Branch Mispredictions Don't Affect Mergesort[⋆]

Amr Elmasry[1], Jyrki Katajainen[1,2], and Max Stenmark[2]

[1] Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] Jyrki Katajainen and Company
Thorsgade 101, 2200 Copenhagen North, Denmark

Abstract. In quicksort, due to branch mispredictions, a skewed pivot-selection strategy can lead to a better performance than the exact-median pivot-selection strategy, even if the exact median is given for free. In this paper we investigate the effect of branch mispredictions on the behaviour of mergesort. By decoupling element comparisons from branches, we can avoid most negative effects caused by branch mispredictions. When sorting a sequence of n elements, our fastest version of mergesort performs $n \log_2 n + O(n)$ element comparisons and induces at most $O(n)$ branch mispredictions. We also describe an in-situ version of mergesort that provides the same bounds, but uses only $O(\log_2 n)$ words of extra memory. In our test computers, when sorting integer data, mergesort was the fastest sorting method, then came quicksort, and in-situ mergesort was the slowest of the three. We did a similar kind of decoupling for quicksort, but the transformation made it slower.

1 Introduction

Branch mispredictions may have a significant effect on the speed of programs. For example, Kaligosi and Sanders [8] showed that in quicksort [6] it may be more advantageous to select a skewed pivot instead of finding a pivot close to the median. The reason for this is that for a comparison against the median the outcome has a fifty percent chance of being smaller or larger, whereas the outcome of comparisons against a skewed pivot is easier to predict. All in all, a skewed pivot will lead to a better branch prediction and—possibly—a decrease in computation time. In a same vein, Brodal and Moruz [3] showed that skewed binary search trees can perform better than perfectly balanced search trees.

In this paper we tackle the following question posed in [8]. Given a random permutation of the integers $\{0, 1, \ldots, n-1\}$, does there exist a faster in-situ sorting algorithm than quicksort with skewed pivots for this particular type of input? We use the word *in-situ* to indicate that the algorithm is allowed to use $O(\log_2 n)$ extra words of memory (as any careful implementation of quicksort).

It is often claimed that quicksort is faster than mergesort. To check the correctness of this claim, we performed some simple benchmarks for the quicksort (`std::sort`) and mergesort (`std::stable_sort`) programs available at the GNU implementation (`g++` version 4.6.1) of the C++ standard library; `std::sort` is

[⋆] © 2012 Springer-Verlag. This is the authors' version of the work. The original publication is available at `www.springerlink.com`.

Table 1. The execution time [ns], the number of conditional branches, and the number of mispredictions on two of our computers (Per and Ares), each per $n \log_2 n$, for the quicksort and mergesort programs taken from the C++ standard library.

Program	`std::sort`				`std::stable_sort`			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	6.5	5.3	1.47	0.45	6.2	5.0	2.05	0.14
2^{15}	6.2	5.2	1.50	0.43	5.9	4.7	2.02	0.09
2^{20}	6.2	5.1	1.50	0.43	6.3	4.7	2.01	0.07
2^{25}	6.1	5.1	1.51	0.43	6.1	4.6	2.01	0.05

an implementation of Musser's introsort [13] and `std::stable_sort` is an implementation of bottom-up mergesort. In our test environment[3], for integer data, the two programs had the same speed within the measurement accuracy (see Table 1). An inspection of the assembly-language code produced by `g++` revealed that in the performance-critical inner loop of `std::stable_sort` all element comparisons were followed by conditional moves. A *conditional move* is written in C as **if** `(a` $\lhd$ `b) x = y`, where `a`, `b`, `x`, and `y` are some variables (or constants), and $\lhd$ is some comparison operator. This instruction, or some of its restricted forms, is supported as a hardware primitive by most computers. By using a branch-prediction profiler (`valgrind`) we could confirm that the number of branch mispredictions per $n \log_2 n$—referred to as the *branch-misprediction ratio*—was much lower for `std::stable_sort` than for `std:.sort`. Based on these initial observations, we concluded that mergesort is a noteworthy competitor to quicksort.

