Design of quickselect

Bruce Lilly
bruce.lilly@gmail.com

March 16, 2019

Abstract

Hacker's guide to *quickselect*. Is that abstract enough?

1 Introduction

Quickselect, in the *libmedian* library, efficiently implements multiple quickselect, *qsort*, C11[1] *qsort_s*, and a similar variant of multiple quickselect *quickselect_s*. This paper will explain the design of the library code, its configuration and build process, and the location of the various source files which comprise the source code. The objective is to explain the rationale for the source code layout and to enable the reader to configure, build, use, understand, and possibly modify the library functions.

The C11 variants suffixed with *_s* exhibit two variations from standard *qsort* and *quickselect*:

1. Run-time bounds checks on several arguments, with constraint violation handling and error indication via return value (standard *qsort* has no return value) and via *errno* (this implementation of *qsort* and *quickselect* also sets *errno* for invalid arguments). The additional tests include the ability to detect huge values for the number of array elements *nmemb* or the element *size*, such as might result from underflow of unsigned integer arithmetic. However, there are some shortcomings in the defined error handling: the sole interface for obtaining the address of the error handling function (e.g. in order to be able to call the handler) is via *set_constraint_handler_s*, which changes the handler function (and returns a pointer to the previous handler). C11 does not indicate whether the handler is per-thread, per-process, or global, and there is no mechanism provided to ensure exclusive access. While a second call to *set_constraint_handler_s* could in theory return the handler to its previous value, it is also possible that another thread or process may have made a change to the handler in the meantime, causing a race condition and resulting in unpredictable behavior.

2. An additional *void ** argument to the sort function which is passed unchanged as an additional argument to the comparison function. This has extensive implications: all internal functions which use comparisons (pivot selection, partitioning, dedicated sorting and selection, etc.) are affected. This appears to be a supposed alternative to using application-specific comparison functions; it does not seem to justify essentially doubling the library code size.

These variants will not be further discussed in detail; note that there are two additional public entry points and that much of the internal code has variations in order to accommodate these variants. Also note that the somewhat unusual code layout is primarily due to the desire to avoid code repetition resulting from the existence of the C11 variants.

2 Implementation

Qsort and *quickselect* look similar from the caller's point of view (Figure 1). The caller provides an array of data elements *base* containing *nmemb* elements each of which has size *size*, and a comparison function *compar*, and calls *qsort* which uses the comparison function to compare data elements and reorders the array in sorted order as determined by the comparison function. In addition, *quickselect* makes use of an array of *nk size_t* order statistic ranks at *pk* provided by the caller, and allows the caller to specify *options* which affect how *quickselect* operates.

Internally (Figure 2), *quickselect* and its qsort implementation are also similar; data size and alignment are used to select an efficient swapping function, *quickselect* may arrange for indirect sorting if that was a specified option, and operation is handed over to *quickselect_loop*. The *dedicated_sort* function is called for sorting (only); it determines the best sorting method for the data size, quantity, and the requested options. If *quickselect*'s divide-and-conquer method is deemed best, control is returned to *quickselect*, otherwise the data are sorted by the alternative method and *dedicated_sort* indicates that sorting is completed. For sorting or selection, the function *special_cases* is called; for selection it handles simple cases; for not-so-simple selection it determines the distribution of order statistic ranks in

