Lean Programs, Branch Mispredictions, and Sorting*

Amr Elmasry and Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark
{elmasry, jyrki}@diku.dk

Abstract. According to a folk theorem, every program can be transformed into a program that produces the same output and only has one loop. We generalize this to a form where the resulting program has one loop and no other branches than the one associated with the loop control. For this branch, branch prediction is easy even for a static branch predictor. If the original program is of length κ, measured in the number of assembly-language instructions, and runs in $t(n)$ time for an input of size n, the transformed program is of length $O(\kappa)$ and runs in $O(\kappa t(n))$ time. Normally sorting programs are short, but still κ may be too large for practical purposes. Therefore, we provide more efficient hand-tailored heapsort and mergesort programs. Our programs retain most features of the original programs—e.g. they perform the same number of element comparisons—and they induce $O(1)$ branch mispredictions. On computers where branch mispredictions were expensive, some of our programs were, for integer data and small instances, faster than the counterparts in the GNU implementation of the C++ standard library.

1 Introduction

Sorting is a well-studied problem and, over the years, many sorting algorithms have been developed and analysed with respect to different performance measures like running time, memory overhead, number of element comparisons, number of element moves, and number of cache misses. Recently, some studies have been devoted to the analysis of the branch-prediction features of various sorting programs (see, for example, [2, 3, 8, 10]). This is the topic of the present paper. Brodal and Moruz [3] proved the following lower bound on the number of branch mispredictions induced by any comparison-based sorting program.


Dangerous material ahead!

Theorem 1. [3] *Consider a deterministic comparison-based sorting program P that sorts a sequence of n elements using $O(\beta n \log_2 n)$ element comparisons, $\beta > 1$. The number of branch mispredictions performed by P is $\Omega(n \log_\beta n)$.*


Dangerous material passed!

An observant reader may notice that there is a conflict between the statement of this theorem and the claims made in the abstract of this paper. There is something fishy going on here.

Before we can proceed, we have to fix our programming notation. We assume that the programs written, say, in C [7] are translated into pure C [6], which is a glorified assembly language with the syntax of C. Let `a`, `b`, `x`, `y`, and `z` be variables; `p` a pointer variable; and λ some label. For the sake of simplicity, we will not specify the types of the variables used. A *pure-C program* is a sequence of possibly labelled statements that are executed sequentially unless the order is altered by a branch statement. Let $\mathcal{A} = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$, $\mathcal{B} = \{\texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{<<}, \texttt{>>}\}$, and $\mathcal{C} = \{\texttt{<}, \texttt{<=}, \texttt{!=}, \texttt{>}, \texttt{>=}\}$ be the set of arithmetic, bitwise, and comparison operators defined in C with their normal meanings. Furthermore, let $\mathcal{U} = \{\texttt{-}, \texttt{\~{}}, \texttt{\&}\}$ be the set of allowed unary operators. Now all *pure-C instructions* are the following:

Load: `x = *p`.
Store: `*p = y`.
Move: `x = y`.
Unary operation: `x =` $\ominus$`y`, where $\ominus \in \mathcal{U}$.
Binary operation: `x = y` $\oplus$ `z`, where $\oplus \in \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$.
Conditional branch: if `(a` $\triangleleft$ `b)` **goto** λ, where $\triangleleft \in \mathcal{C}$.
Unconditional branch: goto λ.

Observe that an instruction like `x = y` $<$ `z` can be realized on a computer by subtracting `y` from `z` and denoting in `x` whether or not the answer is positive. Thus, this instruction does not involve any branch at all. The situation is different with conditional branches. The instruction to be executed after the branch is first known when the result of the comparison is available. In a pipelined computer, this may cause significant delays. To reduce these delays, some history of previous branch decisions is maintained by the hardware and this information is used to decide which instructions are to be executed speculatively. If this prediction fails, some work is wasted. Of course, one could try to execute both branches in parallel and continue from the correct place when the branch decision is known. However, this will lead to more complicated hardware design. For more details on branch prediction, we refer to any textbook on hardware architecture (e.g. [9]).

