viewed at wileyonlinelibrary.com]

Intel-2670, interestingly, using Register Sample Sort did not lead to an improvement, but the combination of insertion sort and Bose–Nelson networks did manage to reach par with the default implementation. For machine Ryzen-1800X, Register Sample Sort also did not lead to an improvement, but `IS+SN` outperformed the default insertion sort by about 5%. Only on machine RK3399 we see a reasonable speedup of Register Sample Sort with sorting networks of up to 7% in the best configuration. These average speedups of our sorting networks used in IPS^{4}o as base cases are shown in more detail in Table 10.

Figures 33, 34, and 35 also show the number of L1 instruction cache misses in blue plotted on the top x-axis. These were measured using the performance counters in the CPU using the Linux `perf` interface, which were only available

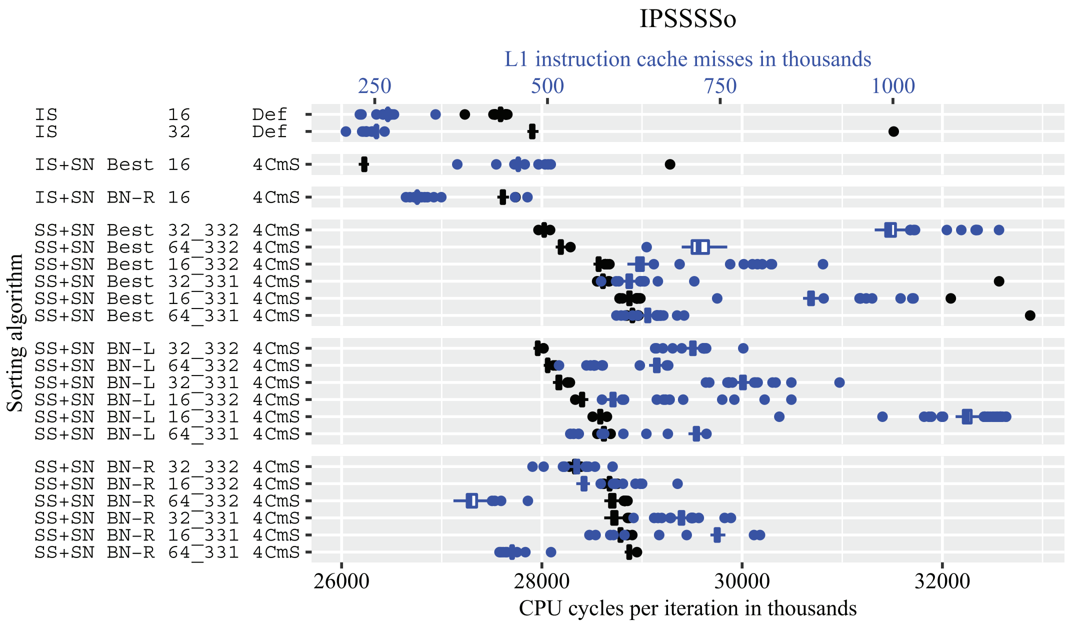

FIGURE 35 Sorting times for IPS^{4}o on machine Ryzen-1800X with different base cases and base case sizes [Color figure can be viewed at wileyonlinelibrary.com]

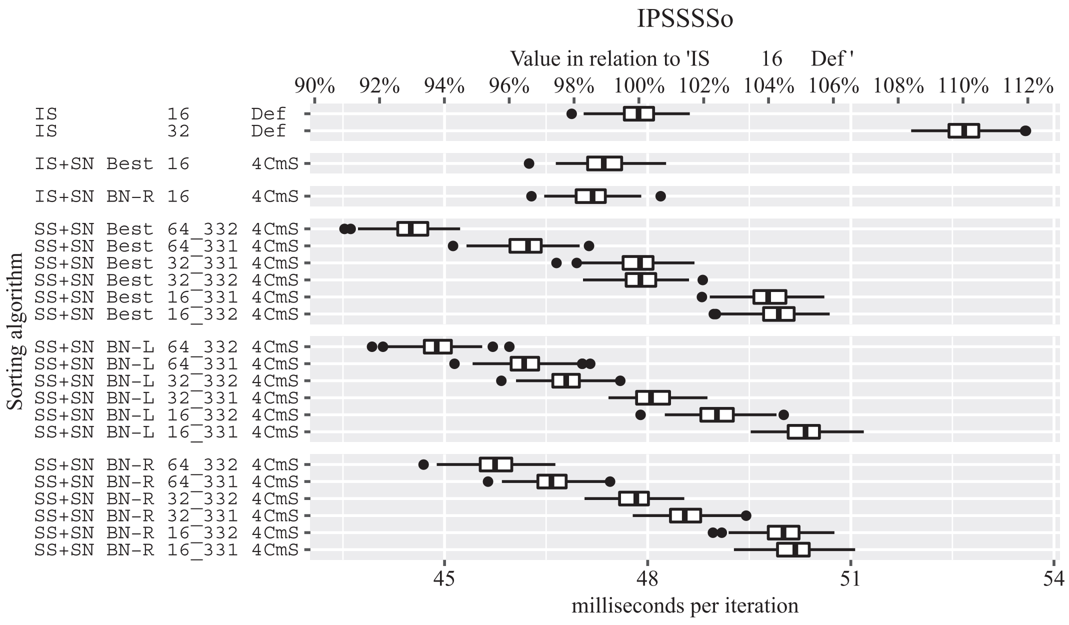

FIGURE 36 Sorting times for IPS^{4}o on machine RK3399 with different base cases and base case sizes