Our main results in this paper are:

1. We optimize (reduce the number of branches executed and branch mispredictions induced) the standard-library mergesort so that it becomes faster than quicksort for our task in hand (Section 2).
2. We describe an in-situ version of this optimized mergesort (Section 3). Even though an ideal translation of its inner loop only contains 18 assembly-language instructions, in our experiments it was slower than quicksort.

[3] The experiments discussed in the paper were carried out on two computers:
Per: Model: Intel® Core$^{\text{TM}}$2 CPU T5600 @ 1.83GHz; main memory: 1 GB; L2 cache: 8-way associative, 2 MB; cache line: 64 B.
Ares: Model: Intel® Core$^{\text{TM}}$ i3 CPU M 370 @ 2.4GHz × 4; main memory: 2.6 GB; L2 cache: 12-way associative, 3 MB; cache line: 64 B.
Both computers run under Ubuntu 11.10 (Linux kernel 3.0.0-15-generic) and `g++` compiler (`gcc` version 4.6.1) with optimization level `-O3` was used. According to the documentation, at optimization level `-O3` this compiler always attempted to transform conditional branches into branch-less equivalents. Micro-benchmarking showed that in Per conditional moves were faster than conditional branches when the result of the branch condition was unpredictable. In Ares the opposite was true. All execution times were measured using `gettimeofday` in `sys/time.h`. For a problem of size n, each experiment was repeated $2^{26}/n$ times and the average was reported.

3. We eliminated all branches from the performance-critical loop of quicksort (Section 4). After this transformation the program induces $O(n)$ branch mispredictions on the average. However, in our experiments the branch-optimized versions of quicksort were slower than `std::sort`.
4. We made a number of experiments for quicksort with skewed pivots (Section 4). We could repeat the findings reported in [8], but the performance improvement obtained by selecting a skewed pivot was not very large. For our mergesort programs the branch-misprediction ratio is significantly lower than that reported for quicksort with skewed pivots in [8].

We took the idea of decoupling element comparisons from branches from Mortensen [12]. He described a variant of mergesort that performs $n \log_2 n + O(n)$ element comparisons and induces $O(n)$ branch mispredictions. However, the performance-critical loop of the standard-library mergesort only contains 14 assembly-language instructions, whereas that of Mortensen's program contains more. This could be a reason why Mortensen's implementation is slower than the standard-library implementation. Our key improvement is to keep the instruction count down while doing the branch optimization.

The idea of decoupling element comparisons from branches was also used by Sanders and Winkel [15] in their samplesort. The resulting program performs $n \log_2 n + O(n)$ element comparisons and induces $O(n)$ branch mispredictions in the expected case. As for Mortensen's mergesort, samplesort needs $O(n)$ extra space. Using the technique described in [15], one can modify heapsort such that it will achieve the same bound on the number of branch mispredictions in addition to its normal characteristics. In particular, heapsort is fully in-place, but it suffers from a bad cache behaviour [5].

Brodal and Moruz [2] proved that any comparison-based program that sorts a sequence of n elements using $O(\beta n \log_2 n)$ element comparisons, for $\beta > 1$, must induce $\Omega(n \log_\beta n)$ branch mispredictions. However, this result only holds under the assumption that every element comparison is followed by a conditional branch depending on the outcome of the comparison. In particular, after decoupling element comparisons from branches the lower bound is no more valid.

The branch-prediction features of a number of sorting programs were experimentally studied in [1]; also a few optimizations were made to known methods. In a companion paper [5] it is shown that any program can be transformed into a form that induces only $O(1)$ branch mispredictions. The resulting programs are called *lean*. However, for a program of length κ, the transformation may make the lean counterpart a factor of κ slower. In practice, the slowdown is not that big, but the experiments showed that lean programs were often slower than moderately branch-optimized programs. In [5], lean versions of mergesort and heapsort are presented. In the current paper, we work towards speeding up mergesort even further and include quicksort in the study.

A reader who is unfamiliar with the branch-prediction techniques employed at the hardware level should recap the basic facts from a textbook on computer organization (e.g. [14]). In our theoretical analysis, we assume that the branch predictor used is static. A typical static predictor assumes that forward branches are not taken and backward branches are taken. Hence, for a conditional branch at the end of a loop the prediction is correct except for the last iteration when stepping out of the loop.

