Brian Chan
CS240
May 2, 2014

Bit Vectors

Definition

Bit Arrays, also known as bit vectors, is an array data structure which stores bits. The number of bits a bit array can store is x*y, where x is the number of bits in a machine word (ex. char, short, int), and y is a positive integer. Each bit inside the array is either 0 or 1, and can be set by using bitwise operators (ex. &, ^, |).

So what do these 0s and 1s mean? Well, they are often used to serve as flags. Take a survey or form, for example. Each bit in a bit array can correspond to a true or false answer to a question on the survey or form, such as “Do you own a car”, “Did you graduate from college”, or “Have you ever failed a drug test”, and so on. Additionally, bit arrays can be used for data compression or a priority queue.

Why use bit arrays? One advantage of Bit vectors is extreme compactness and low memory usage, as compared to similar implementation that do not use bit vectors. Such an implementation could be an array of integers, where each index is set to 0 or 1. In this case, 31 bits are wasted at each index in the array (assuming an int is 32 bits), since a 1 in binary is just 00000000 00000000 00000000 00000001. A bit array/bit vector solves this problem by using each individual bit in the machine word. Therefore, less memory must be allocated, which may be critical when working with large data sets. Another advantage of bit arrays is performance. Since most of the setting, getting, flipping, etc. of bits is done at a low level using bitwise operators, bit arrays often have an edge against other data structures.

Testing/Analysis

After I created a bit array (char based) library, I was curious how correct the above theories are. Thus, I also created an integer array implementation (possible alternative to bit vectors) and equal benchmarks/tests to see how each implementation performed against one another in raw speed and memory usage. To help test memory usage, I used valgrind, and for speed, the UNIX time command and GPROF. There are also various scripts I made to help automate the process (see the entire package).

For these tests, I used my personal laptop:

- IBM/Lenovo Thinkpad T60 (circa 2006) with:
- Intel Core 2 Duo T2500 @2.0ghz with 2MB cache
    - (Speedstep disabled for max processor speed and consistency)
- 2 GB DDR2-667 RAM
- Samsung 256 GB 840 PRO SSD
- Running 32 bit Arch Linux 3.14.1
- Intel GMA 950 graphics

Brian Chan
CS240
May 2, 2014

The Benchmark for both implementations:

A simple nested for loop that loops through the following *n* and *S* times for bit array or int array with array size *S* (see package for more details):

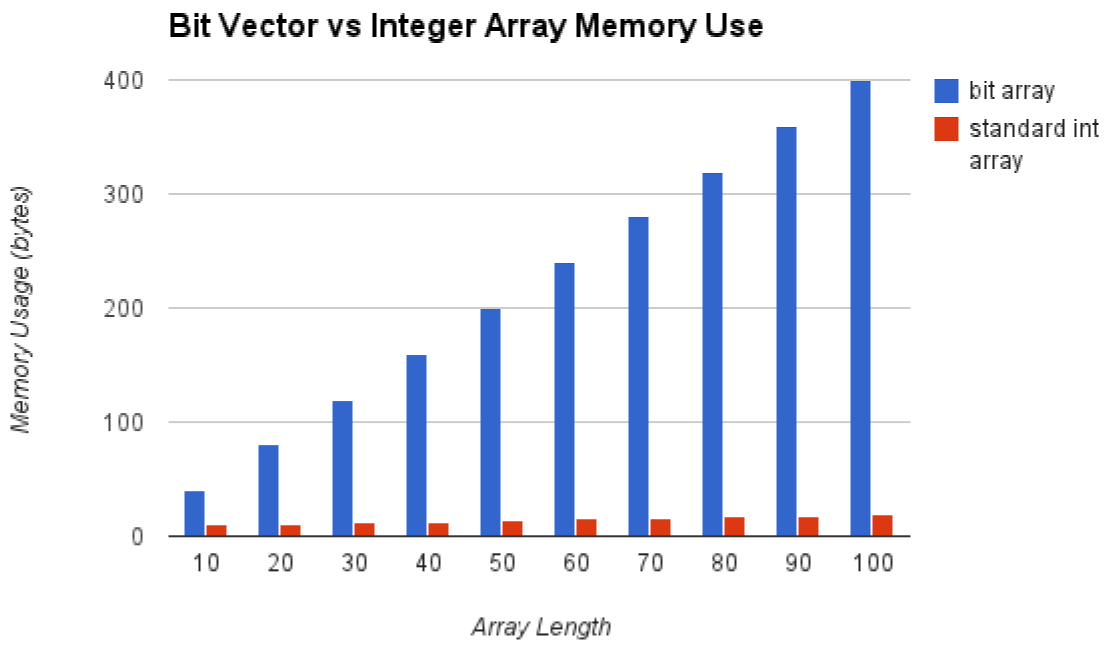
Outer loop (for number of tests/loops)**:**

1. Allocate space

   Inner loop (for each bit or index)**:**

   1. Sets a value at given offset to 1 or 0
   2. Gets a value at given offset
   3. Flips a value at given offset
2. Fills the array
3. Clears the array
4. Frees the array

