- Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.

Low-level optimizations. Structure code to take advantage of the hardware capabilities.

- Unroll loops to reduce overhead and to enable further optimizations.
- Find ways to increase instruction-level parallelism by techniques such as multiple accumulators and reassociation.
- Rewrite conditional operations in a functional style to enable compilation via conditional data transfers.

A final word of advice to the reader is to be vigilant to avoid introducing errors as you rewrite programs in the interest of efficiency. It is very easy to make mistakes when introducing new variables, changing loop bounds, and making the code more complex overall. One useful technique is to use checking code to test each version of a function as it is being optimized, to ensure no bugs are introduced during this process. Checking code applies a series of tests to the new versions of a function and makes sure they yield the same results as the original. The set of test cases must become more extensive with highly optimized code, since there are more cases to consider. For example, checking code that uses loop unrolling requires testing for many different loop bounds to make sure it handles all of the different possible numbers of single-step iterations required at the end.

5.14 Identifying and Eliminating Performance Bottlenecks

Up to this point, we have only considered optimizing small programs, where there is some clear place in the program that limits its performance and therefore should be the focus of our optimization efforts. When working with large programs, even knowing where to focus our optimization efforts can be difficult. In this section, we describe how to use *code profilers*, analysis tools that collect performance data about a program as it executes. We also discuss some general principles of code optimization, including the implications of Amdahl's law, introduced in Section 1.9.1.

5.14.1 Program Profiling

Program *profiling* involves running a version of a program in which instrumentation code has been incorporated to determine how much time the different parts of the program require. It can be very useful for identifying the parts of a program we should focus on in our optimization efforts. One strength of profiling is that it can be performed while running the actual program on realistic benchmark data.

Unix systems provide the profiling program GPROF. This program generates two forms of information. First, it determines how much CPU time was spent for each of the functions in the program. Second, it computes a count of how many times each function gets called, categorized by which function performs the call. Both forms of information can be quite useful. The timings give a sense of

the relative importance of the different functions in determining the overall run time. The calling information allows us to understand the dynamic behavior of the program.

Profiling with GPROF requires three steps, as shown for a C program `prog.c`, which runs with command-line argument `file.txt`:

1. The program must be compiled and linked for profiling. With GCC (and other C compilers), this involves simply including the run-time flag `-pg` on the command line. It is important to ensure that the compiler does not attempt to perform any optimizations via inline substitution, or else the calls to functions may not be tabulated accurately. We use optimization flag `-Og`, guaranteeing that function calls will be tracked properly.

   ```
   linux> gcc -Og -pg prog.c -o prog
   ```

2. The program is then executed as usual:

   ```
   linux> ./prog file.txt
   ```

   It runs slightly (around a factor of 2) slower than normal, but otherwise the only difference is that it generates a file `gmon.out`.

3. GPROF is invoked to analyze the data in `gmon.out`:

   ```
   linux> gprof prog
   ```

The first part of the profile report lists the times spent executing the different functions, sorted in descending order. As an example, the following listing shows this part of the report for the three most time-consuming functions in a program:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 97.58    203.66   203.66        1   203.66   203.66  sort_words
  2.32    208.50     4.85   965027     0.00     0.00  find_ele_rec
  0.14    208.81     0.30 12511031     0.00     0.00  Strlen
```

Each row represents the time spent for all calls to some function. The first column indicates the percentage of the overall time spent on the function. The second shows the cumulative time spent by the functions up to and including the one on this row. The third shows the time spent on this particular function, and the fourth shows how many times it was called (not counting recursive calls). In our example, the function `sort_words` was called only once, but this single call required 203.66 seconds, while the function `find_ele_rec` was called 965,027 times (not including recursive calls), requiring a total of 4.85 seconds. Function `Strlen` computes the length of a string by calling the library function `strlen`. Library function calls are normally not shown in the results by GPROF. Their times are usually reported as part of the function calling them. By creating the "wrapper function" `Strlen`, we can reliably track the calls to `strlen`, showing that it was called 12,511,031 times but only requiring a total of 0.30 seconds.

The second part of the profile report shows the calling history of the functions. The following is the history for a recursive function `find_ele_rec`:

