A new implementation for std::sort

Presenters: Nilay Vaish, Daniel J. Mankowitz

Contributors: Marco Gelmi, MinJae Hwang, Danila Kutenin, Andrea Michi, Marco Selvi

Agenda

- Introsort
- BlockQuickSort
- Reinforcement Learning-based Small Sort

Prior Implementation


```
void __sort(__first, __last, __comp)
{
    while (true)
    {
        // Sort small lengths and break from the loop
        .
        .
        .
        // Choose pivot
        .
        .
        .
        // Partition the current range into two parts around the pivot
        while (true) {
          // Find next two elements to swap and swap them
        }
        // sort one partition recursively, the other iteratively
    }
}
```

Prior Implementation

```
void __sort(__first, __last, __comp)
{
    while (true)
    {
        // Sort small lengths and break from the loop
        .
        .
        .
        // Choose pivot
        .
        .
        .
        // Partition the current range into two parts around the pivot
        while (true) {
          // Find next two elements to swap and swap them
        }
        // sort one partition recursively, the other iteratively
    }
}
```

Prior Implementation

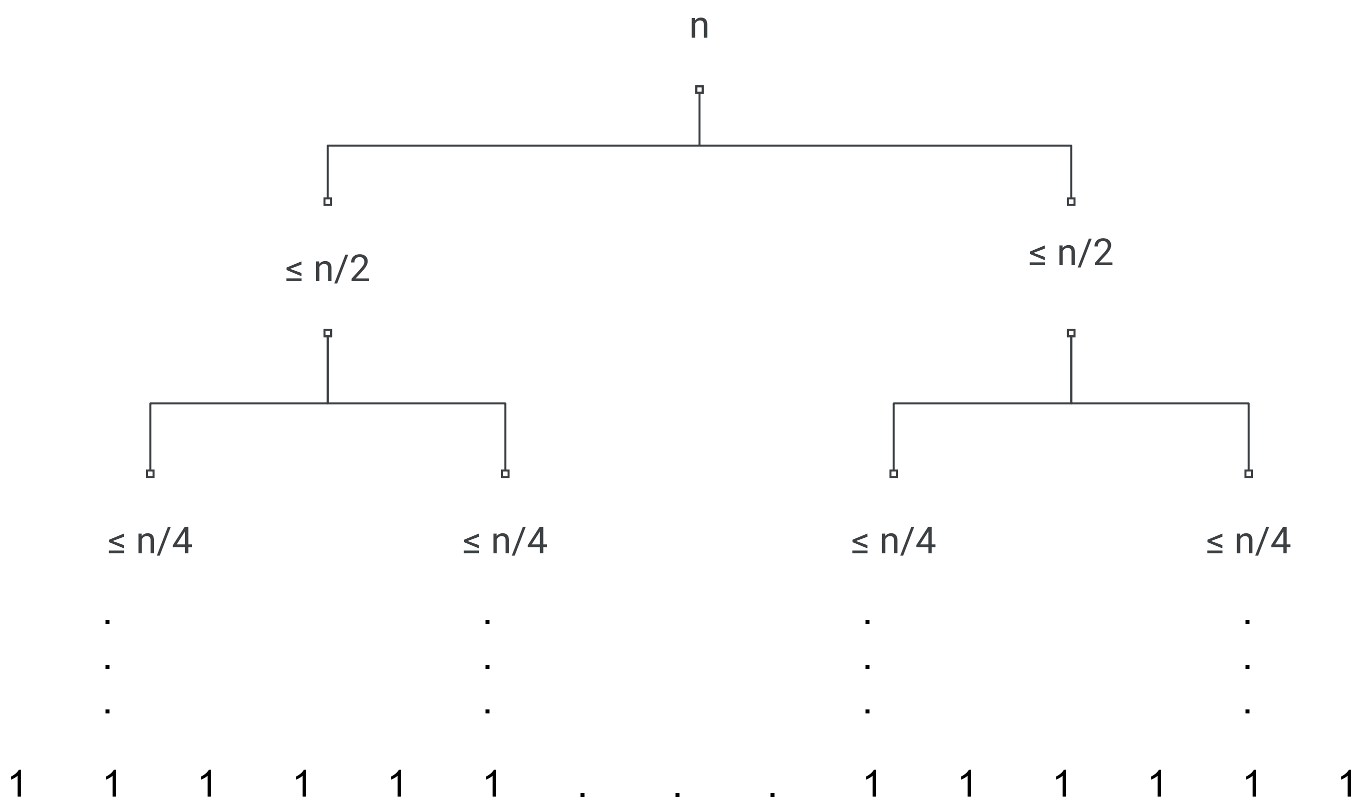
```
void __sort(__first, __last, __comp)
{
    while (true)
    {
        // Sort small lengths and break from the loop
        .
        .
        .
        // Choose pivot
        .
        .
        .
        // Partition the current range into two parts around the pivot
        while (true) {
          // Find next two elements to swap and swap them
        }
        // sort one partition recursively, the other iteratively
    }
}
```