on Intel and AMD machines, and not on the RK3399 system. In Figure 33 we can recognize some expected behavior: IPS^{4}o with insertion sort code (`IS`) is small enough to fit into the L1 cache and thus has the lowest cache miss count. The versions with sorting networks are larger, with the "best" networks (`Best`) incurring the most instruction cache misses. The recursively defined Bose–Nelson networks (`BN-R`) require less than half as many cache misses as the locality-aware (`BN-L`) variants. The same expected behavior is also visible on machine Intel-2670 in Figure 34. There appears to be a relationship between time (CPU cycles) and L1 instruction cache misses, this relationship is neither perfect nor linear, but the smaller sorter code `IS`and `BN-R` is also faster on these systems. On the AMD Ryzen-1800X in Figure 35 the first striking difference is the wide range of results for the L1 cache misses. There are many more outliers on the AMD system, possibly due to a different hardware performance counters or sampling thereof. The actual body of the box plots however

TABLE 10 Average speedups of the fastest sorting network over the fastest insertion sort as base case in IPS^{4}o and unmodified `std::sort`

	Intel-2650: `IS+SN BN-R 16 4CmS`	**Intel-2670:** `IS+SN BN-R 16 4CmS`	**Ryzen-1800X:** `IS+SN BN-R 16 4CmS`	**RK3399:** `SS+SN Best 64 4CmS`
`IS 16 Def`	1.3%	1.36%	4.9%	7%
`IS 32 Def`	−0.7%	−0.08%	6%	15.4%
`std::sort`	58.2%	62.2%	66.6%	38.4%

are usually very small, which leads us to believe that these outliers are more measuring errors than actually a concern. The absolute number of L1 instruction cache misses on the AMD machine starts at around 250k while they are much smaller on the Intel machines. This is probably due to the different hardware measurement methods. The L1 instruction cache misses again appear to have a relationship to CPU cycles; however, it is less clear than on the Intel machines. For example, the first of the two insertion sorts (`IS`) is faster, but has more cache misses than the second one. The same is the case for the insertion sorts with sorting networks (`IS+SN`) in the third and fourth line. The order within the sorting networks is also strange: faster networks often have higher instruction cache miss counts. The correlation between cycles and cache misses is much clearer in Figure 34. As stated before, there are no cache miss numbers in Figure 36, because the hardware or software did not support measuring them. We can conclude that measuring L1 cache misses yields further evidence that the sorting networks are large in code size, there appears to be relationship with running time, but it is less clear and definitely not linear.

5 | CONCLUSION

5.1 | Results and assessment

We have seen that for sorting sets of up to 16 elements it can be viable to use sorting algorithms other than insertion sort. We looked at sorting networks in particular, paying special attention to the implementation of the conditional swap and giving multiple alternative ways of realizing it.

After seeing that the sorting networks outperform insertion sort each on their own for a specific array size in Section 4.2 and 4.3, we saw in Section 4.4 that this improvement does not necessarily transfer to sorting networks being used as base case sorter in Quicksort. Because the networks have a larger code size, the code for Quicksort is removed from the instruction cache and the advantage of not having conditional branches is impaired by that larger code size. But we also saw that for machines with larger instruction caches using sorting networks with Quicksort can lead to visible improvements (about 13%), but we were still able to reach up to 9% improvement over `std::sort` on machine Intel-2650.

Then we integrated the sorting networks into a more advanced sorter, IPS^{4}o, which was possible by adding an intermediate sorter into the procedure. For that we created Register Sample Sort, which is an implementation of Super Scalar Sample Sort that holds the splitters in general-purpose registers instead of an array. When measuring IPS^{4}o with Register Sample Sort as a base case, we found that the instruction cache makes even more of a difference, because we now add the code size for Register Sample Sort on top of the code size for the sorting networks.

We proposed an additional alternative to Register Sample Sort, using a combination of insertion sort and sorting networks: For base cases of 16 elements or less, we used the sorting network, for any size above that insertion sort.

On machine Intel-2650 with a smaller instruction cache of 32 KiB we could not achieve a speedup with any of the variants, on machine Intel-2670 the combination of insertion sort and sorting networks led to a running time on par with insertion sort as base case. The only substantial improvement we achieved with IPS^{4}o was on the ARM machine, where using Register Sample Sort led to an improvement of 7% over the best insertion sort variant.