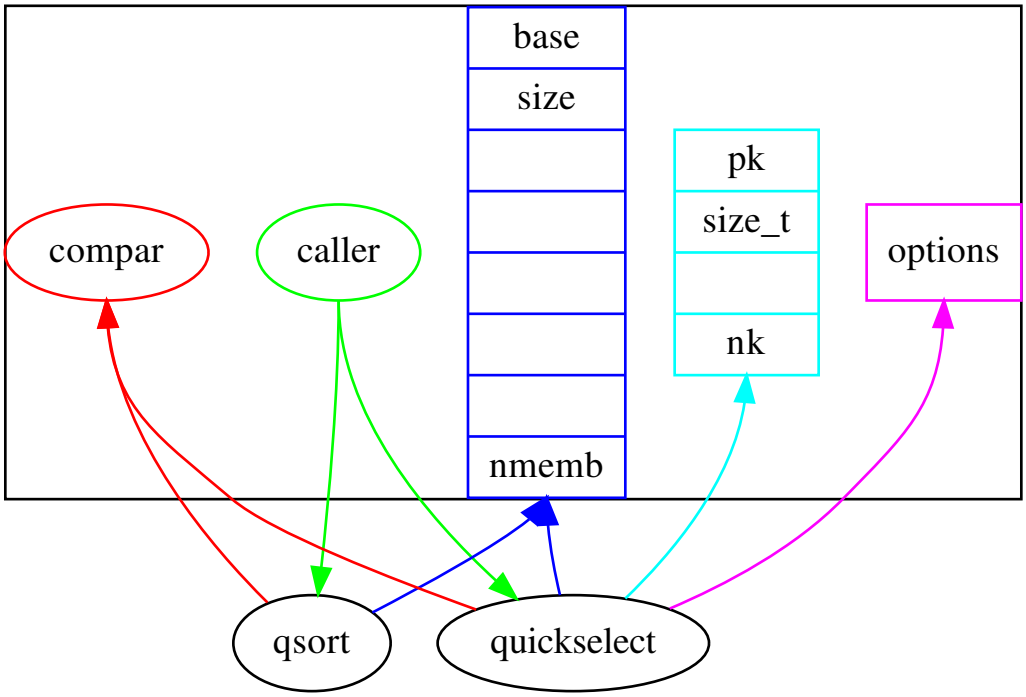

Figure 1: Caller's view of qsort and *quickselect*

three bands. A pivot element for partitioning is chosen by *select_pivot*, and that pivot is used to *partition* the array. Several methods of pivot selection and partitioning are available, depending on the data and specified options. For selection, *rank_tests* determines which of the partitioned regions requires further processing; this step is skipped for sorting. If both regions require processing, the smaller region is processed by recursively calling *quickselect_loop*. If the large region requires processing, its size is compared to the number of other elements to determine whether a more robust pivot selection method should be used by the next iteration; this is determined by *should_repivot*. At the far right, the data array is examined by calls to the caller-supplied comparison function *compar*, and data elements are arranged in sorted order by calling the appropriate swap function or by other types of data movement (e.g. rotations). In the case of indirection, pointers to the data are dereferenced, which presumably also takes place in the comparison function.

Referring to Figure 3 (which – in the interest of thoroughness – looks more complicated than it is):

- A caller provides a comparison function (green node labeled "compar") and calls either public interface: *qsort* (cyan node in library object file qsort.o) or *quickselect* (cyan node in library object quickselect.o). The caller of *quickselect* can specify processing options, for example to specify minimization of comparisons. Options are represented as bits in an unsigned integer; *quickselect_options* returns an unsigned integer containing all available bits set (some options might have been made unavailable during compilation). If the caller specifies bits which are unavailable, or in the event that the caller has supplied invalid arguments (e.g. *compar* is **NULL**), *quickselect* returns with errno set to **EINVAL**. Argument validity checks are also performed by the *qsort* implementations using this code base.

- In the case of *quickselect*, the caller has also provided an array of *size_t* order statistic ranks and the number of ranks, also as a *size_t* argument. Internal operation of *quickselect* requires that the ranks be unique and sorted. A pass is made over the ranks, and if an adjacent pair is found not to be in non-decreasing order, the array of ranks is sorted (using the internal sorting interface, which is a simplification of selection, as will be explained later). Comparison of the *size_t* ranks while checking for non-decreasing order is done directly; sorting uses a comparison function, *size_tcmp*, which is declared in header file compare.h and defined in quickselect_src.h. The comparison function *size_tcmp* is compiled as part of quickselect.o and is externally visible. The array of ranks is then checked for uniqueness; and duplicates are rotated to the end of the array. Only unique ranks are actually used for selection; the full set of ranks is returned to sorted order after selection. Rotation is implemented by element (*size_t*) swaps using an appropriate swapping function. The size and alignment of the *size_t* ranks array is determined using function *alignment_size*, which is defined in header file exchange.h and is compiled into quickselect.o (and also into qsort.o). The same function is used in both object files for determining the size and alignment of the array elements to be sorted or selected. The aligned size is passed to function *swapn*, which returns a pointer to a swap function which swaps in suitable increments of *char*; one of *swap1*, *swap2*, *swap4*, or *swap8*, all of which are also defined in exchange.h and are compiled into the two (four overall including *_s*

