A FASTER METHOD FOR DETERMINING FIRST/LAST ZEROES/ONES IN A BITSET VIA BITSET-STACKING

MATTHEW BENTLEY

Abstract. An O(1) technique for calculating the first or last 1 or 0 in a bitset via the use of 1 or more additional smaller bitsets representing the minimum/maximum value of words in the first bitset.

1. An example

In a given data storage center, there may be many ways to keep track of which hard drives are currently in use, but the simplest and most memory-efficient is probably a bitset ie. a single bit for each hard drive, 1 for occupied, 0 for free. Given that this method consumes the least possible memory (ignoring potential compression of the bitset), how do we utilize it to find a free hard drive? The logical answer is a linear scan over the bitset, since if it is small it will all fit in CPU cache, but for a large number of hard drives eg. 2 million, this may be too slow. The next logical step is to scan at the word level rather than at the bit level and look for the first non-maximum-value word (ie. a word which has at least one 0 present within it). This is advantageous performance-wise for two reasons: word-level access is much faster than bit-level access, which requires additional shift + AND/OR instructions, and by doing the initial scan at the word level, we divide the number of required operations by the bitwidth of the word.

Assuming a 64-bit platform in this example, a single word covers 64 hard drives. Once we've found a word which isn't at it's maximum value, we examine that word to find the first 0 in it. Most modern CPU's have instructions to do this specific task (eg. BSRL/BSRW on Intel x86 CPUs), and languages take advantage of these instructions in functions such as C++'s std::countr_zero. From the results of these we can calculate the bit index of the first 0 in the bitset (word-index * word-bitwidth + sub-word-index). This divides the potential total number of operations we need to do by 64 compared to linear bit scanning. This is good, but it can be better.

2. Definitions

For the purposes of this document, the following terms are defined:

Stack: A collection of more than one bitset, where bits in 'higher' bitsets in the stack represent some state of words in 'lower' bitsets.

Layer: A bitset which is part of a stack. The lowest layer is the initial bitset whose values we are predominantly concerned with ie. the actual data set. Every higher layer in the stack uses bits to represent some state of the words in the next layer below.

3. Bitset stacking

Imagine we take all the values of those words, and make them into a secondary bitset layer - assigning 1 for a maximum-value (all 1's) word in the first bitset, 0 for a non-maximum-value (at least one 0) word. What we have now done is further divided the number of operations we need to do by 64 - as we can now scan the secondary bitset initially to find the first non-maximum-value word ie. the first set of 64 words in the first bitset which aren't all at their maximum-value. We then examine that selected word to find the first 0 within, which gives us the position of the first non-maximum-value word in the first bitset - which we examine in turn to find the index of the first 0 within the first bitset.

This second layer is the size of the first bitset divided by the word-bitdepth. Assuming a 64-bit platform in our example, this is 64 times smaller. Hence, for a 16777216-bit bitset, the second bitset would be 262144 bits ie. 4096 words, or 1.6% the size of the first. But we needn't stop there. This process of stacking one bitset on top of another can be repeated as many times as desired - and each time, the number of scan operations reduces by the word bitdepth, while the additional memory requirement also reduces by the word bitdepth compared to the previous bitset layer in the stack. A third bitset in this example, representing the words in the second, would be 4096 bits, or 64 words long, again 1.6% the size of the second bitset. Since 1.6% of 1.6% is not enough to change the initial 1.6% up to 1.7%, we can say that the cost of bitset stacking for a 64-bit platform is always 1.6% regardless of how many layers are added.

By involving a third bitset stack in this particular example, we reduce the total number of operations down from (when only scanning the words in the first bitset and not using a stack) 262144 word scans + one word examination, to 64 word scans + three word examinations. The larger the bitset, the larger the number of layers in the bitset stack it will benefit from. To make things more efficient, each of these bitsets can be concatenated in memory such that there is only ever one allocation for the entire stack of bitsets. However a small bitset may not benefit from the technique - for example, in a bitset of 1024 bits the total number of words is 16 on a 64-bit platform, and it's debatable (and system-dependent) whether or not examining the first 16 bits of a single 64-bit word would be faster than just checking the status of 16 words.

The maximum performance advantage for bit-searching is gained when enough stacks are layered such that the topmost layer contains only a small number of words, and certainly less than a word-bit-width number of words ie. less than 64 on a 64-bit system. For example, if we can get down to 2 words at the topmost layer, that reduces the bit search down to 2 word scans, followed by 1 word examination for that layer and every layer beneath. When choosing how many words to reduce down to at the topmost layer, the balance point for performance can be established by knowing how many word scans we can complete, on a given system, in the same time as examining a single word for the first 1 or 0 bit. For some systems reducing the topmost layer down to a single word may be the best. But the greater the number of layers we have, the greater the cost when changing bits in the bottom bitset - something we'll explore later on.