```
                                158655725             find_ele_rec [5]
                4.85    0.10  965027/965027      insert_string [4]
[5]       2.4   4.85    0.10  965027+158655725 find_ele_rec [5]
                0.08    0.01  363039/363039      save_string [8]
                0.00    0.01  363039/363039      new_ele [12]
                                158655725             find_ele_rec [5]
```

This history shows both the functions that called `find_ele_rec`, as well as the functions that it called. The first two lines show the calls to the function: 158,655,725 calls by itself recursively, and 965,027 calls by function `insert_string` (which is itself called 965,027 times). Function `find_ele_rec`, in turn, called two other functions, `save_string` and `new_ele`, each a total of 363,039 times.

From these call data, we can often infer useful information about the program behavior. For example, the function `find_ele_rec` is a recursive procedure that scans the linked list for a hash bucket looking for a particular string. For this function, comparing the number of recursive calls with the number of top-level calls provides statistical information about the lengths of the traversals through these lists. Given that their ratio is 164.4:1, we can infer that the program scanned an average of around 164 elements each time.

Some properties of GPROF are worth noting:

- The timing is not very precise. It is based on a simple *interval counting* scheme in which the compiled program maintains a counter for each function recording the time spent executing that function. The operating system causes the program to be interrupted at some regular time interval δ. Typical values of δ range between 1.0 and 10.0 milliseconds. It then determines what function the program was executing when the interrupt occurred and increments the counter for that function by δ. Of course, it may happen that this function just started executing and will shortly be completed, but it is assigned the full cost of the execution since the previous interrupt. Some other function may run between two interrupts and therefore not be charged any time at all.

  Over a long duration, this scheme works reasonably well. Statistically, every function should be charged according to the relative time spent executing it. For programs that run for less than around 1 second, however, the numbers should be viewed as only rough estimates.
- The calling information is quite reliable, assuming no inline substitutions have been performed. The compiled program maintains a counter for each combination of caller and callee. The appropriate counter is incremented every time a procedure is called.
- By default, the timings for library functions are not shown. Instead, these times are incorporated into the times for the calling functions.

5.14.2 Using a Profiler to Guide Optimization

As an example of using a profiler to guide program optimization, we created an application that involves several different tasks and data structures. This application analyzes the *n-gram* statistics of a text document, where an n-gram is a sequence of n words occurring in a document. For $n = 1$, we collect statistics on individual words, for $n = 2$ on pairs of words, and so on. For a given value of n, our program reads a text file, creates a table of unique n-grams and how many times each one occurs, then sorts the n-grams in descending order of occurrence.

As a benchmark, we ran it on a file consisting of the complete works of William Shakespeare, totaling 965,028 words, of which 23,706 are unique. We found that for $n = 1$, even a poorly written analysis program can readily process the entire file in under 1 second, and so we set $n = 2$ to make things more challenging. For the case of $n = 2$, n-grams are referred to as *bigrams* (pronounced "bye-grams"). We determined that Shakespeare's works contain 363,039 unique bigrams. The most common is "I am," occurring 1,892 times. Perhaps his most famous bigram, "to be," occurs 1,020 times. Fully 266,018 of the bigrams occur only once.

Our program consists of the following parts. We created multiple versions, starting with simple algorithms for the different parts and then replacing them with more sophisticated ones:

1. Each word is read from the file and converted to lowercase. Our initial version used the function `lower1` (Figure 5.7), which we know to have quadratic run time due to repeated calls to `strlen`.
2. A hash function is applied to the string to create a number between 0 and $s - 1$, for a hash table with s buckets. Our initial function simply summed the ASCII codes for the characters modulo s.
3. Each hash bucket is organized as a linked list. The program scans down this list looking for a matching entry. If one is found, the frequency for this n-gram is incremented. Otherwise, a new list element is created. Our initial version performed this operation recursively, inserting new elements at the end of the list.
4. Once the table has been generated, we sort all of the elements according to the frequencies. Our initial version used insertion sort.

Figure 5.38 shows the profile results for six different versions of our n-gram-frequency analysis program. For each version, we divide the time into the following categories:

Sort. Sorting n-grams by frequency

List. Scanning the linked list for a matching n-gram, inserting a new element if necessary

Lower. Converting strings to lowercase

Strlen. Computing string lengths

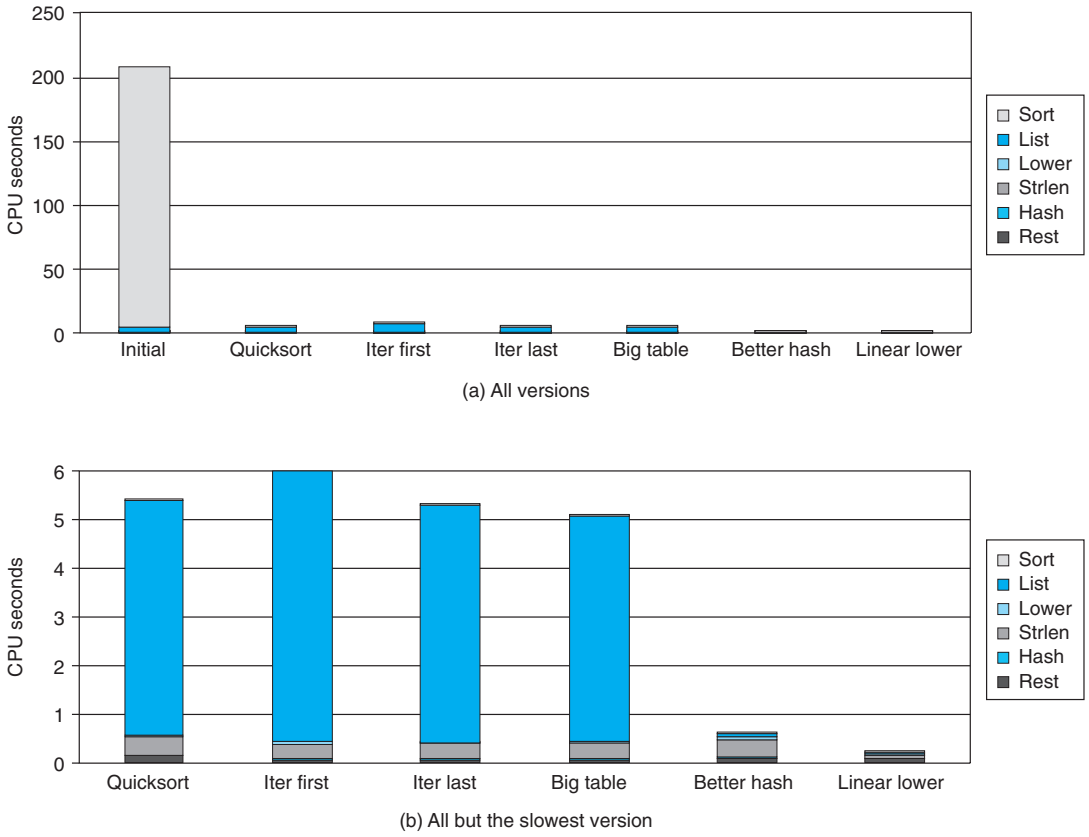

Figure 5.38 Profile results for different versions of bigram-frequency counting program. Time is divided according to the different major operations in the program.

Hash. Computing the hash function

Rest. The sum of all other functions

As part (a) of the figure shows, our initial version required 3.5 minutes, with most of the time spent sorting. This is not surprising, since insertion sort has quadratic run time and the program sorted 363,039 values.

In our next version, we performed sorting using the library function `qsort`, which is based on the quicksort algorithm [98]. It has an expected run time of $O(n \log n)$. This version is labeled "Quicksort" in the figure. The more efficient sorting algorithm reduces the time spent sorting to become negligible, and the overall run time to around 5.4 seconds. Part (b) of the figure shows the times for the remaining version on a scale where we can see them more clearly.

With improved sorting, we now find that list scanning becomes the bottleneck. Thinking that the inefficiency is due to the recursive structure of the function, we replaced it by an iterative one, shown as "Iter first." Surprisingly, the run time increases to around 7.5 seconds. On closer study, we find a subtle difference between the two list functions. The recursive version inserted new elements at the end of the list, while the iterative one inserted them at the front. To maximize performance, we want the most frequent n-grams to occur near the beginning of the lists. That way, the function will quickly locate the common cases. Assuming that n-grams are spread uniformly throughout the document, we would expect the first occurrence of a frequent one to come before that of a less frequent one. By inserting new n-grams at the end, the first function tended to order n-grams in descending order of frequency, while the second function tended to do just the opposite. We therefore created a third list-scanning function that uses iteration but inserts new elements at the end of this list. With this version, shown as "Iter last," the time dropped to around 5.3 seconds, slightly better than with the recursive version. These measurements demonstrate the importance of running experiments on a program as part of an optimization effort. We initially assumed that converting recursive code to iterative code would improve its performance and did not consider the distinction between adding to the end or to the beginning of a list.

Next, we consider the hash table structure. The initial version had only 1,021 buckets (typically, the number of buckets is chosen to be a prime number to enhance the ability of the hash function to distribute keys uniformly among the buckets). For a table with 363,039 entries, this would imply an average *load* of $363{,}039/1{,}021 = 355.6$. That explains why so much of the time is spent performing list operations—the searches involve testing a significant number of candidate n-grams. It also explains why the performance is so sensitive to the list ordering. We then increased the number of buckets to 199,999, reducing the average load to 1.8. Oddly enough, however, our overall run time only drops to 5.1 seconds, a difference of only 0.2 seconds.