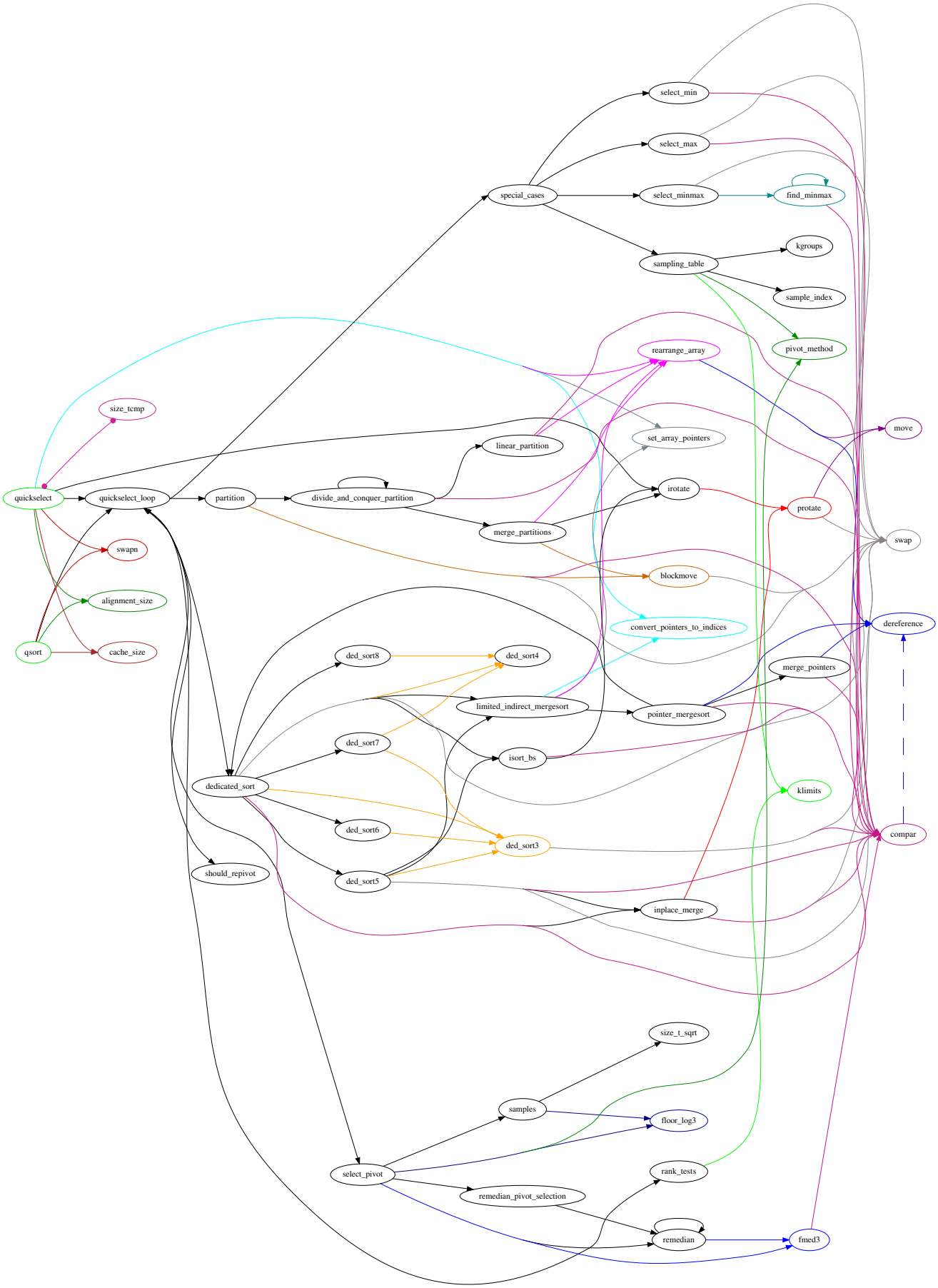

Figure 2: Library code control flow

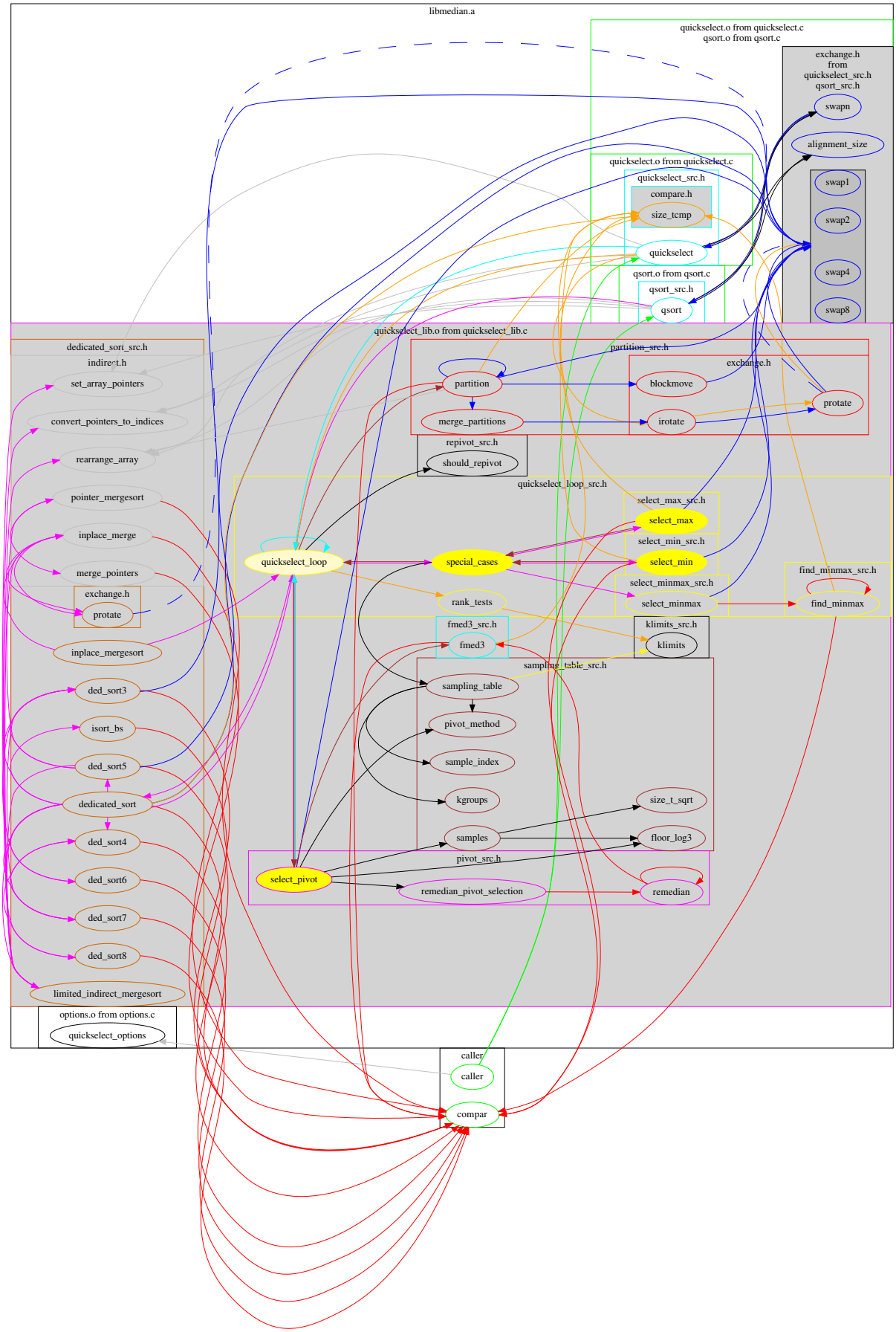

Figure 3: Library code implementation diagram

variants) object files. A pointer to the swap function is passed to support functions which perform swaps, just as the pointer to the caller's comparison function is passed to support functions which perform comparisons.