In closing, we want to mention that this particular implementation only compiles when using the gcc C++ compiler due to compiler-dependent inline-assembly statements. This also means that the code is probably not as fast as it could be due to the inline-assembly not being optimized by the compiler. However, as of today, there is no consistent method to generate conditional move operations from C++ without assembly. The complete project is available on github at https://github.com/JMarianczuk/SmallSorters.

5.2 | Experiences and hurdles

The greatest hurdle we encountered during this project was, as mentioned in Section 4.1, the fact that the compiler reduces its optimizations with increasing compilation effort, when compiling only a single source file. That can lead to performance variations that happen for no “apparent” reason, and is especially tricky when dealing with templated methods that cannot be moved from header files into source files. The solution was to use code generation and to include all logically coherent method invocations in one wrapper method that is then placed in its own source file, to not have different parts of the program influencing each other over the decision which one gets to be optimized and which one not.

5.3 | Future work

Empirically, an important improvement would be to mitigate the instruction cache faults that limited performance in our experiments. Besides looking at sorting networks with smaller memory footprint, one could separate and delay solving the base cases from the remaining logics of a divide-and-conquer sorter such as Quicksort or sample sort. By also bucket-sorting the base cases by their size, one could further improve instruction-cache locality—one could then solve batches of subproblems of identical size. The obvious downside of worsened data locality could be mitigated by performing this separation not globally but on subproblems that fit into an appropriate level of the cache hierarchy (e.g., L3 cache).

Identical subproblem sizes are also possible when implementing the base case of merge sort. For large data sets, this would likely imply using merge sort for intermediate input sizes that fit into the cache and one would want to have a highly tuned implementation that avoids branch mispredictions, for example, based on Reference 37. One could even consider merging circuits as yet another intermediate stage.

One can also look further into implementation techniques for sorting networks. One would like to explore further possibilities to implement the conditional swap for the sorting networks, as well as seeing which of the C++ compilers generate conditional moves when using portable C++ code instead of compiler- and architecture-dependent inline-assembly. That also includes looking at conditional swaps for elements that differ from the 64-bit key and reference value pair that we looked at in this article. We also have not investigated yet how the instruction scheduling policies of the compilers interact with our implementations. For some data types, it would certainly be interesting to consider SIMD instructions.

Going beyond sorting networks, one could look for small case sorters not based on compare-and-swap primitives. These might involve fewer instructions or data-dependencies.

ACKNOWLEDGMENT
Open access funding enabled and organized by Projekt DEAL.

ORCID
Timo Bingmann https://orcid.org/0000-0003-0529-0097

REFERENCES

1. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput*. 1998;20(1):359-392.
2. Bose RC, Nelson RJ. A sorting problem. *J ACM*. 1962;9(2):282-296.
3. Codish M, Cruz-Filipe L, Nebel M, Schneider-Kamp P. Optimizing sorting algorithms by using sorting networks. *Formal Aspects Comput*. 2017;29(3):559-579.
4. Sanders P, Winkel S. Super scalar sample sort. Paper presented at: Proceedings of the 12th European Symposium on Algorithms (ESA). 3221 of LNCS; 2004:784-796; Springer, New York, NY.
5. Axtmann M, Witt S, Ferizovic D, Sanders P. In-Place parallel super scalar samplesort (IPS^{4}o). Paper presented at: Proceedings of the 25th European Symposium on Algorithms (ESA). 87 of LIPIcs; 2017:9:1–9:14; Schloss Dagstuhl. preprint arXiv:1705.02257.
6. Marianczuk J. Engineering faster sorters for small sets of items [Bachelor thesis]. Karlsruhe Institute of Technology, Germany; 2019. arXiv:1908.08111.
7. Knuth DE. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd ed. Addison Wesley: Longman Publishing Co., Inc; 1998.
8. Mahmoud HM. *Sorting: A Distribution Theory*. Hoboken, NJ: John Wiley & Sons; 2000.
9. Batcher KE. Sorting networks and their applications. Paper presented at: Proceedings of the 32 of AFIPS Conference Proceedings American Federation of Information Processing Societies (AFIPS); 1968:307-314; Thomson Book Company, Washington, DC.