2 Tuned Mergesort

In the C++ standard library shipped with our compiler, `std::stable_sort` is an implementation of bottom-up mergesort. First, it divides the input into chunks of seven elements and sorts these chunks using insertionsort. Second, it merges the sequences sorted so far pairwise, level by level, starting with the sorted chunks, until the whole sequence is sorted. If possible, it allocates an extra buffer of size n, where n is the size of the input, then it alternatively moves the elements between the input and the buffer, and produces the final output in the place of the original input. If no extra memory is available, it reverts to an in-situ sorting strategy, which is asymptotically slower than the one using extra space.

One reason for the execution speed is a tight (compact) inner loop. We reproduce it in a polished form below on the left together with its assembly-language translation on the right. When illustrating the assembly-language translations, we use pure C [10], which is a glorified assembly language with the syntax of C [11]. In the following code extracts, `p` and `q` are iterators pointing to the current elements of the two input sequences, `r` is an iterator indicating the current output position, `t1` and `t2` are iterators indicating the first positions beyond the input sequences, and `less` is the comparator used in element comparisons. The additional variables are temporary: `s` and `t` store iterators, `x` and `y` elements, and `done` and `smaller` Boolean values.

```
while (p != t1 && q != t2) {
  if (less(*q, *p)) {
    *r = *q;
    ++q;
  }
  else {
    *r = *p;
    ++p;
  }
  ++r;
}
```

```
test:
  done = (q == t2);
  if (done) goto exit;
entrance:
  x = *p;
  s = p + 1;
  y = *q;
  t = q + 1;
  smaller = less(y, x);
  if (smaller) q = t;
  if (! smaller) p = s;
  if (! smaller) y = x;
  *r = y;
  ++r;
  done = (p == t1);
  if (! done) goto test;
exit:
```

Since the two branches of the **if** statement are so short and symmetric, a good compiler will compile them using conditional moves. The assembly-language translation corresponding to the pure-C code was produced by the `g++` compiler. As shown on the right above, the output contained 14 instructions.

By decoupling element comparisons from branches, each merging phase of two subsequences induces at most $O(1)$ branch mispredictions. In total, the merge procedure is invoked $O(n)$ times, so the number of branch mispredictions induced is $O(n)$. Other characteristics of bottom-up mergesort remain the same; it performs $n \log_2 n + O(n)$ element comparisons and element moves.

Table 2. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for bottom-up mergesort taken from the C++ standard library and our optimized mergesort.

Program	`std::stable_sort`				**Optimized mergesort**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	6.2	5.0	2.05	0.14	4.4	3.5	0.75	0.06
2^{15}	5.9	4.7	2.02	0.09	4.4	3.5	0.66	0.04
2^{20}	6.3	4.7	2.01	0.07	5.2	3.7	0.62	0.03
2^{25}	6.1	4.6	2.01	0.05	5.2	3.7	0.60	0.02

To reduce the number of branches executed and the number of branch mispredictions induced even further, we implemented the following optimizations:

– Handle small subproblems differently: Instead of using insertionsort, we sort each chunk of size four with a straight-line code that has no branches. In brief, we simulate a sorting network for four elements using conditional moves. Insertionsort induces one branch misprediction per element, whereas our routine only induces $O(1)$ branch mispredictions in total.
– Unfold the main loop responsible for merging: When merging two subsequences, we move four elements to the output sequence in each iteration. We do this as long as each of the two subsequences to be merged has at least four elements. Hereafter in this performance-critical loop the instructions involved in the conditional branches, testing whether or not one of the input subsequences is exhausted, are only executed every fourth element comparison. If one or both subsequences have less than four elements, we handle the remaining elements by a normal (not-unfolded) loop.

To see whether or not our improvements are effective in practice, we tested our optimized mergesort against `std::stable_sort`. Our results are reported in Table 2. From these results, it is clear that even improvements in the linear term can be significant for the efficiency of a sorting program.

3 Tuned In-Situ Mergesort

Since the results for mergesort were so good, we set ourselves a goal to show that some variation of the in-place mergesort algorithm of Katajainen et al. [9] will be faster than quicksort. We have to admit that this goal was too ambitious, but we came quite close. We should also point out that, similar to quicksort, the resulting sorting algorithm is no more stable.