Prior Implementation

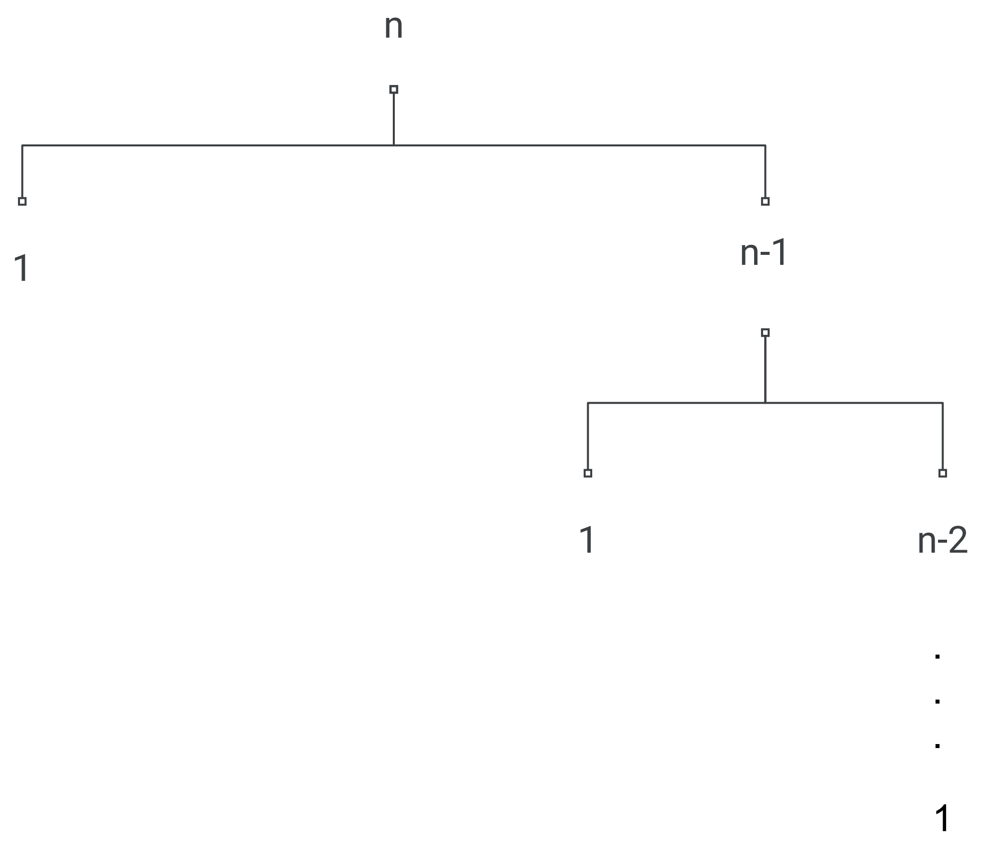
```
void __sort(__first, __last, __comp)
{
    while (true)
    {
        // Sort small lengths and break from the loop
        .
        .
        .
        // Choose pivot
        .
        .
        .
        // Partition the current range into two parts around the pivot
        while (true) {
          // Find next two elements to swap and swap them
        }
        // sort one partition recursively, the other iteratively
    }
}
```