- As is performed for the *quickselect size_t* ranks array, *quickselect* and *qsort* use *alignment_size* and *swapn* to determine a suitable swapping function for the *base* array elements.

- The dedicated sorting function used for very small sub-arrays uses fewer comparisons and sometimes fewer swaps than divide-and-conquer quicksort. The sorting method used depends on the ratio of element size to the basic type size used for swapping; it is also to some extent dependent on the relative CPU performance of comparisons and data transfers. The size ratio is important because it affects the total run-time for swapping (but not for comparisons). That, in turn, affects the relative performance of the dedicated sort vs. divide-and-conquer sorting. The sorting method is also dependent on processing options; stable sorting cannot use optimal sorting networks.

- The ratio of element size to the alignment size used for swapping affects the relative cost of comparisons and swaps because the comparison cost is (mostly) independent of size whereas the cost of swapping is directly dependent on the size ratio; the size ratio determines the number of basic type moves needed to move one array element. Because the cost of moving elements is related to the size ratio, arrays of large elements might be more efficiently sorted indirectly by accessing the elements via pointers for comparisons, swapping pointers instead of elements (pointers are a basic type, so can be moved with a single instruction). The caller of *quickselect* can specify indirect sorting using the supplied *compar* function with internal allocation of pointers and indirection for comparisons. The caller could of course also allocate and initialize pointers (and free the allocated space after sorting), providing a *compar* function which accesses data element indirectly via pointers for comparisons. Such a caller-supplied method must be used for indirect sorting with *qsort*. Internal indirection in *quickselect* is able to cache frequently used pointers (e.g. the pivot element during partitioning), and relieves the caller of the responsibilities of pointer allocation, initialization, dereferencing, deallocation, and error (e.g. **ENOMEM**) handling. Function *set_array_pointers* allocates pointers and sets the pointers to point to array elements. After the pointers have been rearranged by sorting or selection to indicate the desired array order, pointers are converted to array indices using function *convert_pointers_to_indices* and function *rearrange_array* is called to move array elements to the desired positions. After the move, the pointers are freed. Internal functions (pivot selection and partitioning) which use comparisons dereference the pointers (caching the pivot element pointer) before calling the caller-supplied comparison function (which expects pointers to elements rather than pointers to pointers to elements). There is of course an $O(N)$ additional space requirement for the pointers, and dereferencing pointers before calling the comparison function adds some overhead. The performance improvement for indirect sorting of large data elements can be substantial; sorting involves $O(N \log N)$ moves, each of which requires r basic type moves for a ratio r of element size to basic type size. With indirect sorting, the pointers are moved, eliminating the factor of r (but adding $O(NlogN)$ pointer dereferences; the savings here depends on the relative costs of pointer dereference and data movement – pointer dereference is usually considerably less expensive than data movement). Rearranging the array elements with *rearrange_array* uses $O(N)$ data moves with the ratio r instead of $O(N \log N)$ such moves. For big-O constant k, direct sorting would use $krN \log N$ moves, whereas indirect sorting uses $kN \log N + rN$ moves, the savings being $N((r-1)k \log N - r)$ moves. For $k \approx 1$ and large r ($r - 1 \approx r$), the savings is approximately $r/2 N \log N$ moves. However, unlike direct sorting, rearrangement of elements via *rearrange_array* has poor locality of access for random data, and cache effects (described below) may affect performance.

- If stable sorting or selection was requested, *qsort* and *quickselect* try to allocate space for move-efficient (linear complexity) stable partitioning, which requires space proportional to the number of array elements for some temporary variables. If allocation fails, an in-place stable partitioning method having complexity $O(N \log N)$ is used in conjunction with a limited-size (based on data cache size, if that can be determined from the OS) linear-complexity stable partitioning function which uses stack space for the temporary variables.

- The public functions *qsort* and *quickselect* call *quickselect_loop* in library object file quickselect_lib.o. Once processing has entered library code in quickselect_lib.o, it continues from there except for calls to comparison and swapping functions and functions supporting indirection until the sorting or selection is complete.

- In each loop iteration when sorting, *quickselect_loop* calls *dedicated_sort*. The *dedicated_sort* function determines the most appropriate sorting method for the specified options, the number of elements in the sub-array, element size, and the ratio of element size to the basic type size used for swapping. If divide-and-conquer is the best sorting method, *dedicated_sort* returns **EAGAIN**, indicating that *quickselect_loop* should continue processing. For other methods, *dedicated_sort* completes sorting of the sub-array passed to it using the caller's comparison function and returns zero (or an error return value in the unlikely case that some supplied argument is invalid).

- Selection of minimum, maximum, or both the smallest and largest array elements can be performed more efficiently than by divide-and-conquer. Function *special_cases* checks for those specific selection cases and calls

the applicable functions to handle them. For selection other than those special cases, *special_cases* determines the distribution of order statistic ranks and the number of groups of ranks to determine if selection can be more efficiently performed by sorting. In some cases, depending on the number of array elements and the number and distribution of order statistic ranks, sorting might be more efficient than divide-and-conquer selection. One consideration is the number of groups of consecutive order statistic ranks, counted by function *kgroups*. The number of comparisons required for selection is roughly proportional to the number of elements times the logarithm of four times the number of groups, whereas the number of comparisons for a full sort is roughly proportional to the number of elements times the logarithm of the number of elements $O(N \log_2 N)$. Therefore, if the number of groups of consecutive order statistic ranks is one fourth of the number of elements or greater, sorting is more efficient than selection. Another consideration is the distribution of order statistics. The distribution of desired order statistics is determined by analyzing the number of order statistics in three bands; function *klimits* indicates how many ranks are in each band.