Now we can get back to Theorem 1: It holds under the assumption that every element comparison is followed by a conditional branch on the outcome of the comparison. However, this is not necessary, as also pointed out in [3]. In 2001, Mortensen [8] described a version of mergesort that performs $n \log_2 n + O(n)$ element comparisons and only induces a linear number of branch mispredictions. Using the technique described in [10] (cf. Section 2), one can modify heapsort such that it will achieve the same bound on the number of branch mispredictions.

In this paper we take the approach that we write our programs with only a few conditional branches. We say that such programs are *lean*. When the loops are branch-free, except the final conditional branch at the end, and when there are only a constant number of unnested branch-free loops, any reasonable branch predictor can handle the program by performing at most $O(1)$ branch mispredictions. For concreteness, we assume that the branch predictor used by the underlying hardware 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.

To make our programming task easier, we assume the availability of conditional moves[1]; using the C syntax this instruction is denoted as follows:

Conditional move: if `(a` $\triangleleft$ `b) x = y`, where $\triangleleft \in \mathcal{C}$.

This instruction, or some of its restricted form, is supported as a hardware primitive by most current computers. Even if it was not supported, it could be brought back to pure-C instructions without using any branches[2]:

```
*p = x
q = p + 1
*q = y
Δ = a ◁ b
r = p + Δ
x = *r
```

Our contribution goes far beyond sorting. Namely, we prove that every program can be converted into an equivalent form that induces $O(1)$ branch mispredictions. The next theorem states precisely how big the loss of efficiency is.

Theorem 2. *Let P be a program of length κ, measured as the number of pure-C instructions. Assume that the running time of P is $t(n)$ for an input of size n. There exists a program Q of length $O(\kappa)$ that is equivalent to P, runs in $O(\kappa t(n))$ time for the same input as P, and induces $O(1)$ branch mispredictions.*

We prove this theorem in Section 3. It shows that branch mispredictions can always be avoided. We hope that this observation will lead to some new developments in compiler optimizers. Normally sorting programs are short, but a slowdown by a factor of κ can be significant. Therefore, we will consider some more efficient transformations when making existing sorting programs (heapsort and mergesort) lean. We will also study the practical significance of such programs. Actually, in several situations our sorting functions are faster than the corresponding functions (`make_heap`, `sort_heap`, and `stable_sort`) available in the GNU implementation of the C++ standard library.

2 Appetizer: Heap Construction

A *binary heap*, invented by Williams [12], is a binary tree in which each node stores one element. This tree is nearly complete in the sense that all levels are full, except perhaps the last level where elements are stored at the leftmost nodes. The elements are kept in *heap order*, i.e. for each node its element is not smaller than the elements at its descendants. A binary heap can be conveniently represented in an array where the elements are stored in breadth-first order.

Let us now consider Floyd's heap-construction program [4] given in C++ in Fig. 1. This program processes the nodes of the given tree level by level in a bottom-up manner, starting from height one and handling the nodes at each

[1] Sofus Mortensen pointed out to us the usefulness of conditional moves.
[2] Alexandru Enescu made us aware of this transformation.

```
1  template <typename position, typename index, typename comparator>
2  void siftdown(position a, index i, index n, comparator less) {
3    typedef typename std::iterator_traits<position>::value_type element;
4    element copy = a[i];
5  loop:
6    index j = 2 * i;
7    if (j <= n) {
8      if (j < n) {
9        if (less(a[j], a[j + 1])) {
10         j = j + 1;
11       }
12     }
13     if (less(copy, a[j])) {
14       a[i] = a[j];
15       i = j;
16       goto loop;
17     }
18   }
19   a[i] = copy;
20 }
21
22 template <typename position, typename comparator>
23 void make_heap(position first, position beyond, comparator less) {
24   typedef typename std::iterator_traits<position>::difference_type index;
25   position const a = first - 1;
26   index const n = beyond - first;
27   for (index i = n / 2; i > 0; --i) {
28     siftdown(a, i, n, less);
29   }
30 }
```

Fig. 1. Floyd's heap-construction program in C++; the program has the same interface as the C++ standard-library function `make_heap`.

level from right to left, until the traversal reaches the root. For each node the program merges the two subheaps in the subtrees of that node by sifting the element down until the heap order is reestablished.

