Analyzing the GNU Compiler Optimization of the Levenshtein Distance Algorithm

Mikhail Sinitcyn

August 2024

Abstract

The Levenshtein Distance algorithm is widely used to compute the similarity between two sequences. In this report, we analyze the GNU compiler optimizations resulting in a 35-50% reduction in runtime of the algorithm between optimization levels 0 and 1. We observe a 10-fold to 100-fold decrease in the percentage of branch misses for substring and duplicate sequence pairs.

The Levenshtein Distance Algorithm

Purpose

The Levenshtein Distance, is a measure of the similarity between two strings, commonly used in spell checking and fuzzy string matching. It is a dynamic $O(mn)$ space and time complexity algorithm that counts the number of single-character edits (insertions, deletions, or substitutions) needed to transform one string into another.

Algorithm

Per Jurafsky and Martin, 2024, the algorithm works by constructing a matrix where each cell represents the minimum number of edits required to transform a substring of the first string into a substring of the second string, per the following steps:

1. Initialize a matrix with dimensions (m+1) × (n+1), where m and n are the lengths of the two strings.
2. Fill the first row and column with incremental values (0, 1, 2, ...).
3. For each cell (i, j) in the matrix:
    - If the characters at positions i-1 and j-1 match, copy the value from the cell diagonally up and left.

- If they don't match, take the minimum of the three surrounding cells (left, up, diagonal) and add 1.

4. The value in the bottom-right cell (m,n) is the Levenshtein Distance.

Note that substitutions are counted as one edit in this implementation.

Example

Consider the classic example of calculating the Levenshtein Distance between "KITTEN" and "SITTING":

	K	I	T	T	E	N
S	1	2	3	4	5	6
I	2	1	2	3	4	5
T	3	2	1	2	3	4
T	4	3	2	1	2	3
I	5	4	3	2	2	3
N	6	5	4	3	3	2
G	7	6	5	4	4	3

Levenshtein Distance = 3

We find that the minimum edit (Levenshtein) distance between the two sequences is 3, per the bottom-right cell.

Implementation

This $O(mn)$ C implementation of the Levenshtein distance algorithm is taken from the Wikibooks reference, edited to allocate memory for the matrix on the heap rather than the stack such that a wider range of sequences can be evaluated without stack overflow.

Other Existing Implementations

More optimal implementations of the Levenshtein algorithm exist (O(min(m,n)) and approximate distance), but are not addressed in this report as they are algorithmic optimizations rather than compiler optimizations. Additionally, parallelized implementations are theoretically possible, yet not well documented (read Quickenshtein).

Evaluations

All evaluations were performed 1000 times on x86 architecture via the SFU CSIL machines. The following sequence pairs were used as input: Long-Long duplicate pair with edit distance of 0, Long-Long pair with edit distance of 8230, Long-Short duplicate prefix pair with edit distance 9536. The entire call to *levenshtein_distance*() was timed with the C time library. This includes calls to the *min*() function, heap memory allocation and de-allocation. Performance metrics were recorded with the Perf tool.

Metric	**O0**	**O1**
Branches	6.886e+8	4.582e+8
Branch Misses	1.598e+5	1.206e+5
Branch Misses %	2.321e-2	2.632e-2
Instructions	8.684e+9	2.876e+9
Time (s)	3.402e-1	1.831e-1

Table 1: Long-Long (Duplicates)

Metric	**O0**	**O1**
Branches	7.683e+8	5.144e+8
Branch Misses	2.065e+7	1.740e+7
Branch Misses %	2.687	3.384
Instructions	9.759e+9	3.217e+9
Time (s)	5.517e-1	3.175e-1

Table 2: Long-Long (Different)

Optimizations

Register Usage

At optimization level 0, the compiled code primarily uses variables stored on the stack, resulting in slower reads from Random Access Memory (RAM). By comparison, the optimized compilation stores loop counters and temporary variables in registers with up to 200 times faster reads than the stack.

Metric	O0	O1
Branches	5.141e+6	3.297e+6
Branch Misses	8.659e+3	8.212e+3
Branch Misses %	1.684e-1	2.501e-1
Instructions	6.025e+7	2.203e+7
Time (s)	8.003e-3	5.471e-3

Table 3: Long-Short (Substring)