The basic step used in [9] is to sort half of the elements using the other half as a working area. This idea could be utilized in different ways. We rely on the simplest approach: Before applying the basic step, partition the elements around the median. In principle, the standard-library routine `std::nth_element` can accomplish this task by performing a quicksort-type partitioning. After partitioning and sorting, the other half of the elements can be handled recursively. We stop the recursion when the number of remaining elements is less than $n/\log_2 n$ and use introsort to handle them. An iterative procedure-level description of this sorting program is given below. Its interface is the same as that for `std::sort`.

```
template <typename iterator, typename comparator>
void sort(iterator p, iterator r, comparator less) {
  typedef typename std::iterator_traits<iterator>::difference_type index;
  index n = r - p;
  index threshold = n / ilogb(2 + n);
  while (n > threshold) {
    iterator q_1 = p + n / 2;
    iterator q_2 = r - n / 2;
    converse_relation<comparator> greater(less);
    std::nth_element(p, q_1, r, greater);
    mergesort(p, q_1, q_2, less);
    r = q_1;
    n = r - p;
  }
  std::sort(p, r, less);
}
```

Most of the work is done in the basic steps, and each step only uses $O(1)$ extra space in addition to the input sequence. Compared to normal mergesort, the inner loop is not much longer. In the following code extracts, the variables have the same meaning as those used in tuned mergesort: `p`, `q`, `r`, `s`, `t`, `t1`, and `t2` store iterators; `x` and `y` elements; and `done` and `smaller` Boolean values.

```
while (p != t1 && q != t2) {
  if (less(*q, *p)) {
    s = q;
    ++q;
  }
  else {
    s = p;
    ++p;
  }
  x = *r;
  *r = *s;
  *s = x;
  ++r;
}
```

```
test:
  done = (q == t2);
  if (done) goto exit;
entrance:
  x = *p;
  s = p + 1;
  y = *q;
  t = q + 1;
  smaller = less(y, x);
  if (smaller) s = t;
  if (smaller) q = t;
  if (! smaller) p = s;
  if (! smaller) y = x;
  x = *r;
  *r = y;
  --s;
  *s = x;
  ++r;
  done = (p == t1);
  if (! done) goto test;
exit:
```

As shown on the right above, an ideal translation of the loop contains 18 assembly-language instructions, which is only four more than that required by the inner loop of mergesort. Because of register spilling, the actual code produced by the `g++` compiler was a bit longer; it contained 26 instructions. Again, the two branches of the **if** statement were compiled using conditional moves.

For an input of size m, the worst-case cost of `std::nth_element` and `std::sort` is $O(m)$ and $O(m \log_2 m)$, respectively [13]. Thus, the overhead caused by these subroutines is linear in the input size. Both of these routines require at most a logarithmic amount of extra space. To sum up, we rely on standard library components and ensure that our program only induces $O(n)$ branch mispredictions.

Table 3. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two in-situ variants of mergesort.

Program	**In-situ** `std::stable_sort`				**In-situ mergesort**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	49.2	29.7	9.0	2.08	7.3	5.7	1.93	0.26
2^{15}	57.6	35.0	11.1	2.38	7.1	5.6	1.94	0.15
2^{20}	62.7	38.5	12.9	2.53	7.4	5.7	1.92	0.11
2^{25}	68.0	41.3	14.4	2.62	7.6	5.7	1.92	0.09

In our experiments, we compared our in-situ mergesort against the space-economical mergesort provided by the C++ standard library. The library routine is recursive, so (due to the recursion stack) it requires a logarithmic amount of extra space. The performance difference between the two programs is stunning, as seen in Table 3. We admit that this comparison is unfair; the library routine promises to sort the elements stably, whereas our in-situ mergesort does not. However, this comparison shows how well our in-situ mergesort performs.

4 Comparison to Quicksort

In the C++ standard library shipped with our compiler, `std::sort` is an implementation of introsort [13], which is a variant of median-of-three quicksort [6]. Introsort is half-recursive, it coarsens the base case by leaving small subproblems (of size 16 or smaller) unsorted, it calls insertionsort to finalize the sorting process, and it calls heapsort if the recursion depth becomes too large. Since introsort is known to be fast, it was natural to use it as our starting point.