Introsort

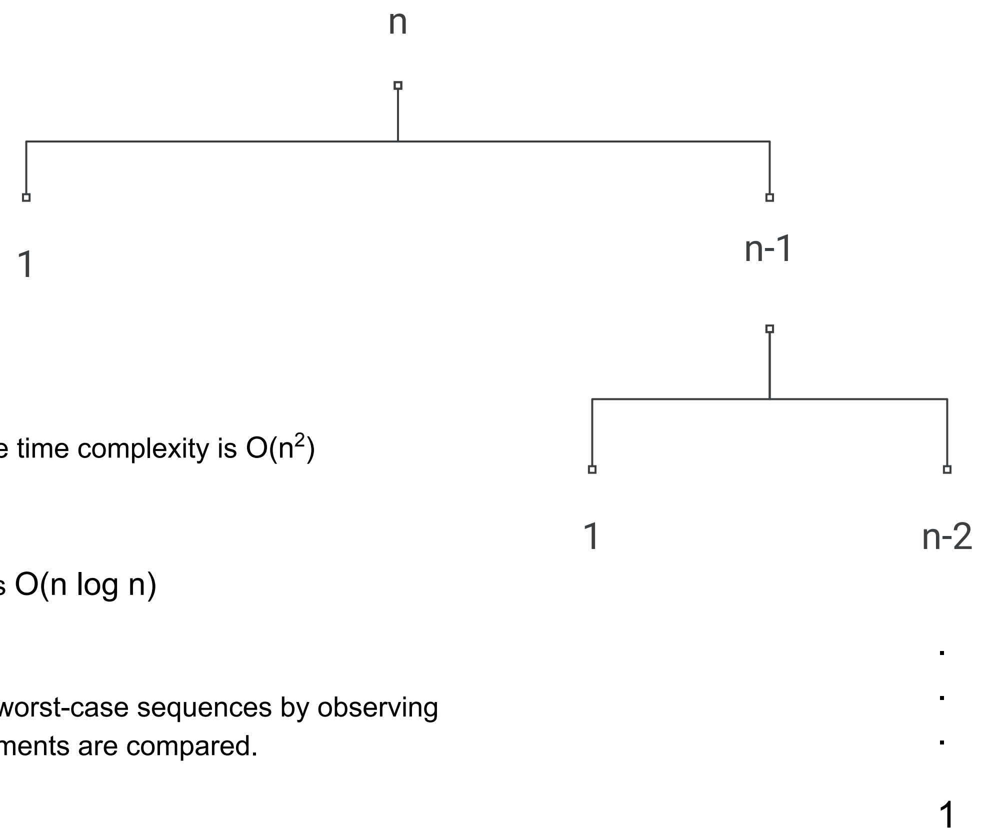
Quicksort: best case



Quicksort: worst case



Quicksort: worst case



- Quicksort's worst-case time complexity is $O(n^2)$
- C++ standard requires $O(n \log n)$
- Possible to construct worst-case sequences by observing the order in which elements are compared.

Improve the worst case: Introsort

```
void __introsort(__first, __last, __comp, __depth)
{
  while (true) {
    if (__depth == 0) {
      // Fallback to heap sort as Introsort suggests.
      _VSTD::__partial_sort<_Compare>(__first, __last, __last);
      return;
    }
    --__depth;
    // Same sorting algorithm as shown earlier.
}
```

Improve the worst case: Introsort

```
void __introsort(__first, __last, __comp, __depth)
{
  while (true) {
    if (__depth == 0) {
      // Fallback to heap sort as Introsort suggests.
      _VSTD::__partial_sort<_Compare>(__first, __last, __last);
      return;
    }
    --__depth;
    // Same sorting algorithm as shown earlier.
}
```

Improve the worst case: Introsort

```
void __introsort(__first, __last, __comp, __depth)
{
  while (true) {
    if (__depth == 0) {
      // Fallback to heap sort as Introsort suggests.
      _VSTD::__partial_sort<_Compare>(__first, __last, __last, __comp);
      return;
    }
    --__depth;
    // Same sorting algorithm as shown earlier.
}
```

Microbenchmark Results

Benchmark	Sorting time per element (ns)	
	Quicksort	Introsort
BM_Sort_uint32_QuickSortAdversary_64	33	45
BM_Sort_uint32_QuickSortAdversary_256	132	69
BM_Sort_uint32_QuickSortAdversary_1024	498	118
BM_Sort_uint32_QuickSortAdversary_16384	3846	175
BM_Sort_uint32_QuickSortAdversary_262144	61431	210

BlockQuickSort

Branches in Quicksort

```
    // Choose pivot
    // Partition the current range into two parts around the pivot
    while (true) {
        while (__comp(*++__i, *__pivot));
        while (!__comp(*--__j, *__pivot));
        if (__i > __j) break;
        swap(*__i, *__j);
    }
```

Branches in Quicksort

```
    // Choose pivot
    // Partition the current range into two parts around the pivot
    while (true) {
        while (__comp(*++__i, *__pivot));
        while (!__comp(*--__j, *__pivot));
        if (__i > __j) break;
        swap(*__i, *__j);
    }
```

Branches in Quicksort

```
    // Choose pivot
    // Partition the current range into two parts around the pivot
    while (true) {
        while (__comp(*++__i, *__pivot));
        while (!__comp(*--__j, *__pivot));
        if (__i > __j) break;
        swap(*__i, *__j);
    }
```

- Outcome of comparison used for branching
- Data dependent branches are hard to predict
- BlockQuickSort reduces branches by separating the data movement from the comparison operation

Reduce Branches: BlockQuickSort

```
uint64_t __left_bitset = 0;
_RandomAccessIterator __iter = __first;
for (int __j = 0; __j < __block_size;) {
  bool __comp_result = !__comp(*__iter, __pivot);
  __left_bitset |= (__comp_result << __j);
  __j++;
  ++__iter;
}
```

Reduce Branches: BlockQuickSort

```
    uint64_t __left_bitset = 0;
    _RandomAccessIterator __i = __first;
    for (int __b = 0; __b < __block_size;) {
      bool __comp_result = !__comp(*__i, __pivot);
      __left_bitset |= (__comp_result << __b);
      __b++;
      ++__i;
    }
```

Reduce Branches: BlockQuickSort

```
void __swap_bitset_pos(__first, __last, __left_bitset, __right_bitset) {
  while (__left_bitset != 0 && __right_bitset != 0) {
    difference_type tz_left = __ctz(__left_bitset);
    __left_bitset = __blsr(__left_bitset);
    difference_type tz_right = __ctz(__right_bitset);
    __right_bitset = __blsr(__right_bitset);
    _VSTD::iter_swap(__first + tz_left, __last - tz_right);
  }
}
```