Loop Optimization

At optimization level 1, the compiler combines a number of optimizations towards the loop structure. As mentioned previously, loop counters and temporary variables are stored in registers for faster reads. The compiler reduces the for-loop overhead by using pointer arithmetic to iterate over the matrix instead of using separate counter variables for each loop and updating with the *incl* instruction. Observe these differences in the compilation snippets below.

```
#C
    for (x = 1; x <= s2len; x++)
        matrix[x][0] = matrix[x-1][0] + 1;
#O0
        movq    -48(%rbp), %rax
        movq    (%rax), %rax
        movl    $0, (%rax)
        movl    $1, -20(%rbp)
        jmp     .L12
    .L13:
        movl    -20(%rbp), %eax
        decl    %eax
        movl    %eax, %eax
        leaq    0(,%rax,8), %rdx
        movq    -48(%rbp), %rax
        addq    %rdx, %rax
        movq    (%rax), %rax
        movl    (%rax), %edx
        movl    -20(%rbp), %eax
        leaq    0(,%rax,8), %rcx
        movq    -48(%rbp), %rax
        addq    %rcx, %rax
        movq    (%rax), %rax
        incl    %edx
        movl    %edx, (%rax)
        incl    -20(%rbp)
    .L12:
        movl    -20(%rbp), %eax
```

```
        cmpl    %eax, -36(%rbp)
        jnb     .L13
#O1
        movq    32(%rsp), %rdi
        testl   %edi, %edi
        je      .L11
        movq    %rsi, %rax
        leal    -1(%rdi), %edx
        leaq    8(%rsi,%rdx,8), %rdi
    .L12:
        movq    8(%rax), %rsi
        movq    (%rax), %rdx
        movl    (%rdx), %edx
        incl    %edx
        movl    %edx, (%rsi)
        addq    $8, %rax
        cmpq    %rdi, %rax
        jne     .L12
```

Notice that the O0 compilation uses a counter variable stored at -20(%rbp), incremented using incl -20(%rbp). The loop condition is checked by comparing this counter with the value at -36(%rbp).

The O1 optimized compilation uses the %rax register directly to iterate over the matrix instead of using a separate memory location for the loop counter. It increments the pointer using addq $8, %rax, which is faster than updating a memory location. The loop comparison uses a pre-computed end address in the %rdi register, thus there is no need to recalculate it with each iteration. It uses pointer arithmetic for more efficient array access, loading and storing values through the pointers stored in the row array.

Instruction Selection

Another source of optimization is the instruction selection. Consider the following comparison

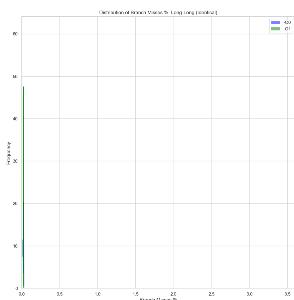
```
#O0
    levenshtein_distance:
        ...
        movl    -36(%rbp), %eax
        incl    %eax
        movl    %eax, %eax
        salq    $3, %rax
        movq    %rax, %rdi
        call    malloc

#1:
    levenshtein_distance:
```

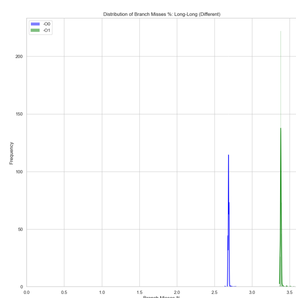
```
...
leal      1(%rax), %edi
salq      $3, %rdi
call      malloc
```

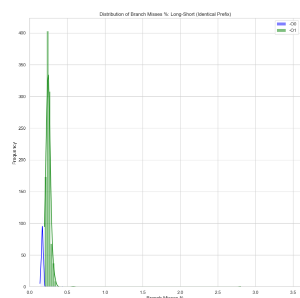
One clear example of improved instruction selection in the -O1 optimized version is the use of the leal (Load Effective Address) instruction to perform the increment and load in one step. We also observe that the the unoptimzed compilation moves the value from the eax register into itself, which is a non-operation.

Conditional Moves

The compiler uses conditional moves in the *min*() function, namely two instances of the *cmovle* instruction to reduce the number of conditional jumps in the compilation and reduce the number of its instructions by 50%. However, although conditional moves may reduce the number of branch mispredictions by computing branches in parallel and selecting the correct outcome, we do not observe a decrease in branch mispredictions.







We observe across all three sets of experiments with varying sequence pairs that, on average, the -O1 optimized code has, 35% less branches and 5-20% less branch misses, resulting in higher branch miss percentage.

Per Agner Fog's Optimization Manual, the performance benefits of conditional moves versus branches depend on the predictability of the branches and (over 75%). In the Levenshtein distance algorithm, the conditions for an arbitrary pair of sequences is not sufficiently predictable as such a pair does not have a predictable pattern of edit diffs. Confirming this heuristic, we observe a 10-fold and 100-fold decrease in percentage of branch misses for substring and duplicate sequence pairs, respectively.