The performance-critical loop of quicksort is tight as shown on the left below; `p` and `r` are iterators indicating how far the partitioning process has proceeded from the beginning and the end, respectively; `v` is the pivot, and `less` is the comparator used in element comparisons; the four additional variables are temporary: `x` and `y` store elements, and `smaller` and `cross` Boolean values.

```
while (true) {
  while (less(*p, v)) {
    ++p;
  }
  --r;
  while (less(v, *r)) {
    --r;
  }
  if (p >= r) {
    return p;
  }
  x = *p;
  *p = *r;
  *r = x;
  ++p;
}
```

```
  --p;
  goto first_loop;
swap:
  *p = y;
  *r = x;
first_loop:
  ++p;
  x = *p;
  smaller = less(x, v);
  if (smaller) goto first_loop;
second_loop:
  --r;
  y = *r;
  smaller = less(v, y);
  if (smaller) goto second_loop;
  cross = (p < r);
  if (cross) goto swap;
  return p;
```

In the assembly-language translation displayed in pure C [10] on the right above, both of the innermost **while** loops contain four instructions and, after rotating the instructions of the outer loop such that the conditional branch becomes its last instruction, the outer loop contains four instructions as well. Due to instruction scheduling and register allocation, the picture was a bit more complicated for the code produced by the compiler, but the simplified code displayed to the right above is good enough for our purposes.

For the basic version of mergesort, the number of instructions executed per $n \log_2 n$, called the *instruction-execution ratio*, is 14. Let us now analyse this ratio for quicksort. It is known that for the basic version of quicksort the expected number of element comparisons performed is about $2n \ln n \approx 1.39 n \log_2 n$ and the expected number of element exchanges is one sixth of this number [16]. Combining this with the number of instructions executed in different parts of the performance-critical loop, the expected instruction-execution ratio is

$$(4 + (1/6) \times 4) \times 1.39 \approx 6.48\,.$$

This number is extremely low; even for our improved mergesort the instruction-execution ratio is higher (11 instructions).

The key issue is the conditional branches at the end of the innermost **while** loops; their outcome is unpredictable. The performance-critical loop can be made lean using the program transformation described in [5]. A bit more efficient code is obtained by numbering the code blocks and executing the moves inside the code blocks conditionally. We identify three code blocks in the performance-critical loop of Hoare's partitioning algorithm: the two innermost loops and the swap. By converting the **while** loops to **do-while** loops, we could avoid some code repetition. The outcome of the program transformation is given on the left below; variable `lambda` indicates the code block under execution. It turns out that the corresponding code is much shorter for Lomuto's partitioning algorithm described, for example, in [4]. Now the performance-critical loop only contains one **if** statement, and the swap inside it can be executed conditionally. The code obtained by applying the program transformation is shown on the right below.

```
int lambda = 2;
--p;
do {
  if (lambda == 1) *p = y;
  if (lambda == 1) *r = x;
  if (lambda == 1) lambda = 2;
  if (lambda == 2) ++p;
  x = *p;
  smaller = less(x, v);
  if (lambda != 2) smaller = true;
  if (! smaller) lambda = 3;
  if (lambda == 3) --r;
  y = *r;
  smaller = less(v, y);
  if (lambda != 3) smaller = true;
  if (! smaller) lambda = 1;
} while (p < r);
return p;
```

```
while (q < r) {
  x = *q;
  condition = less(x, v);
  if (condition) ++p;
  if (condition) *q = *p;
  if (condition) *p = x;
  ++q;
}
return p;
```

The resulting programs are interesting in several respects. When we rely on Hoare's partitioning algorithm, in each iteration of the loop two element compari-

Table 4. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two branch-optimized variants of quicksort.