Reduce Branches: BlockQuickSort

```
void __swap_bitset_pos(__first, __last, __left_bitset, __right_bitset) {
  while (__left_bitset != 0 && __right_bitset != 0) {
    difference_type tz_left = __ctz(__left_bitset);
    __left_bitset = __blsr(__left_bitset);
    difference_type tz_right = __ctz(__right_bitset);
    __right_bitset = __blsr(__right_bitset);
    _VSTD::iter_swap(__first + tz_left, __last - tz_right);
  }
}
```

Reduce Branches: BlockQuickSort

```
void __swap_bitset_pos(__first, __last, __left_bitset, __right_bitset) {
  while (__left_bitset != 0 && __right_bitset != 0) {
    difference_type tz_left = __ctz(__left_bitset);
    __left_bitset = __blsr(__left_bitset);
    difference_type tz_right = __ctz(__right_bitset);
    __right_bitset = __blsr(__right_bitset);
    _VSTD::iter_swap(__first + tz_left, __last - tz_right);
  }
}
```

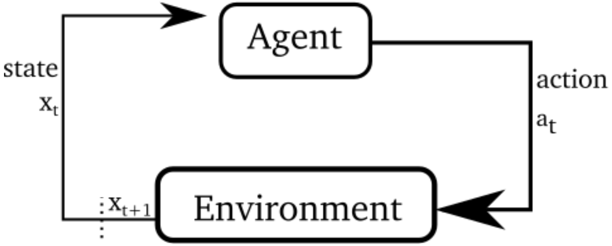
Microbenchmark Results

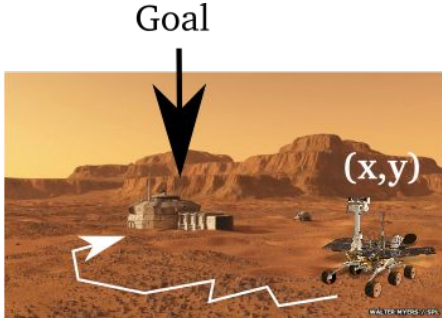
Benchmark	Sorting time per element (ns)	
	Quicksort	BlockQuickSort
BM_Sort_uint32_Random_64	18.6	18.5
BM_Sort_uint32_Random_256	26.2	21.3
BM_Sort_uint32_Random_1024	33.4	23.3
BM_Sort_uint32_Random_16384	47.7	26.7
BM_Sort_uint32_Random_262144	62.6	30.1

Small Sort Optimization with ML

What is RL?

Episode (state s_0, action a_0, next state s_1, next action a_1, ...)





What do we want?

A **policy** π : what action a should I take in state s?

Deterministic: a = π(s)

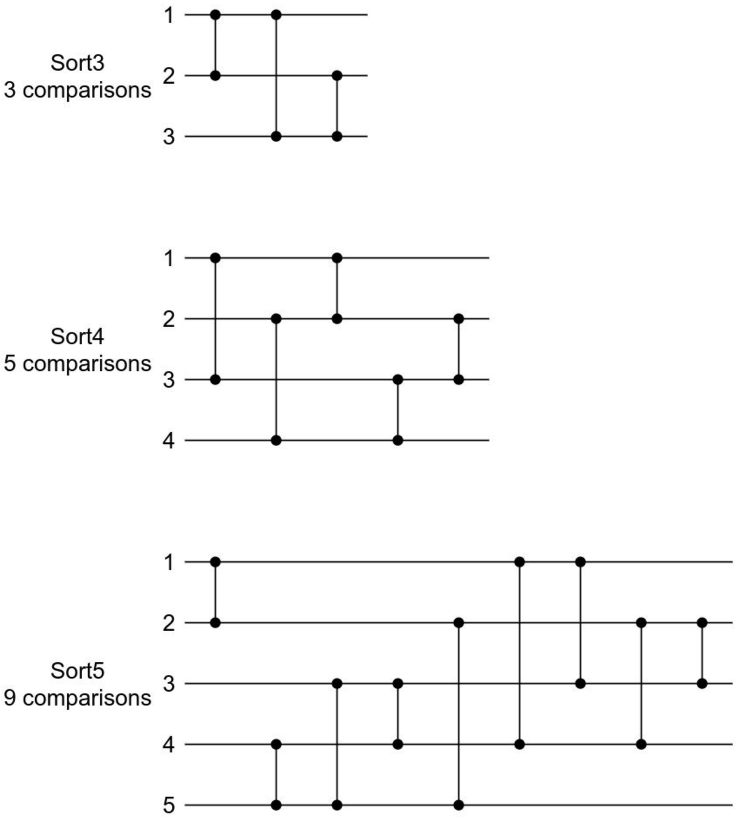
Stochastic: a ~ π(a|s)

Reinforcement Learning System

- Defined by a Markov Decision Process
  - States
    - Assembly program generated thus far
    - State of memory and registers
  - Actions
    - Assembly instructions - Intel AT&T syntax
  - Rewards
    - Correctness
    - Latency/program length