10. Ajtai M, Komlós J, Szemerédi E. An O(nlogn) sorting network. Paper presented at: Proceedings of the 15th ACM Symposium on Theory of Computing (STOC); 1983:1-9; ACM, New York, NY.
11. Inoue H, Moriyama T, Komatsu H, Nakatani T. A high-performance sorting algorithm for multicore single-instruction multiple-data processors. *Softw Pract Exper*. 2012;42(6):753-777.
12. Furtak T, Amaral JN, Niewiadomski R. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. Paper presented at: Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA); 2007:348-357; ACM, New York, NY.
13. Xiaochen T, Rocki K, Suda R. Register level sort algorithm on multi-core SIMD processors. Paper presented at: Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms; 2013:1-8; ACM, New York, NY.
14. Bramas BA. Novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake. *Int J Adv Comput Sci Appl*. 2017;8(10):337–344. arXiv preprint arXiv:1704.08579.
15. Hou K, Wang H, Feng WC. A framework for the automatic vectorization of parallel sort on x86-based processors. *IEEE Trans Parall Distrib Syst (TPDS)*. 2018;29(5):958-972.
16. Hibbard TN. An empirical study of minimal storage sorting. *Commun ACM*. 1963;6(5):206-213.
17. Yin Z, Zhang T, Müller A, et al. Efficient parallel sort on AVX-512-based multi-core and many-core architectures. Paper presented at: Proceedings of the 21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems (HPCC/SmartCity/DSS); 2019:168-176; Zhangjiajie, China: IEEE.
18. Müller R, Teubner J, Alonso G. Sorting networks on FPGAs. *VLDB J*. 2012;21(1):1-23.
19. Sklyarov V, Skliarova I. High-performance implementation of regular and easily scalable sorting networks on an FPGA. *Microprocessors Microsyst*. 2014;38(5):470-484.
20. Hoare CAR. Quicksort. *Comput J*. 1962;5(1):10-16.
21. Sedgewick R. *Algorithms*. Boston, MA: Addison-Wesley; 1983.
22. Parberry IA. Computer-assisted optimal depth lower bound for nine-input sorting networks. *Math Syst Theory*. 1991;24(1):101-116.
23. Bundala D, Zavodny J. Optimal sorting networks. Paper presentes at: Proceedings of the 8th International Conference on Language and Automata Theory and Applications (LATA). 8370 of LNCS; 2014:236-247; Springer, New York, NY.
24. Codish M, Cruz-Filipe L, Frank M, Schneider-Kamp P. Twenty-five comparators is optimal when sorting nine inputs (and Twenty-Nine for Ten). Paper presented at: Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence (ICTAI); 2014:186-193; Limassol, Cyprus: IEEE.
25. Ehlers T, Müller M. New bounds on optimal sorting networks. Paper presented at: Proceedings of the 11th Conference on Computability in Europe (CiE). 9136 of LNCS; 2015:167-176; Springer, New York, NY.
26. Dobbelaere B. SorterHunter; 2017. https://github.com/bertdobbelaere/SorterHunter. Website Accessed 2019.
27. Free Software Foundation Using the GNU compiler collection (GCC): how to use inline assembly language in C code. https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html. Accessed 2019.
28. Gamble JM. Sorting network generator. http://pages.ripco.net/~jgamble/nw.html. Accessed 2019.
29. Frazer WD, McKellar AC. Samplesort: a sampling approach to minimal storage tree sorting. *J ACM*. 1970;17(3):496-507.
30. Blelloch GE, Leiserson CE, Maggs BM, Plaxton CG, Smith SJ, Zagha M. A Comparison of sorting algorithms for the connection machine CM-2. Paper presented at: Proceedings of the 3rd Symposium on Parallel Algorithms and Architectures (SPAA); 1991:3-16; ACM, New York, NY.
31. Gerbessiotis AV, Valiant LG. Direct bulk-synchronous parallel algorithms. *J Parall Distrib Comput*. 1994;22(2):251-267.
32. Axtmann M, Sanders P. Robust massively parallel sorting. Paper presented at: Proceedings of the 19th Workshop on Algorithm Engineering & Experiments (ALENEX); 2017:83-97; SIAM. preprint arXiv:1606.08766.
33. Leischner N, Osipov V, Sanders P. GPU sample sort. Paper presented at: Proceedings of the 24th IEEE International Symposium on Parallel & Distributed Processing (IPDPS); 2010:1-10; Atlanta, GA; IEEE.
34. Bingmann T, Sanders P. Parallel string sample sort. Paper presented at: Proceedings of the 21th European Symposium on Algorithms (ESA). 8125 of LNCS; 2013:169-180; Springer, New York, NY. preprint arXiv:1305.1157.
35. Bingmann T Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools [PhD thesis]. Karlsruhe Institute of Technology, Germany; 2018.
36. Musser DR. Introspective sorting and selection algorithms. *Softw Pract Exper*. 1997;27(8):983-993.
37. Elmasry A, Katajainen J, Stenmark M. Branch mispredictions don't affect mergesort. Paper presented at: Proceedings of the Symposium on Experimental Algorithms (SEA). 7276 of LNCS; 2012:160-171; Springer, New York, NY.