On further inspection, we can see that the minimal performance gain with a larger table was due to a poor choice of hash function. Simply summing the character codes for a string does not produce a very wide range of values. In particular, the maximum code value for a letter is 122, and so a string of n characters will generate a sum of at most $122n$. The longest bigram in our document, "honorificabilitudinitatibus thou" sums to just 3,371, and so most of the buckets in our hash table will go unused. In addition, a commutative hash function, such as addition, does not differentiate among the different possible orderings of characters with a string. For example, the words "rat" and "tar" will generate the same sums.

We switched to a hash function that uses shift and EXCLUSIVE-OR operations. With this version, shown as "Better hash," the time drops to 0.6 seconds. A more systematic approach would be to study the distribution of keys among the buckets more carefully, making sure that it comes close to what one would expect if the hash function had a uniform output distribution.

Finally, we have reduced the run time to the point where most of the time is spent in `strlen`, and most of the calls to `strlen` occur as part of the lowercase conversion. We have already seen that function `lower1` has quadratic performance, especially for long strings. The words in this document are short enough to avoid the disastrous consequences of quadratic performance; the longest bigram is just 32 characters. Still, switching to `lower2`, shown as "Linear lower," yields a significant improvement, with the overall time dropping to around 0.2 seconds.

With this exercise, we have shown that code profiling can help drop the time required for a simple application from 3.5 minutes down to 0.2 seconds, yielding a performance gain of around 1,000×. The profiler helps us focus our attention on the most time-consuming parts of the program and also provides useful information about the procedure call structure. Some of the bottlenecks in our code, such as using a quadratic sort routine, are easy to anticipate, while others, such as whether to append to the beginning or end of a list, emerge only through a careful analysis.

We can see that profiling is a useful tool to have in the toolbox, but it should not be the only one. The timing measurements are imperfect, especially for shorter (less than 1 second) run times. More significantly, the results apply only to the particular data tested. For example, if we had run the original function on data consisting of a smaller number of longer strings, we would have found that the lowercase conversion routine was the major performance bottleneck. Even worse, if it only profiled documents with short words, we might never detect hidden bottlenecks such as the quadratic performance of `lower1`. In general, profiling can help us optimize for *typical* cases, assuming we run the program on representative data, but we should also make sure the program will have respectable performance for all possible cases. This mainly involves avoiding algorithms (such as insertion sort) and bad programming practices (such as `lower1`) that yield poor asymptotic performance.

Amdahl's law, described in Section 1.9.1, provides some additional insights into the performance gains that can be obtained by targeted optimizations. For our n-gram code, we saw the total execution time drop from 209.0 to 5.4 seconds when we replaced insertion sort by quicksort. The initial version spent 203.7 of its 209.0 seconds performing insertion sort, giving $\alpha = 0.974$, the fraction of time subject to speedup. With quicksort, the time spent sorting becomes negligible, giving a predicted speedup of $209/\alpha = 39.0$, close to the measured speedup of 38.5. We were able to gain a large speedup because sorting constituted a very large fraction of the overall execution time. However, when one bottleneck is eliminated, a new one arises, and so gaining additional speedup required focusing on other parts of the program.

5.15 Summary

Although most presentations on code optimization describe how compilers can generate efficient code, much can be done by an application programmer to assist the compiler in this task. No compiler can replace an inefficient algorithm or data