- When special-case selection does not apply, divide-and-conquer multiple selection (similar to quicksort) is used. The first step is to select a sample of the array; the number of samples is determined from tables based on the number of elements, the pivot selection method to be used, and on the distribution of desired order statistic ranks. Function *pivot_method* determines the pivot selection method based on the number of elements, the ratio of element size to the size of the basic type to be used for data movement, and the requested processing options. The number of samples to be used for pivot selection is returned by function *samples* which may consult tables of breakpoints for some methods, and may compute the number of samples using functions *size_t_sqrt* or *floor_log3*. The relevant tables are in file sampling_tables.c.

- Having determined the appropriate method and the number of samples, function *select_pivot* applies the method to the samples to obtain a pointer to the pivot element. One of four pivot selection methods is used:

  1. Remedian with base 3 of the sample of elements using internal function *remedian_pivot_selection*, which calls *remedian* which is recursive and which also calls internal function *fmed3*. The ternary median of three elements is found by *fmed3* using a minimal number of calls to the comparison function.
  2. The median or other order statistic of a set of samples may also be used to obtain a fast estimate for the pivot. For sorting, the median is used to obtain an estimate of the array median for a good partition. When selecting order statistics from the base array, if the desired ranks are all towards one end of the array, a pivot closer than the median to that end of the array may result in greater reduction of problem size. In median of samples (whether the median or other rank is selected), samples are swapped to the middle of the array for selection of the desired pivot using *quickselect_loop*.
  3. Median-of-medians is used after a particularly lopsided partition; *select_pivot* repeatedly calls *fmed3* to obtain medians of sets of three elements. The medians are swapped (using the appropriate swapping function) to the middle position of the set (minimizing the number of swaps using bias in the ternary median-of-3). The median of the medians (now located in the middle third of the array) is found by selection of the median using *quickselect_loop*.
  4. Remedian with base 3 of a large subset (at least 1/3) of elements using internal function *remedian* which is recursive and which also calls internal function *fmed3*. The ternary median of three elements is found by *fmed3* using a minimal number of calls to the comparison function. This variant of remedian is used when repivoting is required and partial order stability is also specified. Median-of-medians cannot be used in that case because it rearranges elements, disrupting partial order stability.

- The pivot element and the range of elements comparing equal to the pivot and bracketing a partition around the pivot (only the pivot element itself unless median of samples or median-of-medians was used) is passed to *partition* to partition the array elements around the pivot element. The *partition* function uses the bracketing information (if available) to avoid recomparisons. One of three partitioning methods is used:

  1. Non-stable split-end partitioning. Comparison and swapping functions are used, as is *blockmove* for moving a minimal number of elements to effectively reorder regions with similar ordering with respect to the pivot element. Partitioning by this method uses $N - 1$ comparisons for a sub-array of N elements and performs $O(N)$ moves.
  2. If stable sorting or selection has been specified and if sufficient space for temporary variables has been allocated, a linear-complexity stable partitioning method is used. A single scan of the sub-array uses $N - 1$ comparisons to classify each element according to its comparison result against the pivot element, with a ternary value according to less-than, equal-to, and greater-than comparison results. A set of indices are assigned for each sub-array position indicating the sub-array element original index which should occupy the partitioned sub-array position, effectively computing the permutation of the original sub-array which corresponds to the stable partitioning of that sub-array around the pivot element. The indices for less-than

element positions are assigned during the initial pass which determines comparison results. Indices for greater-than element positions are assigned during a linear pass over the comparison classification results, and equal-to element positions are assigned indices in an additional partial scan of the comparison results. Finally, one pass over the array of indices follows cycles in that permutation, moving each element to its correct partitioned location exactly once; the total number of element moves is $O(N)$. However, for random data, the locality of access involved in following the permutation cycles is poor, and consequently for large sub-arrays (where "large" depends on the size of elements, the number of elements, and the size of machine data caches) there may be cache-related performance penalties.

3. A stable in-place divide-and-conquer implementation which recursively partitions unpartitioned regions of the sub-array and then merges partitions using rotations. The number of comparisons is also $N - 1$, but because of divide-and-conquer and rotations, the movement complexity is $O(N \log N)$, which leads to $O(N \log N)$ stable selection and $O(N(logN)^2)$ stable sorting. To compensate for the higher complexity while avoiding cache misses, a limited-size version of the linear-complexity method is used when divide-and-conquer has reduced the sub-array size to the point where the elements and temporary variables will fit in the machine's data cache.

- After partitioning, the sizes of the regions resulting from the partition are examined. For selection only, a region can be removed from further processing if there are no desired order statistic ranks in that region. Function *klimits* makes this determination using an efficient binary search through the (sorted and unique) order statistic ranks. The selection-specific code including the call to *klimits* is encapsulated in inline function *rank_tests*. For sorting or selection, a region with only one element is necessarily in its correct position, and need not be processed. Elements comparing equal to the pivot are also in position and not in need of further processing. For selection, a region containing no desired order statistic ranks requires no processing. The smaller of the two regions of elements not comparing equal to the pivot element, if it requires processing at all, is processed by a recursive call to *quickselect_loop* unless the large region requires no processing, in which case the small region is processed iteratively.