How to cite this article: Bingmann T, Marianczuk J, Sanders P. Engineering faster sorters for small sets of items. *Softw Pract Exper*. 2020;1–40. https://doi.org/10.1002/spe.2922

APPENDIX

TABLE A1 Average number of CPU cycles per iteration of OneArrayRepeat experiment on machine Intel-2650

	Overall		Array size														
	Rank	GeoM	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`IS Def`	25	2.17	15.4	41.9	78.8	119	170	207	258	293	354	394	452	487	561	582	650
`IS POp`	28	2.29	17.1	45.4	83.1	128	180	219	268	302	367	409	476	503	582	607	681
`IS STL`	31	2.41	24.9	53.0	90.1	133	184	222	275	310	379	411	489	511	587	608	678
`IS AIF`	32	2.63	23.8	51.0	90.9	150	215	264	319	357	416	461	534	564	629	650	723
`SN BN-L 4CmS`	1	1.09	12.7	21.5	37.6	53.7	70.9	83.5	**105**	135	164	199	238	267	315	322	**365**
`SN BN-L 4Cm`	3	1.11	9.46	24.3	**35.3**	55.3	72.2	73.4	107	136	174	213	249	289	346	351	405
`SN BN-L 2CPm`	11	1.31	8.95	24.6	38.5	62.4	87.6	127	146	187	227	252	275	335	386	396	433
`SN BN-L 2CPp`	15	1.53	6.78	35.3	40.5	78.4	113	141	166	213	262	322	350	396	449	483	504
`SN BN-L Tie`	21	2.09	19.7	39.0	66.4	97.0	137	171	213	267	328	372	446	525	617	657	764
`SN BN-L ISwp`	22	2.10	19.8	42.1	74.8	96.3	136	164	213	262	348	366	488	499	635	623	708
`SN BN-L JXhg`	26	2.25	18.6	40.3	69.7	108	144	186	236	307	380	408	492	583	678	714	777
`SN BN-P 4CmS`	6	1.13	12.8	**21.5**	37.5	49.0	67.0	83.1	110	137	185	215	246	288	333	358	395
`SN BN-P 4Cm`	7	1.15	9.00	24.4	35.4	**44.0**	70.7	84.0	112	133	195	226	270	319	360	396	452
`SN BN-P 2CPm`	10	1.20	6.83	25.2	39.1	65.3	85.2	110	136	175	207	222	249	285	338	353	397
`SN BN-P 2CPp`	13	1.37	**6.70**	34.9	40.3	84.3	101	128	147	210	218	250	282	326	376	395	447
`SN BN-P Tie`	18	2.03	18.2	38.9	66.2	96.1	134	162	210	251	320	369	440	478	587	662	763
`SN BN-P ISwp`	19	2.04	21.0	42.4	74.1	98.6	136	168	210	250	321	346	425	480	589	573	676
`SN BN-P JXhg`	29	2.31	18.4	39.0	72.1	119	153	192	244	299	379	451	506	586	682	726	825
`SN BN-R 4Cm`	4	1.11	9.00	24.0	36.1	54.1	71.8	**72.9**	110	153	199	209	**228**	289	324	351	408
`SN BN-R 4CmS`	8	1.17	13.0	23.0	36.4	54.1	**66.5**	79.8	124	164	219	227	267	280	315	344	408
`SN BN-R 2CPm`	14	1.45	11.4	28.1	46.6	74.0	113	126	150	182	237	268	303	349	406	423	491
`SN BN-R 2CPp`	16	1.69	13.8	31.2	52.0	91.0	126	135	161	260	290	304	355	384	477	501	575
`SN BN-R Tie`	23	2.12	19.2	38.9	68.6	101	134	169	215	291	346	397	463	531	595	657	751
`SN BN-R ISwp`	24	2.15	19.9	39.4	75.6	100	140	168	231	299	349	409	453	537	596	641	707
`SN BN-R JXhg`	30	2.34	18.9	40.5	72.6	110	152	190	262	311	411	446	530	570	682	708	811
`SN Best 4CmS`	2	1.10	13.4	21.7	37.4	52.8	70.5	83.5	105	136	174	**196**	234	258	324	331	371
`SN Best 4Cm`	5	1.11	9.27	24.2	35.4	53.8	72.5	79.7	107	**123**	**163**	211	256	291	363	377	418
`SN Best 2CPm`	9	1.18	9.33	24.3	38.7	62.5	88.0	128	146	149	178	210	234	**250**	**311**	**319**	372
`SN Best 2CPp`	12	1.35	7.10	35.2	40.9	78.9	113	142	166	195	191	251	274	291	350	363	412
`SN Best ISwp`	17	2.02	19.4	43.6	74.0	102	149	171	221	262	299	346	416	442	554	558	667
`SN Best Tie`	20	2.04	19.7	39.7	66.4	99.8	139	183	217	262	311	348	426	487	568	613	724
`SN Best JXhg`	27	2.28	19.0	39.7	72.2	115	158	191	248	290	351	405	512	573	663	703	806

TABLE A2 Average number of CPU cycles per iteration of OneArrayRepeat experiment on machine Intel-2670

