Rishabh Kabra
Anurag Kashyap
6.172
24 September 2102

# Project 1 Beta Write-Up

*Overview:* We attempted to optimize a program (`bitarray.c`) that rotates bit arrays. The overarching strategy was to: (1) implement an asymptotically efficient bit-array rotation algorithm, (2) find bottlenecks in performance by profiling, (3) optimize those particular bottlenecks, and (4) implement Bentley's Rules-based optimizations where feasible. Via this strategy, our beta was able to achieve a **597,342x** performance boost relative to the original program (as measured on a quiet computer with the provided sample input of size on the order of ~$10^6$ bits). In parallel, a test suite was metaprogrammatically developed to thoroughly test the program, exploring as many permutations of inputs as possible—ultimately achieving **98.20%** test coverage (per `gcov`).

---

*Performance gains:* The original algorithm rotated each bit in an *n*-sized bit-array *k*-times to perform a *k*-wise circular rotation and thus exhibits $O(nk)$ runtime. Our first step was to move to the alternative $O(n)$ algorithm described in the project handout. Given a bit-array *B* with length `bitarray_length` to be rotated left by `bit_left_amount`, let *B* be the left-to-right concatenation, *L* + *R*, of a `bit_left_amount`-sized subarray *L* and a (`bitarray_length` – `bit_left_amount`) sized subarray *R*. Let `reverse`(*A*) represent the bit-array formed by reversing the bits of bit-array *A*. Then rotating *B* by `bit_left_amount` is equivalent to `reverse`(`reverse`(*L*) + `reverse`(*R*)). Using this linear-time algorithm and a bit-by-bit swapping implementation led to a **29,181x** gain in performance.

Our second consideration was to determine a better way to exploit the bitarray representation at the byte level. Considering the bits were being stored in a char buffer, we decided to speed up our array reversal implementation by moving and swapping whole chars instead of individual bits. Coupled with a table lookup to reverse individual chars[1], this word-level strategy for reversing subarrays proved effective and yielded a further **5.31x** performance gain.

Thirdly, we modified our data representation to further improve our reversal implementation. Since the workload of the reverse method working at the word-level was inversely proportional to the size of the word type (which was char, hitherto), it was intuitive to try using words larger than chars to store bits and operate on during reversal. So long as we could reverse our longer words fast enough, the decrease in work promised performance gains. We discovered empirically that **unsigned longs** proved ideal for our representation. A second bit hack[2] (based on dividing a long and using char reversal and bit-shifting) allowed us to reverse longs well, and being 64-bit types, unsigned longs made optimum use of register-capacity, thereby yielding another **2.27x** gain in performance.

A fourth optimization was adding predefined lookup tables for the bitmasks used to actually execute our word-level reversing and swapping. These bitmasks were previously calculated repeatedly, and using lookup tables gave us a small **1.05x** boost**.** Our last significant improvement came from

[1] Bit Twiddling Hacks, by Sean Eron Anderson (seander@cs.stanford.edu).
[2] Ibid.

optimizing our reversal implementation by partially unrolling loops, using better logical flow, and making fewer calls for reversing individual words. This yielded a final **1.51x** performance gain.

*Unsuccessful attempts*: Some strategies did prove unfruitful towards optimizing performance. A reverse lookup table for 16-bit shorts that decreased the number of operations required to reverse longs actually proved to be a performance hit. Likewise, using an extended char reverse lookup table (for reversing longs) that predefined differently shifted values also proved ineffective. Using a lookup table for the bitmask method that bitarray_get() and bitarray_set() rely on didn't yield any gains. Even the generally recommended use of pointer accesses instead of array indexing actually increased our runtimes.

---

*Correctness and tests*: We expect the program to be complete, at least with regard to input bit-arrays of maximum size on the order of ~4,000 bits. This upper bound exists because we could not use individual tests within our test suite that involved bit-arrays of size greater than 4,096 bits. Thus, if a bug were to arise within our program, we would expect it to arise when the program is given very large (greater than ~4,000 bits) bit-array inputs.

The test suite was generated by metaprogrammatically creating one or more tests for all permutations of: bitarray content, bitarray size, bitarray offset, bit subarray length, and rotation amount. The following were the different possible values for each component of the permutation:

- bitarray content: random bits, all zeroes, all ones, alternating, zeroes then ones
- bitarray size: 1, 5, 10, 20, 30, 60 +/- 1, 100 +/- 2, 250 +/- 5, 500 +/- 11, 1000 +/- 22, 2000 +/- 44, 4000 +/- 88
- bit offset: 0, 1, maximum offset, random offset
- bit subarray length: 0, 1, maximum subarray length, random subarray length
- rotation amount: 0, +/- 1, +/- (subarray length)/2, +/- (subarray length)/4, +/- (subarray length)/7, +/- (subarray length)/11

Using this approach, ~23,500 tests (32 MBs worth) were generated. In addition, some regression tests and corner case tests were handwritten. Our program passes all these tests yielding 98.20% test coverage, so we are confident of the code's general correctness. The only code not covered is the conditional code in `bitarray_new()` that is invoked when memory has not been allocated for the new bit-array such that the value of the bit-array is `NULL`; given the nature of the test suite, it is impossible to create a test that invokes said code.