- If the larger region requires processing, the size of the larger region is compared to the original number of elements to determine whether that large region should be repivoted by using median-of-medians (or remedian of elements if partial order stability is required) for the next pivot selection instead of remedian of samples or median of samples. Function *should_repivot* determines this from the region sizes, tables of repivoting factors, and a count maintained of the number of times the ratio of region sizes has exceeded a threshold. The large region is iteratively processed in *quickselect_loop*'s main loop. The repivot factor tables are in file repivot_tables.c, compiled into library object file repivot_tables.o.

- Any temporary space allocated for internal indirection or for stable sorting/selection is freed.

3 Source code layout

Meeting a number of goals resulted in a somewhat unusual source code layout. These goals include:

- Support for the C11 variants without repetition of common code. This is partially achieved by macros which provide for the variation of function return type and arguments.

- Modularity of source code; separate functions in separate files, avoiding huge files.

- Avoiding unnecessary repetition of code. Some code, implemented as separate functions, many of which may be expanded inline, is used by multiple callers (e.g. *klimits*, *fmed3*, and basic rotation and swapping functions). Other code is most readily implemented recursively (e.g. *find_minmax*, *remedian*, the mergesort implementation in *dedicated_sort*, and the in-place stable *partition* variant). Both considerations apply to *dedicated_sort*, which is called from multiple callers.

- The ability to build for various tradeoffs between object code size and run-time performance. Use of related inline functions linked together provides fast execution but large object file size. Conversely, individual functions can be called as needed to avoid multiple copies of code, but function call overhead limits run-time performance. The layout described in the previous section tends to favor run-time performance, but it was desired to arrange the source code so that a library could be built for small object code size without entailing enormous effort.

- Avoiding errors due to inconsistencies, e.g. between function declarations and definitions.

The following general file layout was used to achieve those goals:

- Configuration parameters placed in a single header file, quickselect_config.h.

- Declarations, including argument lists, for externally-visible functions defined as macros. The macro can be instantiated and followed by a semicolon to act as a declaration (possibly prefixed with qualifiers such as *extern*). The same macro can be instantiated and followed by the function definition in curly braces for the function implementation. The same source ensures consistency at some small inconvenience (the header file may need to be examined to see the parameter list).

- Being able to choose between separate object file creation or combined inline linkage is made possible by placing the source code in header files, which can the be #included separately (for separate object files) or in combination (for inline linkage) in a source file which consists predominantly of such #include directives.

- Use of feature test macros in combination with the inclusion of source code as header files permits building the C11 variants from a common code base, ensuring that bug fixes, performance improvements, feature enhancements, etc. are applied to all variants.

- Individual files can be kept to a reasonable size while still permitting combination into a single object file by inclusion of multiple source files (as header files) when building.

The source code file layout is detailed in Table 1 and can also be seen in Figure 3.

4 Data tables

Several aspects of program operation are controlled by tabular data:

- Breakpoints for increasing the number of samples used for pivot selection are maintained in 5 tables: two for sorting (one each for remedian of samples and for median of samples pivot selection) and three for selection (one for median of samples, and two for remedian of samples depending on the distribution of the desired order statistic ranks).

- Tables of thresholds compared to the ratio of the number of elements in the large region resulting from a partition to the number of remaining elements from the original sub-array which was partitioned are used to determine when to repivot the large region resulting from partitioning. Two tables are used for selection and two for sorting, one each depending on whether median of samples or remedian of samples (each of which has a different limit on the rank of the pivot) was used to select the pivot which resulted in the lopsided partition.

5 Building details

The order of object files in a library archive is important. Sorting of disordered order statistic ranks uses standard sorting via the internal *quickselect_loop* function (no extra "context" is necessary or desirable, and arguments once checked need not be checked again when passing unchanged to another function). Consequently, the object files for standard sorting should appear after the C11 variants, as those variants may need to call the standard functions. Object files containing the data tables appear last; the tables are used by all variants.

6 Trimming code for specific applications

For some applications, it may be desirable to reduce object code by featurecide; for example by eliminating the ability to perform stable sorts. Stable sorting code can be excluded by compiling libmedian with **QUICKSELECT_STABLE** defined as 0.

If order statistic selection is not required, a sorting-only version may be built with reduced code size. This will require modification by hand. The order statistics array and count may be eliminated, as well as checks for order statistics, special-case selection. Note, however, that selection is used internally to select (as pivot element) the median of medians to prevent sorting performance degradation from linearithmic to polynomial complexity for adverse inputs. In the normal build, *quickselect_loop* and *select_pivot* (with median-of-samples or median-of-medians pivot selection method) are co-functions: *quickselect_loop* calls *select_pivot* to select a pivot element, and *select_pivot* calls *quickselect_loop* to select the median of samples or medians.