When studying the program, we saw several optimization opportunities:

opt$_1$: Remove the **if** statement on line 8. If `siftdown` is always called with an odd `n`, no node will ever have one child. If the total number of elements is even, insert the last element into the heap using the function `siftup` [12].

opt$_2$: Make the element moves on lines 4 and 19 conditional since they are unnecessary when the element at the root is not moved.

opt$_3$: Replace the lines 9–11 with `j += less(a[j], a[j + 1]);` this removes a branch for which the value of the condition is difficult to predict. This type of optimization was one of the key tools used by Sanders and Winkel [10].

After inlining `siftdown`, the program has the following structure:

while (*C*$_1$)
  S$_1$
  while (*C*$_2$)
    S$_2$
  S$_3$

where C_1 and C_2 are conditions; S_1, S_2, and S_3 are blocks of code. The nested loops can be fused by executing the statements in S_1 and S_3 under the condition

```
1  template <typename position, typename comparator>
2  void make_heap(position first, position beyond, comparator less) {
3    typedef typename std::iterator_traits<position>::difference_type index;
4    typedef typename std::iterator_traits<position>::value_type element;
5    position const a = first - 1;
6    index const n = beyond - first;
7    index const m = (n & 1) ? n : n - 1;
8    index i = m / 2;
9    index j = i;
10   index hole = j;
11   element copy;
12   while (i > 0) {
13     if (i == j) hole = j;
14     if (i == j) copy = a[j];
15     j = 2 * j;
16     j += less(a[j], a[j + 1]);
17     a[hole] = a[j];
18     bool smaller = less(copy, a[j]);
19     if (smaller) hole = j;
20     bool outer = (2 * j > m) || (! smaller);
21     if (outer) a[hole] = copy;
22     if (outer) i = i - 1;
23     if (outer) j = i;
24   }
25   siftup(a, n, less);
26 }
```

Fig. 2. Heap-construction program $\mathbf{F}_*$ inducing only $O(1)$ branch mispredictions.

that we are in the outer loop. The conditional branches can then be replaced by conditional moves. The outcome of these transformations is shown in Fig. 2.

We implemented the following programs, run some simple performance tests for four values of n, and measured the execution times used[3]. In each test the input was a random permutation of the numbers (of type `int`) from 0 to $n - 1$.

std: Heap construction using `make_heap` from the standard library.
F: Floyd's program given in Fig. 1.
$\mathbf{F}_1$**:** Floyd's program that uses the first optimization described above.
$\mathbf{F}_{123}$**:** Floyd's program that uses the three optimizations described above.
$\mathbf{F}_*$**:** Our lean version of Floyd's heap-construction program given in Fig. 2.

The results are reported in Table 1. As seen, it may be advantageous to avoid conditional branches. However, it does not seem to be so important to write absolutely branch-free programs. One problem encountered was that we could not force the compiler to use conditional moves, and our handwritten assembly code was much slower than that produced by the compiler.

[3] All the experiments discussed throughout the paper were carried out on a laptop computer (model Intel® Core™2 CPU P8700 @ 2.53GHz) running under Ubuntu 11.10 (Linux kernel 3.0.0-16-generic) using `g++` compiler (`gcc` version 4.6.1) with optimization level `-O3`. The size of L2 cache of this computer was about 3 MB and that of the main memory 3.8 GB. At optimization level `-O3`, the used compiler always attempted to transform conditional branches into branch-less equivalents. All execution times were measured using the function `gettimeofday` in `sys/time.h`. Initial micro-benchmarking showed that in this computer conditional moves were faster than conditional branches when the result of the branch condition was unpredictable.

Table 1. Execution times of some heap-construction programs. Each experiment was repeated $2^{26}/n$ times, each with a separate input array, and the average execution time divided by n is reported in nanoseconds.

Program / n	**std**	**F**	$\mathbf{F}_1$	$\mathbf{F}_{123}$	$\mathbf{F}_*$
2^{10}	16.1	11.4	10.3	6.4	7.1
2^{15}	16.0	11.4	10.5	6.8	7.5
2^{20}	19.9	16.2	16.1	10.0	11.4
2^{25}	20.8	16.4	15.6	12.9	14.6

3 General Program Transformation

In this section we prove Theorem 2. The proof turns out to be an incarnation of the well-known folk theorem[4] stating that any program can be transformed into an equivalent form that has only one loop. For an informative survey on this folk theorem, we refer to the paper by Harel [5], where he mentions that in fact there exists two types of proofs for this theorem: a local proof and a global proof. The local strategy relies on local program transformations; for example, how to eliminate nested loops (cf. Section 2), how to eliminate neighbouring loops, and how to distribute a loop over an **if** statement. However, the proof given here relies on the global strategy (since it is simpler).