- Goal: Learn a policy that finds a correct, low latency program

Sorting Networks*



- https://danlark.org/2022/04/20/changing-stdsort-at-googles-scale-and-beyond/

Conditional Swap

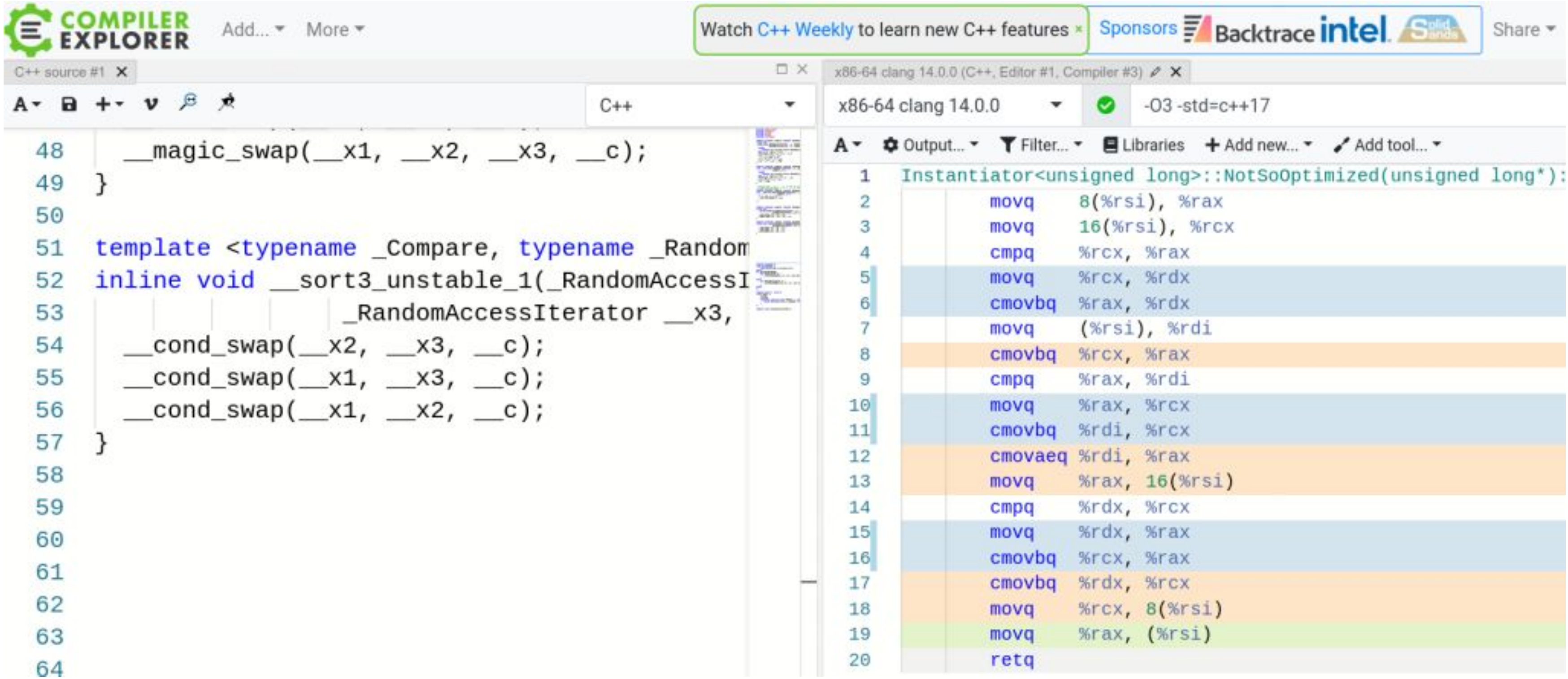
```cpp
#include <functional>

void cond_swap(int* __x, int* __y,
               std::less<int> __c) {
  bool __r = __c(*__x, *__y);
  int __tmp = __r ? *__x : *__y;
  *__y = __r ? *__y : *__x;
  *__x = __tmp;
}
```

x86-64 clang 14.0.0 — -O3 -std=c++17 -stdlib=libc++

```asm
cond_swap(int*, int*, std::__1::less<int>):
        movl    (%rdi), %eax      # From pointers
        movl    (%rsi), %ecx
        cmpl    %ecx, %eax        # Compare
        movl    %ecx, %edx        # Temporary
        cmovll  %eax, %edx        # Swap if less
        cmovll  %ecx, %eax
        movl    %eax, (%rsi)      # To pointers
        movl    %edx, (%rdi)
        retq
```