	Overall		**Array size**														
	Rank	**GeoM**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
`IS Def`	22	2.21	8.67	33.9	66.0	103	135	169	205	246	281	321	377	421	466	500	526
`IS POp`	26	2.32	10.4	37.3	69.4	110	143	181	218	255	294	337	383	423	478	523	566
`IS STL`	31	2.59	18.0	52.7	78.2	117	159	198	239	276	322	387	446	486	522	572	611
`IS AIF`	32	2.69	17.1	51.0	78.5	131	179	222	260	298	334	396	446	493	524	562	612
`SN BN-L 4Cm`	1	1.08	2.70	13.1	22.6	39.0	45.6	**61.5**	**76.9**	105	126	178	204	262	300	312	348
`SN BN-L 4CmS`	3	1.09	3.39	21.2	**22.1**	41.7	49.1	64.5	77.8	111	124	172	190	240	256	282	299
`SN BN-L 2CPm`	11	1.31	**−0.93**	21.3	26.5	49.8	60.0	90.0	103	153	168	201	223	278	303	330	345
`SN BN-L 2CPp`	15	1.61	−0.05	30.6	31.9	67.0	79.0	110	125	189	214	246	273	347	360	392	415
`SN BN-L ISwp`	21	2.19	11.0	37.5	62.8	82.4	105	144	171	226	285	311	410	441	551	593	677
`SN BN-L Tie`	24	2.22	12.0	34.3	55.7	79.5	119	142	181	243	281	318	431	479	551	626	691
`SN BN-L JXhg`	27	2.38	11.0	34.0	61.7	99.0	128	172	199	264	313	337	421	504	579	657	693
`SN BN-P 4CmS`	7	1.14	3.62	20.9	22.2	38.8	45.7	64.6	79.9	112	148	182	200	256	285	330	339
`SN BN-P 4Cm`	8	1.14	2.59	12.8	22.4	**37.6**	**43.3**	61.8	79.4	110	161	195	221	282	318	356	394
`SN BN-P 2CPm`	10	1.22	−0.12	26.5	25.3	47.0	51.0	78.0	89.4	137	147	182	195	246	285	309	335
`SN BN-P 2CPp`	14	1.45	0.07	30.6	32.4	61.6	67.1	96.0	107	164	178	213	232	293	311	358	390
`SN BN-P ISwp`	18	2.13	10.4	38.6	61.3	85.2	104	140	173	217	258	304	382	428	510	558	631
`SN BN-P Tie`	20	2.17	12.0	33.7	55.0	81.5	105	139	192	229	266	317	407	472	541	604	664
`SN BN-P JXhg`	29	2.45	10.5	39.1	62.4	96.5	119	159	203	259	327	396	423	556	590	687	755
`SN BN-R 4Cm`	4	1.09	2.60	**12.0**	22.7	38.8	45.4	63.0	77.8	139	158	172	**176**	255	305	306	314
`SN BN-R 4CmS`	6	1.12	4.00	13.2	22.3	41.4	49.7	63.8	94.0	130	170	188	228	242	**234**	285	312
`SN BN-R 2CPm`	12	1.41	4.31	28.4	30.0	53.5	64.9	93.7	109	198	174	204	226	288	311	344	367
`SN BN-R 2CPp`	16	1.73	4.60	33.7	37.4	72.4	83.3	114	132	237	222	260	279	346	374	425	463
`SN BN-R Tie`	23	2.22	12.1	33.0	53.6	88.0	109	143	179	248	287	343	404	466	542	625	707
`SN BN-R ISwp`	25	2.25	10.9	36.3	61.1	85.7	117	142	194	244	282	338	403	490	541	595	676
`SN BN-R JXhg`	30	2.50	10.3	36.1	60.8	103	135	167	212	288	341	381	475	527	582	670	731
`SN Best 4CmS`	2	1.09	3.66	20.7	22.6	42.3	49.6	64.8	78.2	109	**121**	**165**	181	228	274	286	308
`SN Best 4Cm`	5	1.10	2.95	13.7	22.1	38.9	46.1	61.6	77.8	**105**	125	181	212	267	309	339	359
`SN Best 2CPm`	9	1.18	−0.72	21.7	26.4	49.4	60.2	90.0	103	130	132	169	189	**215**	253	**278**	**294**
`SN Best 2CPp`	13	1.43	−0.12	30.9	32.6	67.4	78.3	110	125	157	157	193	218	255	298	324	358
`SN Best ISwp`	17	2.10	10.8	35.7	63.3	90.3	114	152	181	223	240	300	345	389	478	533	609
`SN Best Tie`	19	2.16	12.0	35.1	56.4	82.7	110	150	193	234	253	306	388	449	504	599	632
`SN Best JXhg`	28	2.41	10.9	34.9	63.3	104	133	182	204	250	285	358	421	495	588	661	726

TABLE A3 Average number of CPU cycles per iteration of OneArrayRepeat experiment on machine Ryzen-1800X