The proof has two parts. In the first part any program is transformed into an equivalent pure-C program; here we assume that the reader can verify the details for how this is done. The second part is more interesting. Using pure C as a tool, the full proof of this part is quite straightforward.

Lemma 1. *Every program written*[5] *in C can be transformed into an equivalent pure-C program, provided that the usage of additional variables and memory is allowed. This transformation increases both the size and the running time of the original program by a constant factor.*

Sketch of Proof. This is what a compiler does. Any textbook on compiler construction (e.g. [1]) explains how recursion is removed, how function calls are handled, how loops are translated, and how **if** statements are translated into code with three-operand instructions; that is what pure C in principle provides. □

Lemma 2. *Every pure-C program can be transformed into an equivalent pure-C program that has only one conditional branch, provided that the usage of additional variables and memory is allowed. If the size of the original program is κ and the running time is $t(n)$ for an input of size n, then the size of the transformed program is $O(\kappa)$ and the running time is $O(\kappa t(n))$.*

[4] Andreas Milton Maniotis pointed out the relevance of this folk theorem to us.

[5] You can safely replace "C" with "a reasonable subset of your favourite programming language". The problem is that many languages have dark corners (e.g. exception handling in C++) that may not be possible to handle as efficiently as stated in the lemma.

Proof. For the sake of simplicity, we assume that the given pure-C program is a single function that has only one exit point at the end. Also, we assume that the last instruction is a noop. As stated, we assume that the number of pure-C instructions, including the noop, is κ. First, we number the original instructions from `1` to κ. Second, we transform each instruction separately. The very first instruction of the transformed program gets the label `1`; all other instructions can remain unlabelled. To simulate the original behaviour, we need three extra variables: γ is used as a program counter, and `s` and `t` are temporary variables.

We can assume that the given program does not have any conditional moves since these can be removed using the transformation described in Section 1. The task left is to show how each pure-C instruction is transformed. Let `i` be the line number of the instruction in question. In addition, assume that λ is the line number of the target of a branch.

Case	**Original instruction**	**Transformed code**
Load	`i: x = *p`	`t = *p` **if** `(`γ `== i) x = t` γ `=` γ `+ 1`

Store	`i: *p = y`	`t = *p`
		if `(`γ `== i) t = y`
		`*p = t`
		γ `=` γ `+ 1`

Move	`i: x = y`	**if** `(`γ `== i) x = y` γ `=` γ `+ 1`
Unary operation	`i: x =` $\ominus$`y`	`t =` $\ominus$`y` **if** `(`γ `== i) x = t` γ `=` γ `+ 1`
Binary operation	`i: x = y` $\oplus$ `z`	`t = y` $\oplus$ `z` **if** `(`γ `== i) x = t` γ `=` γ `+ 1`

Conditional branch	`i:` **if** `(a` $\triangleleft$ `b)` **goto** λ	`s = (a` $\triangleleft$ `b)`
		`t = (`γ `== i)`
		`t = s + t`
		`s =` γ `+ 1`
		if `(t == 2)` γ `=` λ
		if `(t != 2)` γ `= s`

Unconditional branch	`i:` **goto** λ	`s =` γ `+ 1`
		`t = (`γ `== i)`
		if `(t == 1)` γ `=` λ
		if `(t != 1)` γ `= s`

Last noop	κ`:`	**if** `(`γ `!=` κ`)` **goto** `1`

The last instruction is the only unconditional branch in the whole function.