Sort3 with Condition Swap



Special Swap

```cpp
// Ensures that *__x, *__y and *__z are ordered according to the comparator __c,
// under the assumption that *__y and *__z are already ordered.
template <class _Compare, class _RandomAccessIterator>
inline void __partially_sorted_swap(_RandomAccessIterator __x, _RandomAccessIterator __y,
                                    _RandomAccessIterator __z, _Compare __c) {
  using value_type = typename iterator_traits<_RandomAccessIterator>::value_type;
  bool __r = __c(*__z, *__x);
  value_type __tmp = __r ? *__z : *__x;
  *__z = __r ? *__x : *__z;
  __r = __c(__tmp, *__y);
  *__x = __r ? *__x : *__y;
}

template <class _Compare, class _RandomAccessIterator>
inline void __sort3(_RandomAccessIterator __x1, _RandomAccessIterator __x2,
                    _RandomAccessIterator __x3, _Compare __c) {
  _VSTD::__cond_swap<_Compare>(__x2, __x3, __c);
  _VSTD::__partially_sorted_swap<_Compare>(__x1, __x2, __x3, __c);
}
```

Special Swap

```
// Ensures that *__x, *__y and *__z are ordered according to the comparator __c,
// under the assumption that *__y and *__z are already ordered.
template <class _Compare, class _RandomAccessIterator>
inline void __partially_sorted_swap(_RandomAccessIterator __x, _RandomAccessIterator __y,
                                    _RandomAccessIterator __z, _Compare __c) {
  using value_type = typename iterator_traits<_RandomAccessIterator>::value_type;
  bool __r = __c(*__z, *__x);
  value_type __tmp = __r ? *__z : *__x;
  *__z = __r ? *__x : *__z;
  __r = __c(__tmp, *__y);
  *__x = __r ? *__x : *__y;
}

template <class _Compare, class _RandomAccessIterator>
inline void __sort3(_RandomAccessIterator __x1, _RandomAccessIterator __x2,
                    _RandomAccessIterator __x3, _Compare __c) {
  _VSTD::__cond_swap<_Compare>(__x2, __x3, __c);
  _VSTD::__partially_sorted_swap<_Compare>(__x1, __x2, __x3, __c);
}
```

Special Swap

```
// Ensures that *__x, *__y and *__z are ordered according to the comparator __c,
// under the assumption that *__y and *__z are already ordered.
template <class _Compare, class _RandomAccessIterator>
inline void __partially_sorted_swap(_RandomAccessIterator __x, _RandomAccessIterator __y,
                                    _RandomAccessIterator __z, _Compare __c) {
  using value_type = typename iterator_traits<_RandomAccessIterator>::value_type;
  bool __r = __c(*__z, *__x);
  value_type __tmp = __r ? *__z : *__x;
  *__z = __r ? *__x : *__z;
  __r = __c(__tmp, *__y);
  *__x = __r ? *__x : *__y;
}

template <class _Compare, class _RandomAccessIterator>
inline void __sort3(_RandomAccessIterator __x1, _RandomAccessIterator __x2,
                    _RandomAccessIterator __x3, _Compare __c) {
  _VSTD::__cond_swap<_Compare>(__x2, __x3, __c);
  _VSTD::__partially_sorted_swap<_Compare>(__x1, __x2, __x3, __c);
}
```

Special Swap

Diff Viewer x86-64 clang 14.0.0 vs x86-64 clang 14.0.0

Left: x86-64 clang (trunk) -O3 -s... | Assembly

```
Instantiator<unsigned long>::NotSoOptimized(unsigned long*):
        movq    8(%rsi), %rax
        movq    16(%rsi), %rcx
        cmpq    %rcx, %rax
        movq    %rcx, %rdx
        cmovbq  %rax, %rdx
        movq    (%rsi), %rdi
        cmovbq  %rcx, %rax
        cmpq    %rax, %rdi
        movq    %rax, %rcx
        cmovbq  %rdi, %rcx
        cmovaeq %rdi, %rax
        movq    %rax, 16(%rsi)
        cmpq    %rdx, %rcx
        movq    %rdx, %rax
        cmovbq  %rcx, %rax
        cmovbq  %rdx, %rcx
        movq    %rcx, 8(%rsi)
        movq    %rax, (%rsi)
        retq
```

Right: x86-64 clang (trunk) -O3 -s... | Assembly

```
Instantiator<unsigned long>::Optimized(unsigned long*):
        movq    8(%rsi), %rax
        movq    16(%rsi), %rcx
        cmpq    %rcx, %rax
        movq    %rcx, %rdx
        cmovbq  %rax, %rdx
        movq    (%rsi), %rdi
        cmovbq  %rcx, %rax
        cmpq    %rdi, %rax
        movq    %rdi, %rcx
        cmovbq  %rax, %rcx
        cmovbq  %rdi, %rax
        movq    %rax, 16(%rsi)
        cmpq    %rdx, %rcx
        cmovaeq %rdx, %rdi
        movq    %rdi, (%rsi)
        cmovbq  %rdx, %rcx
        movq    %rcx, 8(%rsi)
        retq
```

Special Swap

Diff Viewer x86-64 clang 14.0.0 vs x86-64 clang 14.0.0

A Left: x86-64 clang (trunk) -O3 -s... Assembly | Right: x86-64 clang (trunk) -O3 -s... Assembly