Memory Usage



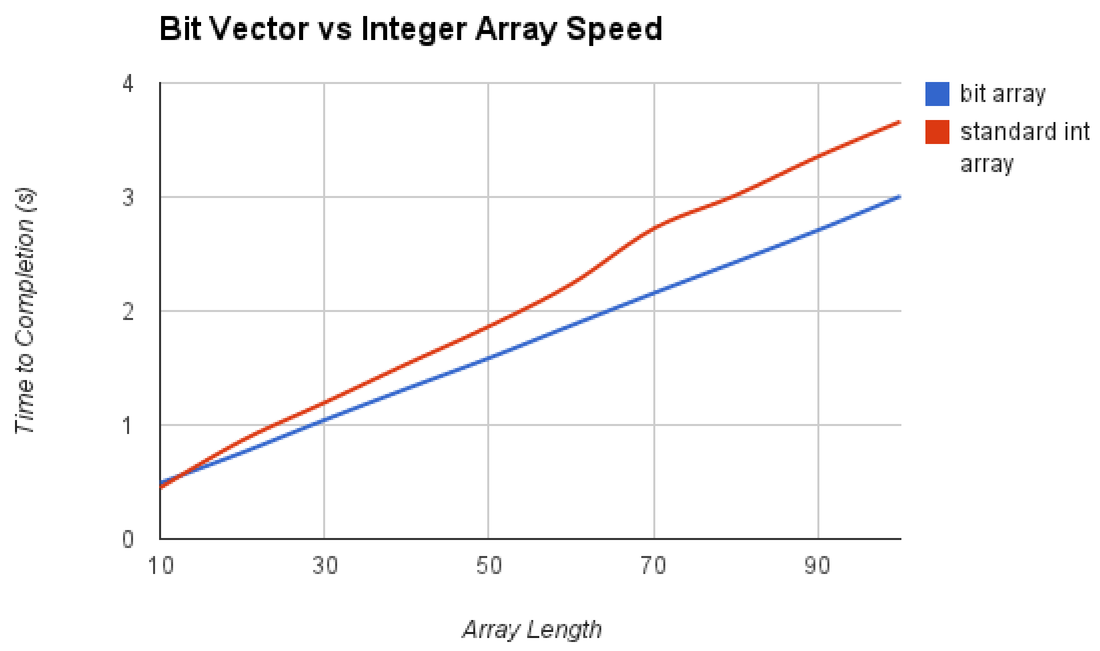
Brian Chan
CS240
May 2, 2014

	Standard Int array	**Bit array**
Array Length:	**Memory usage (bytes)**	**Memory usage (bytes)**
10	40	10
20	80	11
30	120	12
40	160	13
50	200	14
60	240	15
70	280	16
80	320	17
90	360	18
100	400	19

These tests were completed using the benchmark with one loop, while varying the array size. First, the most immediate observation is that the array of integers consumes much more memory than the bit array implementation. With an array of length 10, 50 and 100, the integer array already uses 4 times, 14.3 times, and 21 times more memory, respectively. Both implementations grow at a linear rate of big O of n. However, the difference in memory consumption as array length increases is exponential. Large data sets (ex. array length of hundreds or longer to store info from corporate employees) will especially benefit from bit arrays. From these results, bit arrays can store the same amount of information than using an array of integers, but with a far smaller footprint.

Brian Chan
CS240
May 2, 2014

Performance



	Standard Int array	**Bit array**
Array Length:	**Time to Completion (s)**	**Time to Completion (s)**
10	.449	.491
20	.865	.758
30	1.196	1.044
40	1.535	1.319
50	1.865	1.587
60	2.236	1.873
70	2.722	2.156
80	3.015	2.43
90	3.355	2.709
100	3.661	3.006

Gprof profile for Int array:

Brian Chan
CS240
May 2, 2014

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 27.27      0.12     0.12 10000000    12.00    12.00  flip_val
 18.18      0.20     0.08                             main
 15.91      0.27     0.07  1000000    70.00    70.00  clear_all
 14.77      0.34     0.07 10000000     6.50     6.50  get_val
 13.64      0.40     0.06  1000000    60.00    60.00  fill_all
 10.23      0.44     0.04 10000000     4.50     4.50  set_val
```

Gprof profile for bit array:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 28.21      0.11     0.11 10000000    11.00    11.00  set_bit
 25.64      0.21     0.10                             main
 23.08      0.30     0.09 10000000     9.00     9.00  flip_bit
 15.38      0.36     0.06 10000000     6.00     6.00  get_bit
  5.13      0.38     0.02  1000000    20.00    23.33  make_bit_arr
  2.56      0.39     0.01  3000000     3.33     3.33  reset_bit_arr
  0.00      0.39     0.00  2000000     0.00     3.33  clear_bit_arr
  0.00      0.39     0.00  1000000     0.00     3.33  fill_bit_arr
  0.00      0.39     0.00  1000000     0.00     0.00  free_bit_arr
```

The results of this performance test is interesting. First, the graph. In general, the bit array faster, but for very small arrays less than 15, the int array is faster. Both implementations scale linearly, but the difference in performance won't start to become significant until arrays of length 70 or more.

The Gprof results are even more startling. The calls to set_bit and get_bit in the bit array implementation is equal or slightly slower than their int array function counterparts. This may be due to the fact that for the int arrays, the computer is simply getting and returning the value at the offset, while in the bit array, the array index and bit index must be calculated, and then shifted and operated by the bitwise operators. Even though bitwise calculations are faster since they are done on a very low level, the amount of instructions needed is greater, and so the cummulative processing time is slightly longer. Additionally, most processors are optimized to access integers (32 bit) faster than any other data type (as a side note, I also created a bit array based on an integer word size, but found that there was actually a ~5% decrease in performance as opposed to my char based version). Contrastingly, clear, fill, and flip operations in the bit array implementation are significantly faster, probably since the cummalitive operations using the bitwise operators are fewer.

Brian Chan
CS240
May 2, 2014

Wrapping Up

My testing confirms that bit arrays indeed take up a significantly less amount of space to store the same amount of flags/data than in arrays of integers, which may be an alternative one might use instead. However, the performance benefits can be mixed. Getting operations take about the same time, but setting operations take a bit longer, since there simply more calculations involved. On a personal note, this exploration of bit arrays was also extremely interesting, along with the statistics and analysis.