	Overall		Array size														
	Rank	**GeoM**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
`IS Def`	19	4.14	15.0	46.9	80.3	125	167	220	258	290	327	369	397	446	464	510	559
`IS POp`	21	4.23	10.9	45.8	84.9	134	183	228	266	298	339	382	414	462	495	555	591
`IS STL`	31	5.09	21.1	63.3	103	150	200	255	298	340	392	433	485	536	577	629	695
`IS AIF`	32	5.27	22.3	59.5	103	149	201	264	314	364	415	459	516	557	599	662	747
`SN BN-L 4Cm`	1	1.08	6.13	10.3	14.8	27.4	35.7	**44.8**	56.8	**70.0**	**82.4**	**99.5**	113	**145**	156	163	**181**
`SN BN-L 4CmS`	2	1.14	6.72	13.6	15.1	28.4	35.3	53.9	60.0	73.8	84.0	109	**113**	145	154	**154**	186
`SN BN-L 2CPm`	11	1.82	2.86	17.6	26.3	52.7	54.6	88.3	86.3	133	167	200	222	272	285	317	335
`SN BN-L 2CPp`	15	2.10	3.15	27.2	25.6	70.1	72.0	90.8	94.5	162	201	236	251	295	313	341	351
`SN BN-L JXhg`	17	3.92	15.1	37.0	65.9	91.8	125	161	205	275	325	372	425	497	561	619	691
`SN BN-L Tie`	23	4.24	15.1	40.0	76.3	103	133	189	223	292	337	405	451	527	590	671	791
`SN BN-L ISwp`	27	4.33	14.2	41.0	76.8	104	130	176	219	304	370	404	494	541	661	724	782
`SN BN-P 4Cm`	4	1.18	6.69	10.9	**14.4**	25.7	**33.4**	46.6	**55.2**	75.0	99.4	106	127	173	178	214	231
`SN BN-P 4CmS`	7	1.24	6.65	13.1	15.2	**24.3**	39.0	51.8	59.1	80.2	93.7	130	125	154	183	240	231
`SN BN-P 2CPm`	9	1.61	**2.55**	17.0	20.3	49.9	48.8	79.0	92.4	127	140	171	186	227	237	270	284
`SN BN-P 2CPp`	13	1.87	3.12	26.8	25.2	68.2	61.6	84.1	90.4	147	156	185	206	242	266	295	311
`SN BN-P JXhg`	20	4.23	14.4	42.6	68.9	99.1	128	168	227	277	350	420	466	549	588	733	802
`SN BN-P Tie`	24	4.25	16.2	34.1	63.2	97.7	140	199	226	295	347	402	461	535	631	725	790
`SN BN-P ISwp`	26	4.32	13.9	40.3	73.9	106	143	191	237	309	338	418	467	538	622	715	752
`SN BN-R 4Cm`	3	1.15	5.87	11.2	14.8	25.6	35.0	48.5	56.2	93.4	93.7	113	124	148	**153**	187	202
`SN BN-R 4CmS`	8	1.29	6.06	10.9	15.2	28.8	35.8	59.8	71.8	89.8	132	157	161	147	156	197	216
`SN BN-R 2CPm`	14	2.06	8.12	19.1	26.8	55.5	71.8	87.0	93.9	140	179	209	225	287	298	325	347
`SN BN-R 2CPp`	16	2.38	8.72	31.5	28.3	75.2	75.2	92.6	107	180	209	244	256	299	321	354	390
`SN BN-R JXhg`	18	4.10	19.4	42.4	57.6	96.5	132	177	221	289	334	382	442	504	557	635	667
`SN BN-R Tie`	25	4.31	15.6	37.1	61.9	111	146	193	228	322	372	426	464	553	607	684	709
`SN BN-R ISwp`	29	4.37	14.3	41.3	80.4	99.8	141	189	252	328	359	415	468	529	576	702	790
`SN Best 4Cm`	5	1.19	8.05	**10.2**	14.8	28.8	36.1	45.2	59.3	74.3	85.9	104	126	164	188	206	233
`SN Best 4CmS`	6	1.23	7.30	13.6	16.1	29.7	36.4	55.2	63.9	74.7	87.6	103	117	162	185	211	224
`SN Best 2CPm`	10	1.62	3.27	17.5	20.4	53.3	55.2	80.8	86.5	130	129	162	202	200	226	264	276
`SN Best 2CPp`	12	1.86	3.84	27.0	25.9	70.5	72.1	91.1	94.7	142	139	170	196	214	249	274	292
`SN Best JXhg`	22	4.24	16.0	39.8	70.8	102	137	180	223	278	333	397	437	543	634	714	774
`SN Best Tie`	28	4.34	16.6	39.2	72.5	106	138	205	242	294	337	386	455	529	598	715	814
`SN Best ISwp`	30	4.41	17.2	37.5	76.1	120	149	194	247	309	339	410	457	517	608	681	777