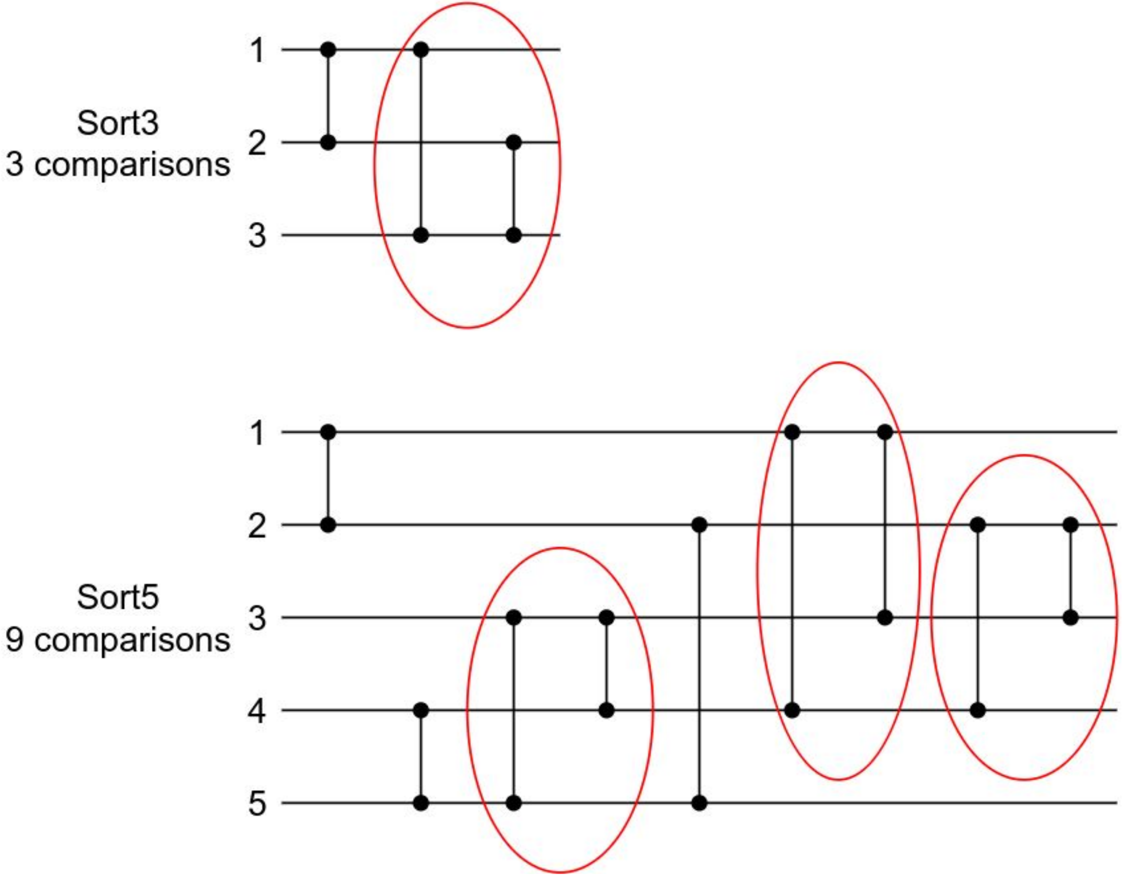
Left:

```
Instantiator<unsigned long>::NotSoOptimized(unsigned long*)
        movq    8(%rsi), %rax
        movq    16(%rsi), %rcx
        cmpq    %rcx, %rax
        movq    %rcx, %rdx
        cmovbq  %rax, %rdx
        movq    (%rsi), %rdi
        cmovbq  %rcx, %rax
        cmpq    %rax, %rdi
        movq    %rax, %rcx
        cmovbq  %rdi, %rcx
        cmovaeq %rdi, %rax
        movq    %rax, 16(%rsi)
        cmpq    %rdx, %rcx
        movq    %rdx, %rax
        cmovbq  %rcx, %rax
        cmovbq  %rdx, %rcx
        movq    %rcx, 8(%rsi)
        movq    %rax, (%rsi)
        retq
```

Right:

```
Instantiator<unsigned long>::Optimized(unsigned long*):
        movq    8(%rsi), %rax
        movq    16(%rsi), %rcx
        cmpq    %rcx, %rax
        movq    %rcx, %rdx
        cmovbq  %rax, %rdx
        movq    (%rsi), %rdi
        cmovbq  %rcx, %rax
        cmpq    %rdi, %rax
        movq    %rdi, %rcx
        cmovbq  %rax, %rcx
        cmovbq  %rdi, %rax
        movq    %rax, 16(%rsi)
        cmpq    %rdx, %rcx
        cmovaeq %rdx, %rdi
        movq    %rdi, (%rsi)
        cmovbq  %rdx, %rcx
        movq    %rcx, 8(%rsi)
        retq
```