Program	**Optimized quicksort (Hoare)**				**Optimized quicksort (Lomuto)**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	9.2	6.5	2.93	0.43	6.5	5.2	2.14	0.40
2^{15}	9.5	6.4	3.24	0.42	6.5	5.1	2.34	0.40
2^{20}	9.7	6.5	3.33	0.42	6.6	5.2	2.40	0.40
2^{25}	9.8	6.5	3.46	0.42	6.7	5.3	2.42	0.40

sons are performed. Since the loop is executed $\sim 2n \ln n$ times on the average, the expected number of element comparisons increases to $\sim 2.78 n \log_2 n$. This increase does not occur for Lomuto's partitioning algorithm. Note that in the above C code we allow conditional moves between memory locations, and even allow conditional arithmetic. If we are restricted to only use conditional moves (as in pure-C), these instructions need to be substituted by pure-C instructions. Because the underlying hardware only supports conditional moves to registers, the assembly-language instruction counts were a bit higher than that indicated by the C code above; the actual counts were 20 and 11, respectively. This means that the expected instruction-execution ratio (that is a 1.39 factor of the instruction counts) is around 27.8 when Hoare's partitioning is in use and around 15.29 when Lomuto's partitioning is in use. Thus, the cost of eliminating the unpredictable branches is high in both cases!

When testing these branch-optimized versions of quicksort, we observed that the compiler was not able to handle that many conditional moves. In some architectures each such move requires more than one clock cycle, so it may be more efficient to use conditional branches. The performance of our branch-optimized quicksort programs is reported in Table 4. Compared to introsort (see Table 1), these programs are slower. To avoid branch mispredictions, it would be necessary to write the programs in assembly language.

We also tested other variants of introsort by trying different pivot-selection strategies: random element, first element, median of the first, middle and last element, and α-skewed hypothetical element. Since in our setup the elements are given in random order, the simplest pivot-selection strategy—select the first element as the pivot—was already fast, but it was slower than the median-of-three pivot-selection strategy used by introsort. On the other hand, the selection of a skewed pivot indeed improved the performance. In our test environment, for small problem instances the median pivot worked best (i.e. $\alpha = 1/2$), whereas for large problem instances $\alpha = 1/5$ turned out to be the best choice. The results for the naive and α-skewed pivot-selection strategies are given in Table 5.

From these experiments, our conclusion is that the performance of the sorting programs considered is ranked as follows: mergesort, quicksort, and in-situ mergesort. Still, quicksort is the fastest method for in-situ sorting.

Table 5. The execution time [ns], the number of conditional branches, and the number of mispredictions, each per $n \log_2 n$, for two other variants of introsort.

Program	**Introsort (naive pivot)**				**Introsort ((1/5)-skewed pivot)**			
	Time		**Branches**	**Mispredicts**	**Time**		**Branches**	**Mispredicts**
n	**Per**	**Ares**			**Per**	**Ares**		
2^{10}	7.0	5.6	1.78	0.45	6.4	5.1	1.48	0.37
2^{15}	6.6	5.3	1.78	0.43	6.1	4.8	1.53	0.36
2^{20}	6.5	5.1	1.74	0.42	6.0	4.7	1.55	0.35
2^{25}	6.4	5.1	1.72	0.42	6.0	4.7	1.56	0.34

5 Advice for Practitioners

Like sorting programs, most programs can be optimized with respect to different criteria: the number of branch mispredictions, cache misses, element comparisons, or element moves. Optimizing one of the parameters may mean that the optimality with respect to another is lost. Not even the optimal bounds are the best in practice; the best choice depends on the environment where the programs are run. The task of a programmer is difficult. As any activity involving design, good programming requires good compromises.

In this paper we were interested in reducing the cost caused by branch mispredictions. In principle, there are two ways of removing branches from programs:

1. Store the result of a comparison in a Boolean variable and use this value in normal integer arithmetic (i.e. rely on the `setcc` family of instructions available in Intel processors).
2. Move the data from one place to another conditionally (i.e. rely on the `cmovcc` family of instructions available in Intel processors).

In Intel's architecture optimization reference manual [7, Section 3.4.1], a clear guideline is given how these two types of optimizations should be used.

> Use the `setcc` and `cmov[cc]` instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to `setcc` or `cmov[cc]` trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel ... processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.

Complicated optimizations often mean complicated programs with many branches. As a result, it will be more difficult to remove the branches by hand. Fortunately, an automatic way of eliminating all branches, except one, is known

[5]. However, the performance of the obtained program is not necessarily good due to the high constant factor introduced in the running time. In our work we got the best results by starting from a simple program and reducing branches from it by hand.