We will leave it for the reader to verify that these transformations lead to a correct behaviour. Clearly, the claim about the size of the transformed program holds. Since in each iteration at least one instruction is executed, the running time can increase at most by a factor of κ. □

4 Heapsort

In heapsort [12], after building a binary heap, the maximum is removed from the root, the last element is put into that place, the heap is made one smaller, the maximum is put at the place of the earlier last element, and the heap order is reestablished by calling `siftdown`. This process is repeated until the heap is empty and the array contains the elements in sorted order. Now it is more difficult to avoid the extra branch inside the inner loop of `siftdown` as in $\mathbf{opt}_1$, since `n` is even every other call. Williams [12] solved this problem as follows:

$\mathbf{opt}_4$: Keep the last element in its place during the execution of `siftdown`. Due to this sentinel, the last real leaf has always a sibling. Since the sentinel is equal to the sifted element, `siftdown` will never visit that node.

Of the earlier optimizations, $\mathbf{opt}_1$ and $\mathbf{opt}_2$ are only relevant for heap construction, but this part of the code will not dominate the overall costs.

The structure of the sorting function is identical to that of the heap-construction function; there are two nested loops. Therefore, the loop fusion can be done in the same way as before. Naturally, we also implemented some ad-hoc improvements, e.g. whenever possible, we made conditional moves unconditional if the move is harmless. A lean version of the sorting function is given in Fig. 3.

For our experiments, we considered the following versions of heapsort.

```
1  template <typename position, typename comparator>
2  void sort_heap(position first, position beyond, comparator less) {
3    typedef typename std::iterator_traits<position>::difference_type index;
4    typedef typename std::iterator_traits<position>::value_type element;
5    index n = beyond - first;
6    if (n < 2) return;
7    position const a = first - 1;
8    element out = a[1];
9    element in = a[n];
10   index j = 1;
11   index hole = 1;
12   while (n > 2) {
13     j = 2 * j;
14     j += less(a[j], a[j + 1]);
15     a[hole] = a[j];
16     if (less(in, a[j])) hole = j;
17     bool outer = (2 * j >= n);
18     if (outer) a[hole] = in;
19     if (outer) a[n] = out;
20     if (outer) n = n - 1;
21     if (outer) j = 1;
22     if (outer) out = a[1];
23     if (outer) in = a[n];
24     if (outer) hole = 1;
25   }
26   if (less(a[2], a[1])) {
27     std::swap(a[1], a[2]);
28   }
29 }
```

Fig. 3. Sorting a heap such that at most $O(1)$ branch mispredictions are induced; the program has the same interface as the C++ standard-library function `sort_heap`.

Table 2. Execution times of some heapsort programs. Each experiment was repeated $2^{26}/n$ times, each with a separate input array, and the average execution time divided by $n \log_2 n$ is reported in nanoseconds.

Program / n	**std**	**F**	**W**	$\mathbf{W}_{34}$	$\mathbf{H}_*$
2^{10}	8.9	8.0	6.4	4.4	4.9
2^{15}	8.1	8.3	6.5	5.1	5.5
2^{20}	11.1	11.0	10.0	8.1	8.7
2^{25}	21.3	21.0	20.0	36.0	36.4

std: Heapsort using `make_heap` and `sort_heap` from the standard library.
F: Floyd's program taken from [4] and converted into C++.
W: Williams' program taken from [12] and converted into C++.
$\mathbf{W}_{34}$**:** Williams' program where the result of the comparison determining which child contains a smaller value is used in index arithmetic (as in $\mathbf{opt}_3$).
$\mathbf{H}_*$**:** Our lean version of heapsort; the heap-construction function is described in Fig. 2 and the sorting function in Fig. 3.

The observed execution times are reported in Table 2. The branch-optimized versions were fast for small problem instances, but they were unexceptionally slow for large problem instances. We could not find the reason for the poor behaviour. We suspected that this was due to caching, but we could not confirm this with our cache profiler (`valgrind`). We decided not to dwell on this anomaly, but we just report what happened to our programs in our test environment.

We should also mention some facts that are not visible from these figures. First, all element comparisons are performed inside the inner loops which means that we will not get any performance slowdown for them. Heapsort sorts an array of size n in-place with at most $2n \log_2 n + O(n)$ element comparisons, and this also holds for our lean version. Second, our transformation increases the number of (conditional) element moves since the moves done in the outer loop are performed for each iteration of the inner loop. For the lean version the number of element moves is upper bounded by $5n \log_2 n + O(n)$, whereas for Floyd's implementation the corresponding bound is $n \log_2 n + O(n)$. Third, for the library version, for random data when $n = 2^{25}$, the observed number of element comparisons and element moves was $\sim 1.01 n \log_2 n$ and $\sim 1.11 n \log_2 n$, respectively. In a generic environment, where the cost of an operation is not known, it is important that the number of these operations is as close as possible to the theoretical optimum.