If order selection will only be used to find the median, logic for recursion during selection can be eliminated (the median (or medians in the case of an even number of elements) will only appear in at most one region, which can be processed iteratively). However, when finding the median, the median is likely to be near one end of the region remaining after the first partition. Using median of samples to select for a pivot nearer to that end than the median reduces the asymptotic cost of finding the median from $2N$ comparisons and $0.5N$ swaps (when only the median is used for a pivot) to $1.5N$ comparisons and $0.25N$ swaps.

function	declaration	source	inclusion	object
alignment_size	exchange.h	exchange.h	qsort_src.h quickselect_src.h	(inline)
blockmove	exchange.h	exchange.h	partition_src.h	(inline)
convert_pointers_to_indices	indirect.h	indirect.h	dedicated_sort_src.h quickselect_src.h	(inline)
ded_sort**N**	(static)	dedicated_sort_src.h	dedicated_sort.c	(inline)
dedicated_sort	quickselect.h	dedicated_sort_src.h	dedicated_sort.c quickselect_loop_src.h	(inline) dedicated_sort.o
dedicated_sort_s	quickselect.h	dedicated_sort_src.h	dedicated_sort_s.c quickselect_loop_src.h	(inline) dedicated_sort_s.o
limited_indirect_mergesort	(static)	dedicated_sort_src.h	dedicated_sort.c	(inline)
find_minmax	quickselect.h	find_minmax_src.h	find_minmax.c quickselect_loop_src.c	(inline) find_minmax.o
find_minmax_s	quickselect.h	find_minmax_src.h	find_minmax_s.c quickselect_loop_src.c	(inline) find_minmax_s.o
floor_log3	quickselect.h	sampling_table_src.h	quickselect_lib.c sampling_table.o	(inline) sampling_table.o
fmed3	quickselect.h	fmed3_src.h	pivot_src.h	(inline) fmed3.o
fmed3_s	quickselect.h	fmed3_src.h	pivot_src.h	(inline) fmed3_s.o
inplace_merge	quickselect_config.h	indirect.h	dedicated_sort_src.h	(inline)
irotate	exchange.h	exchange.h	partition_src.h quickselect_src.h	(inline)
isort_bs	(static)	insertion_sort_src.h	dedicated_sort.c	(inline)
kgroups	(inline)	sampling_table_src.h	quickselect_lib.c quickselect_lib_s.c sampling_table.c	(inline) sampling_table.o
klimits	quickselect.h	klimits_src.h	quickselect_loop_src.h sampling_table_src.h quickselect_lib.c quickselect_lib_s.c	(inline) klimits.o
merge_partitions	(static)	partition_src.h	quickselect_lib.c	(inline)
merge_pointers	(inline)	indirect.h	dedicated_sort_src.h	(inline)
partition	quickselect.h	partition_src.h	partition.c quickselect_lib.c	partition.o
partition_s	quickselect.h	partition_src.h	partition_s.c quickselect_lib_s.c	partition.o
pivot_method	quickselect.h	sampling_table_src.h	pivot_src.h quickselect_lib.c quickselect_lib_s.c sampling_table.c	(inline) sampling_table.o
pointer_mergesort	(inline)	indirect.h	dedicated_sort_src.h indirect_mergesort.c	(inline) indirect_mergesort.o
protate	exchange.h	exchange.h	select_max_src.h select_min_src.h select_minmax_src.h	(inline)
qsort	quickselect.h (stdlib.h)	qsort_src.h	qsort.c	qsort.o
qsort_s	quickselect.h (stdlib.h)	qsort_src.h	qsort_s.c	qsort_s.o
quickselect	quickselect.h	quickselect_src.h	quickselect.c	quickselect.o
quickselect_loop	quickselect.h	quickselect_loop_src.h	quickselect_lib.c	quickselect_lib.o
quickselect_loop_s	quickselect.h	quickselect_loop_src.h	quickselect_lib_s.c	quickselect_lib_s.o
quickselect_s	quickselect.h	quickselect_src.h	quickselect_s.c	quickselect_s.o
rank_tests	(static)	quickselect_loop_src.h	quickselect_lib.c quickselect_lib_s.c	(inline)
rearrange_array	(static)	indirect.h	dedicated_sort_src.h partition_src.h quickselect_src.h	(inline)
remedian	(static)	pivot_src.h	quickselect_lib.c	(inline)
remedian_s	(static)	pivot_src.h	quickselect_lib_s.c	(inline)
remedian_pivot_selection	(static)	repivot_src.h	quickselect_lib.c	(inline)
samples	quickselect.h	sampling_table_src.h	pivot_src.h quickselect_lib.c quickselect_lib_s.c sampling_table.c	(inline) sampling_table.o
sample_index	quickselect.h	sampling_table_src.h	quickselect_lib.c quickselect_lib_s.c	(inline) sampling_table.o
sampling_table	quickselect.h	sampling_table_src.h	quickselect_lib.c quickselect_lib_s.c sampling_table.c	(inline) sampling_table.o
select_max	quickselect.h	select_max_src.h	quickselect_loop_src.h select_max.c	(inline) select_max.o
select_max_s	quickselect.h	select_max_src.h	quickselect_loop_src.h select_max_s.c	(inline) select_max_s.o
select_min	quickselect.h	select_min_src.h	quickselect_loop_src.h select_min.c	(inline) select_min.o
select_min_s	quickselect.h	select_min_src.h	quickselect_loop_src.h select_min_s.c	(inline) select_min_s.o
select_minmax	quickselect.h	select_minmax_src.h	quickselect_loop_src.h select_minmax.c	(inline) select_minmax.o
select_minmax_s	quickselect.h	select_minmax_src.h	quickselect_loop_src.h select_minmax_s.c	(inline) select_minmax_s.o
select_pivot	quickselect.h	pivot_src.h	quickselect_loop_src.c pivot.c	(inline) pivot.o
select_pivot_s	quickselect.h	pivot_src.h	quickselect_loop_src.c pivot_s.c	(inline) pivot_s.o
set_array_pointers	indirect.h	indirect.h	dedicated_sort_src.h quickselect_src.h	(inline)
should_repivot	quickselect.h	repivot_src.h	quickselect_loop_src.h should_repivot.c	(inline) should_repivot.o
size_tcmp	compare.h	quickselect_src.h	quickselect_src.h	(inline) quickselect.o
size_t_sqrt	quickselect.h	sampling_table_src.h	pivot_src.h	(inline) sampling_table.o
special_cases	(static)	quickselect_loop_src.h	quickselect_lib.c	(inline)
swap1 swap2 swap4 swap8	exchange.h	exchange.h	qsort_src.h quickselect_src.h	(inline)
swapn	exchange.h	exchange.h	qsort_src.h quickselect_src.h	(inline)