Implicitly, we assumed that the elements being manipulated are small. For large elements, it may be necessary to execute each element comparison and element move in a loop. However, in order for the general $O(n \log_2 n)$ bound for sorting to be valid, element comparisons and element moves must be constant-time operations. If this was not the case, e.g. if we were sorting strings of characters, the comparison-based methods would not be optimal any more. On the other hand, if the elements were large but of fixed length, the loops involved in element comparisons and element moves could be unfolded and conditional branches could be avoided. Nonetheless, the increase in the number of element moves can become significant.

At this point we can reveal that we started this research by trying to make quicksort lean (as was done with heapsort and mergesort in [5]). However, we had big problems in forcing the compiler(s) to use conditional moves, and our hand-written assembly-language code was slower than the code produced by the compiler. So be warned; it is not always easy to eliminate unpredictable branches without a significant penalty in performance.

6 Afterword

We leave it for the reader to decide whether quicksort should still be considered the quickest sorting algorithm. It is definitely an interesting randomized algorithm. However, many of the implementation enhancements proposed for it seem to have little relevance in contemporary computers.

We are clearly in favour of mergesort instead of quicksort. If extra memory is allowed, mergesort is stable. Multi-way mergesort removes most of the problems associated with expensive element moves. The algorithm itself does not—even though our in-situ mergesort does—use random access. This would facilitate an extension to the interface of the C++ standard-library sort function: The input sequence should only support forward iterators, not random-access iterators.

In algorithm education at many universities a programming language is used that is far from a raw machine. Under such circumstances it gives little meaning to talk about the branch-prediction features of sorting programs. A cursory examination showed that in one of our test computers (Ares) the Python standard-library sort was a factor of 5.79 slower than the C++ standard-library sort when sorting million integers.

References

1. Biggar, P., Nash, N., Williams, K., Gregg, D.: An experimental study of sorting and branch prediction. ACM Journal of Experimental Algorithmics 12, Article 1.8 (2008)

2. Brodal, G.S., Moruz, G.: Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In: Proceedings of the 9th International Workshop on Algorithms and Data Structures. Lecture Notes in Computer Science, vol. 3608, pp. 385–395. Springer-Verlag (2005)
3. Brodal, G.S., Moruz, G.: Skewed binary search trees. In: Proceedings of the 14th Annual European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 4168, pp. 708–719. Springer-Verlag (2006)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3nd edn. (2009)
5. Elmasry, A., Katajainen, J.: Lean programs, branch mispredictions, and sorting. In: Proceedings of the 6th International Conference on Fun with Algorithms. Lecture Notes in Computer Science, vol. 7288, pp. 119–130. Springer-Verlag (2012)
6. Hoare, C.A.R.: Quicksort. The Computer Journal 5(1), 10–16 (1962)
7. Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual, version 025 (1997–2011)
8. Kaligosi, K., Sanders, P.: How branch mispredictions affect quicksort. In: Proceedings of the 14th Annual European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 4168, pp. 780–791. Springer-Verlag (2006)
9. Katajainen, J., Pasanen, T., Teuhola, J.: Practical in-place mergesort. Nordic Journal of Computing 3(1), 27–40 (1996)
10. Katajainen, J., Träff, J.L.: A meticulous analysis of mergesort programs. In: Proceedings of the 3rd Italian Conference on Algorithms and Complexity. Lecture Notes in Computer Science, vol. 1203, pp. 217–228. Springer-Verlag (1997)
11. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall, 2nd edn. (1988)
12. Mortensen, S.: Refining the pure-C cost model. Master's Thesis, Department of Computer Science, University of Copenhagen (2001)
13. Musser, D.R.: Introspective sorting and selection algorithms. Software—Practice and Experience 27(8), 983–993 (1997)
14. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, The Hardware/Software Interface. Morgan Kaufmann Publishers, 4th edn. (2009)
15. Sanders, P., Winkel, S.: Super scalar sample sort. In: Proceedings of the 12th Annual European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 3221, pp. 784–796. Springer-Verlag (2004)
16. Sedgewick, R.: The analysis of Quicksort programs. Acta Informatica 7(4), 327–355 (1977)