5 Mergesort

In its basic form mergesort is a divide-and-conquer algorithm that splits the input into two pieces, sorts the pieces recursively, and merges the sorted pieces into one. In the literature, several variations of this theme have been proposed. We only consider two-way mergesort; two of its variants are relevant: 1) The aforementioned recursive top-down mergesort and 2) bottom-up mergesort that

merges the sorted subarrays pairwise level by level, starting with subarrays of size one, until the whole array is sorted. For both versions it is convenient to assume that, in addition to the input array `a` of size n, we have another auxiliary array `b` of the same size available. In bottom-up mergesort the two arrays are alternatively used as an input array. If the number of passes over the arrays is odd, one additional pass is made that moves the the elements from array `b` back to array `a`, providing an illusion that sorting is done in-place.

Of the existing mergesort programs, the bottom-up version seemed to be the simplest. We wanted to avoid special cases since, from our previous experience, we knew that they will cause extra overhead. To reduce the number of branch mispredictions to $O(n)$, we applied the following optimizations:

opt$_5$: Handle small subproblems separately. This is a standard optimization used by many implementations, but it is a mistake to rely on insertionsort since it will induce one branch misprediction per element. We instead scan the element array once and sort each chunk of four elements by a straight-line code that has no branches. (In brief, we simulate a sorting network for four elements since in such a network the element comparisons to be made are oblivious.) The whole scan only induces $O(1)$ branch mispredictions.

opt$_6$: Decouple element comparisons from branches by using conditional moves. This was the key optimization applied by Mortensen [8]. He used the back-to-back `merge` [11, Chapter 8], which simplifies the inner loop but complicates the outer loop. To avoid these complications, we modified the standard `merge`.

After applying **opt$_5$** and inlining `merge`, the main loop handling the remaining passes contained three nested loops. To take the final step and get the number of branch mispredictions from $O(n)$ to $O(1)$, we eliminated the nested loops one by one. The resulting program is given in Fig. 4. At the heart of the program, we merge two consecutive sorted subarrays, each with `size` elements, except at the end where the second subarray may not exist or the last subarray may be shorter than `size`. Initially, `size` is set to 4. Of the variables used, `i` is the index of the current element in the first subarray, `j` is the index of the current element in the second subarray, and `k` is the index of the current element in the output array. In every iteration one element is moved from array `a` to array `b` and the indices are updated accordingly. When `k` reaches the value `n`, we double `size`, reinitialize the indices, and swap the names of the two arrays. The value of `t1` is a boundary index for the first subarray, and the value of `t2` is the boundary index for the second subarray. In other words, `i` is always at most `t1`, `j` is always at most `t2`, and `k` is always at most `t2`. Once the merging of the two subarrays is done, i.e. when `k` = `t2`, both `t1` and `t2` are increased by the value of the variable `size` (note that neither `t1` nor `t2` can exceed `n`). The algorithm iterates as long as there are still subarrays to be merged, i.e. when `size` < `n`.

When processing n elements, in the initial four-element sort, the number of element comparisons and element moves is $O(n)$. In each of the remaining passes, in every iteration one element is moved from the input array to the output array and for each such move exactly one element comparison is performed. The number of remaining passes is bounded by $\log_2 n$, so the total number of element

```
template <typename input, typename output, typename index, typename comparator>
input remaining_passes(input a, output b, index n, comparator less) {
  index size = 4;
  index i = 0;
  index j = 0;
  index k = 0;
  index t1 = 0;
  index t2 = 0;
  input p = 0;
  while (size < n) {
    bool next = (k == t2);
    if (next) i = t2;
    if (next) t1 = std::min(t2 + size, n);
    if (next) t2 = std::min(t1 + size, n);
    if (next) j = t1;
    bool second = (i == t1) || ((j < t2) && less(a[j], a[i]));
    if (second) b[k] = a[j++];
    if (second == false) b[k] = a[i++];
    k += 1;
    bool outer = (k == n);
    if (outer) size = size << 1;
    if (outer) k = 0;
    if (outer) t2 = 0;
    p = a;
    if (outer) a = b;
    if (outer) b = p;
  }
  return a;
}
```

Fig. 4. A single-loop version of the remaining passes of bottom-up mergesort.

comparisons performed is at most $n \log_2 n + O(n)$. By a simple inspection of the code, it is seen that the number of (conditional) element moves performed is 2 per iteration, i.e. $2n \log_2 n + O(n)$ in total. For the standard-library version, both the number of element comparisons and that of element moves is $n \log_2 n + O(n)$.