Table 1: Source code layout

Prevention of performance degradation with adverse inputs is inextricably linked with selection in this implementation; completely giving up selection means giving up protection against performance degradation. Limiting selection to median only will only slightly reduce code size. If order statistic selection is eliminated or used exclusively for median(s) selection, some tables can be eliminated, but that also provides only a small reduction in size.

The C11 *_s* variants can be eliminated if they are not desired; doing so will considerably reduce the library size. The easiest way to do so is to simply use **ar** to delete the relevant object files (ending in *_s.o*). If the library is likely to have to be rebuilt, the specification for those object files may be removed or commented out in the *makefile.files*

table	access functions	source	object
ends_sampling_table	sampling_table	sampling_tables.c	sampling_tables.o
middle_sampling_table	sampling_table	sampling_tables.c	sampling_tables.o
mos_middle_sampling_table	sampling_table	sampling_tables.c	sampling_tables.o
mos_selection_repivot_table	sampling_table	repivot_tables.c	repivot_tables.o
mos_sorting_repivot_table	sampling_table	repivot_tables.c	repivot_tables.o
mos_sorting_sampling_table	sampling_table	sampling_tables.c	sampling_tables.o
ros_selection_repivot_table	sampling_table	repivot_tables.c	repivot_tables.o
ros_sorting_repivot_table	sampling_table	repivot_tables.c	repivot_tables.o
sorting_sampling_table	sampling_table	sampling_tables.c	sampling_tables.o

Table 2: Tables

file to prevent their regeneration.

Library *libmedian* includes functionality beyond *qsort* and *quickselect*; there are object files which implement indirect mergesort and repeated median filtering. These may also be removed if they are not required.

7 Performance tuning

Table values can be adjusted to tune performance for particular machine architectures or for specific applications. The testing framework (described in the next section) may be useful for examining application-specific or architecture-specific performance details.

8 Building the library and testing framework

A testing framework consists of main program code residing in src/median_test.c and library code in lib/libmedian_test/src files. Many of the source files provide implementations of various sorting algorithms which have been analyzed and compared with *quickselect*. The code used in *quickselect* is largely duplicated in the testing framework (with some additional debugging/analysis functionality; however some differences in implementation may have crept in) but the tables used are the same. The code execution can be varied somewhat without recompilation by varying command-line options which affect parameters that would normally be determined at compile-time or based on data characteristics. Library code for *quickselect* has also been optimized somewhat to improve run-time performance and reduce code size, whereas the code in the testing framework emphasizes analysis and debugging features over performance and code size. An ongoing effort is underway to eliminate implementation differences by using a common code base with conditional compilation of analysis and debugging features.

The testing framework can be used as-is for testing *quickselect* and its *qsort* implementation with simple data types (*short*, plain and *long* integers, double-precision floating point numbers) and fairly small (50 bytes) data structures which include character strings and structured data with a few small integer components. Various structured and random input sequences may be generated, and there is provision for reading text representations of long integer input sequences from user-supplied files (long integer data is converted to floating-point, smaller integer, and structured data (including strings) to support those data types). Statistics regarding number of comparisons, swaps, rotations, merges, levels of recursion, etc. and run-time performance are generated.

Machine architectural information (data type sizes and alignment, timing of some basic operations applied to basic data types) can be generated by the testing framework. Some care may be required in interpretation of operation timing information vs. compiler optimization, and further note that even modestly complex real code may exhibit instruction cache effects on some machines which do not affect simple loops of basic operations. Extended operation sets (e.g. incorporating SIMD (single instruction, multiple data) instructions) and compilers (and/or run-time libraries) which exploit such operations may perform differently in simple loops vs. production code.

Somewhat non-standard floating point representations (e.g. those which may be used on Solaris systems and its derivatives) may exhibit unusual performance characteristics. The library code of *quickselect* does not directly use floating-point variables or operations, but the testing framework does, and use of non-standard representation may exhibit some anomalies in statistics reported by the testing framework and in operations performed on floating-point data types.

The testing framework would have to be modified to incorporate application-specific data structures.

Building the testing framework also builds the *libmedian* library and a few other libraries used by the testing framework. The build process is handled by *make* supplemented by a few shell scripts. It is usually as simple as:

```
make depend

make
```

Details may be found in the file README.makefiles. Building with the default configuration is usually a good starting point. The header file quickselect_config.h may be modified to suit local needs, and the necessary library object files can be updated by running make again. In some cases, it may be necessary to run "make clean-depend" to regenerate dependency data before re-running make to build the libraries and executables. As described above, *makefile.files* may be modified to exclude some library code, and "make depend" will have generated a makefile with "overrides" in the file name, which can be used to customize the build for a particular machine (and/or compiler on that machine, and/or operating system if the machine is configured for multiple OSes). This is also described in README.makefiles.

References

[1] ISO/IEC JTC1/SC22/WG14 - C. ISO/IEC9899:201x. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, April, 2011. [23 September 2017].

List of Figures

List of Tables