TABLE A4 Average number of nanoseconds per iteration of OneArrayRepeat experiment on machine RK3399

	Overall		**Array size**														
	Rank	**GeoM**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
IS Def	20	1.84	17.6	38.4	56.5	86.9	114	138	172	205	249	284	326	369	409	465	513
IS POp	22	2.13	26.7	39.5	61.9	94.8	127	162	195	235	284	331	373	433	485	539	595
IS AIF	25	2.57	42.6	69.3	88.3	122	154	184	221	271	315	363	408	467	522	583	647
IS STL	26	2.59	37.4	62.2	94.1	123	155	194	232	268	324	369	422	472	531	592	656
SN BN-L 4Cm	3	1.07	22.2	31.5	**38.8**	51.7	63.0	**68.2**	87.4	106	129	150	167	189	202	239	256
SN BN-L 4CmS	5	1.45	26.1	36.2	45.5	66.3	83.0	103	110	146	180	205	233	270	321	377	384
SN BN-L ISwp	10	1.54	23.4	34.4	48.0	66.7	87.0	103	121	176	205	246	266	308	341	372	417
SN BN-L 2CPp	11	1.55	21.1	35.9	50.8	68.3	88.1	103	125	172	205	241	266	304	339	374	419
SN BN-L JXhg	14	1.57	27.4	33.7	53.6	75.2	84.2	105	130	169	204	241	260	303	329	372	405
SN BN-L 2CPm	24	2.56	24.7	52.0	66.9	113	141	188	222	307	359	428	461	561	613	675	728
SN BN-L Tie	31	3.40	31.1	56.5	82.4	129	174	219	256	427	511	651	717	829	904	1028	1040
SN BN-P 4Cm	1	1.03	**17.1**	**28.6**	40.4	51.7	**59.6**	72.8	**79.6**	109	126	151	**163**	190	204	233	249
SN BN-P 2CPp	7	1.54	22.9	33.3	46.1	70.1	86.1	110	123	171	212	233	266	300	346	372	399
SN BN-P ISwp	8	1.54	22.0	32.0	47.3	70.8	86.0	111	130	169	206	240	266	299	350	370	401
SN BN-P JXhg	17	1.60	30.4	36.9	51.2	73.8	93.8	114	130	174	195	232	283	292	329	369	406
SN BN-P 4CmS	18	1.64	30.2	37.5	49.2	63.9	79.5	105	118	156	198	248	289	347	409	463	497
SN BN-P 2CPm	27	2.62	26.2	50.5	72.5	108	143	188	216	309	369	448	495	576	637	716	758
SN BN-P 2CPm	27	2.62	26.2	50.5	72.5	108	143	188	216	309	369	448	495	576	637	716	758
SN BN-P Tie	32	3.85	46.4	67.5	95.4	171	221	283	323	465	552	654	717	846	935	1056	939
SN BN-R 4Cm	4	1.25	27.6	29.7	41.8	52.6	61.7	70.6	86.7	129	185	228	271	244	258	256	274
SN BN-R ISwp	13	1.56	21.5	31.6	48.6	68.7	87.2	107	133	180	218	248	275	316	344	386	416
SN BN-R 2CPp	16	1.57	25.3	34.3	46.1	66.8	86.1	103	132	184	218	254	272	304	342	384	413
SN BN-R JXhg	19	1.68	27.5	42.9	52.6	69.4	86.6	109	152	189	229	269	281	315	347	392	440
SN BN-R 4CmS	21	1.89	31.5	38.6	48.7	61.1	84.0	125	169	198	243	290	346	426	484	574	634
SN BN-R 2CPm	28	2.66	29.5	49.9	72.7	108	142	191	237	304	364	427	479	576	647	730	800
SN BN-R Tie	29	3.17	36.5	69.0	85.3	134	174	221	261	372	444	533	572	706	747	798	847
SN Best 4Cm	2	1.04	25.6	29.5	41.3	**50.8**	61.7	69.4	84.7	**106**	**120**	**149**	164	**165**	**198**	**217**	**236**
SN Best 2CPp	6	1.50	22.8	39.6	46.7	72.5	87.0	107	125	165	187	242	253	276	303	341	379
SN Best JXhg	9	1.54	29.1	37.2	50.7	73.7	86.3	105	130	162	201	221	245	283	315	355	377
SN Best ISwp	12	1.55	31.7	37.4	49.0	74.8	91.2	107	137	166	178	239	253	280	305	346	381
SN Best 4CmS	15	1.57	26.4	41.8	49.8	61.9	86.3	107	110	154	183	229	270	304	379	423	476
SN Best 2CPm	23	2.54	30.4	52.9	70.3	110	142	193	219	283	341	408	453	527	597	659	701
SN Best Tie	30	3.32	34.0	60.3	89.6	130	175	224	260	423	462	591	670	705	830	956	1002