In our practical experiments, we considered the following programs:

std: Bottom-up mergesort behind `stable_sort` from the standard library.

M: Mortensen's back-to-back top-down mergesort taken from the web page associated with his thesis [8].

$\mathbf{M}_{56}$: A version of bottom-up mergesort that applies the two optimizations mentioned above. This variant induces $O(n)$ branch mispredictions.

$\mathbf{M}_{*}$: Our lean version of bottom-up mergesort partially described in Fig. 4.

The running times of these programs in our test computer are reported in Table 3. Briefly, the results show that moderate optimization is good and extreme optimization for branch mispredictions does not pay off.

In our test environment, for integer data, the standard-library quicksort (`std::sort`) and mergesort (`std::stable_sort`) had the same speed within the limits of measurement accuracy. So, our moderately-optimized mergesort ($\mathbf{M}_{56}$) was faster than quicksort. An inspection of the assembly code revealed that for both `std::stable_sort` and $\mathbf{M}_{56}$ all unpredictable branches were followed by conditional moves. Moreover, by using a branch-prediction profiler (`valgrind`) we could confirm that the number of branch mispredictions per $n \log_2 n$ was

Table 3. Execution times of some mergesort programs. Each experiment was repeated $2^{26}/n$ times, each with a separate input array, and the average execution time divided by $n \log_2 n$ is reported in nanoseconds.

Program / n	std	M	$\mathbf{M}_{56}$	$\mathbf{M}_*$
2^{10}	4.2	4.7	3.3	6.8
2^{15}	3.9	4.7	3.3	7.2
2^{20}	4.1	4.8	3.6	7.7
2^{25}	3.9	4.8	3.7	7.9

lower for $\mathbf{M}_{56}$ than for `std::stable_sort`. The difference was little (for $n = 2^{20}$, 0.02 versus 0.07) and can be explained by the fact that small subproblems were handled differently. Finally, be aware that Mortensen's and our programs will fail if an extra array for n elements cannot be allocated; the standard-library mergesort will switch to an in-place version and is still able to perform its task.

6 Conclusion

Branch prediction affects the performance of your programs. Easy-to-predict branches are friendly and need not be eliminated. It may be a good idea to eliminate unpredictable branches as long as no extra complications are introduced.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. Pearson Education, Inc., 2nd edn. (2007)
2. Biggar, P., Nash, N., Williams, K., Gregg, D.: An experimental study of sorting and branch prediction. ACM J. Exp. Algorithmics 12, Article 1.8 (2008)
3. Brodal, G., Moruz, G.: Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In: 9th International Workshop on Algorithms and Data Structures. LNCS, vol. 3608, pp. 385–395. Springer (2005)
4. Floyd, R.W.: Algorithm 245: Treesort 3. Commun. ACM 7(12), 701 (1964)
5. Harel, D.: On folk theorems. Commun. ACM 23(7), 379–389 (1980)
6. Katajainen, J., Träff, J.L.: A meticulous analysis of mergesort programs. In: 3rd Italian Conference on Algorithms and Complexity. LNCS, vol. 1203, pp. 217–228. Springer (1997)
7. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall, 2nd edn. (1988)
8. Mortensen, S.: Refining the pure-C cost model. Master's Thesis, Department of Computer Science, University of Copenhagen (2001)
9. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, The Hardware/Software Interface. Morgan Kaufmann Publishers, 4th edn. (2009)
10. Sanders, P., Winkel, S.: Super scalar sample sort. In: 12th Annual European Symposium on Algorithms. LNCS, vol. 3221, pp. 784–796. Springer (2004)
11. Sedgewick, R.: Algorithms in C++, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching. Addison-Wesley Publishing Company, Inc., Reading, 3rd edn. (1998)
12. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)