Multiple Special Swaps



Micro-benchmarks

ARMv8

name	old cpu/op	new cpu/op	delta	
BM_Sort_uint32_Random_1	3.85ns ± 0%	4.01ns ± 0%	+3.95%	(p=0.000 n=99+63)
BM_Sort_uint32_Random_4	4.83ns ± 0%	2.05ns ± 0%	-57.50%	(p=0.000 n=97+94)
BM_Sort_uint32_Random_16	9.59ns ± 0%	9.15ns ± 0%	-4.59%	(p=0.000 n=98+95)
BM_Sort_uint32_Random_64	16.2ns ± 0%	15.7ns ± 0%	-2.99%	(p=0.000 n=94+99)
BM_Sort_uint32_Random_256	22.3ns ± 1%	21.7ns ± 0%	-2.77%	(p=0.000 n=100+100)
BM_Sort_uint32_Random_1024	28.5ns ± 0%	27.7ns ± 0%	-2.64%	(p=0.000 n=99+100)
BM_Sort_uint32_Random_16384	40.3ns ± 1%	39.4ns ± 1%	-2.17%	(p=0.000 n=98+100)
BM_Sort_uint32_Random_262144	51.8ns ± 2%	50.9ns ± 2%	-1.69%	(p=0.000 n=100+100)
BM_Sort_uint64_Random_1	4.02ns ± 0%	3.93ns ± 2%	-2.32%	(p=0.000 n=96+95)
BM_Sort_uint64_Random_4	5.03ns ± 0%	2.18ns ± 0%	-56.68%	(p=0.000 n=95+96)
BM_Sort_uint64_Random_16	9.63ns ± 0%	9.22ns ± 0%	-4.32%	(p=0.000 n=98+98)
BM_Sort_uint64_Random_64	16.2ns ± 0%	15.9ns ± 0%	-2.18%	(p=0.000 n=100+99)
BM_Sort_uint64_Random_256	22.4ns ± 0%	22.1ns ± 0%	-1.49%	(p=0.000 n=98+98)
BM_Sort_uint64_Random_1024	28.4ns ± 0%	28.0ns ± 0%	-1.16%	(p=0.000 n=98+100)
BM_Sort_uint64_Random_16384	40.0ns ± 1%	39.7ns ± 1%	-0.81%	(p=0.000 n=96+99)
BM_Sort_uint64_Random_262144	51.6ns ± 2%	51.4ns ± 2%	-0.48%	(p=0.000 n=98+99)

Micro-benchmarks

ARMv8

```
name                           old cpu/op   new cpu/op   delta
BM_Sort_uint32_Random_1         3.85ns ± 0%  4.01ns ± 0%   +3.95%   (p=0.000 n=99+63)
BM_Sort_uint32_Random_4         4.83ns ± 0%  2.05ns ± 0%  -57.50%   (p=0.000 n=97+94)
BM_Sort_uint32_Random_16        9.59ns ± 0%  9.15ns ± 0%   -4.59%   (p=0.000 n=98+95)
BM_Sort_uint32_Random_64        16.2ns ± 0%  15.7ns ± 0%   -2.99%   (p=0.000 n=94+99)
BM_Sort_uint32_Random_256       22.3ns ± 1%  21.7ns ± 0%   -2.77%  (p=0.000 n=100+100)
BM_Sort_uint32_Random_1024      28.5ns ± 0%  27.7ns ± 0%   -2.64%   (p=0.000 n=99+100)
BM_Sort_uint32_Random_16384     40.3ns ± 1%  39.4ns ± 1%   -2.17%   (p=0.000 n=98+100)
BM_Sort_uint32_Random_262144    51.8ns ± 2%  50.9ns ± 2%   -1.69%  (p=0.000 n=100+100)

BM_Sort_uint64_Random_1         4.02ns ± 0%  3.93ns ± 2%   -2.32%   (p=0.000 n=96+95)
BM_Sort_uint64_Random_4         5.03ns ± 0%  2.18ns ± 0%  -56.68%   (p=0.000 n=95+96)
BM_Sort_uint64_Random_16        9.63ns ± 0%  9.22ns ± 0%   -4.32%   (p=0.000 n=98+98)
BM_Sort_uint64_Random_64        16.2ns ± 0%  15.9ns ± 0%   -2.18%   (p=0.000 n=100+99)
BM_Sort_uint64_Random_256       22.4ns ± 0%  22.1ns ± 0%   -1.49%   (p=0.000 n=98+98)
BM_Sort_uint64_Random_1024      28.4ns ± 0%  28.0ns ± 0%   -1.16%   (p=0.000 n=98+100)
BM_Sort_uint64_Random_16384     40.0ns ± 1%  39.7ns ± 1%   -0.81%   (p=0.000 n=96+99)
BM_Sort_uint64_Random_262144    51.6ns ± 2%  51.4ns ± 2%   -0.48%   (p=0.000 n=98+99)
```

Thank You

Google

References

1. DAVID R. MUSSER. Introspective Sorting and Selection Algorithms.

2. Stefan Edelkamp and Armin Weiß. 2019. BlockQuicksort: Avoiding Branch Mispredictions in Quicksort. ACM J. Exp. Algorithmics 24, Article 1.4 (2019), 22 pages.

3. Danila Kutenin. Changing std::sort at Google's Scale and Beyond.