For the example above we can add a fourth bitset layer of 1 word ie. 64 bits, and reduce the entire search operation down to 4 word examinations. The final stack layers are as follows:

(1) 16777216 bits/262144 words.
(2) 262144 bits/4096 words.
(3) 4096 bits/64 words.
(4) 64 bits/1 word.

4. Detecting the first 1 instead of the first 0

To scan for the first 1 in a bitset using this technique, we need to change the word representation in the bitset stack layers. Specifically, instead of the second bitset representing the maximum-value/non-maximum-value status of each word in the first bitset, it must represent the zero/non-zero status of each of those words. The same follows for any additional bitset layers - the third must represent the zero/non-zero status of words in the second bitset, and so on. We can see from this that a bitset stack designed for detecting the first 0 cannot also be used to detect the first 1. They are two separate approaches, though one could maintain separate stacks for both of those operations, at a doubling of cost in terms of memory usage and an increase in the overhead incurred when changing bits in the first bitset (this last will be described below).

5. Detecting the last 1 or 0 instead of the first

To scan for the last 0 or 1 in the first bitset, obviously we just scan bitset words from back to front instead of from front to back, in the highest level of the bitset stack.

6. Stacking disadvantages

The overhead for this technique occurs when words in the first bitset change from maximum-word-value to non-maximum-word-value or vice-versa (when searching for the first/last 0), or from zero to non-zero and vice-versa (when searching for the first/last 1). When this occurs we have to update the corresponding bit representing that word, in the second bitset. Finding that word in the second bitset is easy to do using the first bitset's word index, and dividing by the word bitdepth, then taking the remainder as the sub-word bit index. If we update this bit and find that the word it is within has, as above, changed from maximum-word-value to non-maximum-word-value or vice-versa (when searching for the first 0), or from zero to non-zero and vice-versa (when searching for the first 1), we must update the corresponding bit representing that word in the third bitset, and so on.

This is not a major cost - there is a branch decision, then (if the required change has occurred) a read of the second bitset, and another branch decision, and so on. Therefore the strength of benefit for bitset stacking depends largely on the ratio of bit searches to bit changes. If searches are infrequent compared to bit changes, it is likely this will not be the method of choice. In the hard drive example above, presumably we're searching for an unoccupied hard drive so that we can occupy it - hence there is a 1-to-1 ratio of bit searches to bit changes. Let's say we've created a bitset stack of 4 layers - the topmost being a single word with each bit representing the 64 words in the layer below. For the entire search-for-bit-and-modify-bit operation, we will have reduced a (without bitset stacking) 262144 word check + 1 word examination + 1 bit-change operation, down to a (with bitset stacking) 0 word check + 4 word examinations + (at maximum) 4 bit-change operations. That's a large improvement, even with the costs described for changing bits.

Note: the word status checks in the bitset stack (to determine whether or not the next layer up needs to be modified) could be avoided if branching is costly on a given system - in this case we would simply write the modified word's status to the next layer above, and repeat until the topmost layer is reached.

7. Further alterations

We can tweak this model. If desired we can make the second bitset layer represent not 1 word per bit, but multiple - for example, for the hard drive example, if we make the second bitset represent 4 words in the first bitset, per bit, then we get a second bitset of 65536 bits instead of 262144. The difference here is that a '1' bit in the second bitset would represent (in the case of searching for first 0) 4 maximum-value words in the first bitset, and a '0' bit in the second bitset would represent 4 words in the first bitset where at least 1 of those words is *not* at maximum value. Similarly if searching for first 1 in the first bitset, a '1' bit in the second bitset would represent 4 non-zero words in the first bitset, and a '0' bit in the second bitset would represent 4 zero words in the first bitset.

We can repeat the process for the third bitset layer, representing 4 words in the second layer per bit, it would reduce the size of the third (in our hard drive example) down from 4096 bits to 256. Of course, this increases the number of operations necessary when changing bits in the first bitset, as multiple words need to be checked per bit change. However if there is an extremely high search-to-bit-change ratio, it may be worth the cost in terms of reduction in memory usage for the additional bitsets.

8. Size considerations

At this point we've been talking purely in terms of initial bitsets whose size is a multiple of the system's word bit-width eg. 64 - which fits neatly into representing the word values into a second bitset and so on. However, for non-multiples-of-word-bitwidth sizes everything still works - we just need to store the size of each layer and make sure we don't go beyond the end of the actual number of bits represented, when reading bits rather than words. Essentially we end up with 1 or more 'empty' bits at the end of each bitset in the last word. In addition, when searching for first/last 1's we will want to fill those 'empty' bits with 0's so that we don't get false positives when searching at the word level. Likewise when searching for first/last 0's we would fill the 'empty' bits with 1's.

9. Summary of operations

9.1. When stack is set up for searching for first/last 0.

9.1.1. *Search for first/last 0.*

(1) Scan words of topmost bitset layer until a non-maximum-value word A is found - from front to back if searching for first 1, from back to front if searching for last.
(2) Examine that word to find the sub-word index B of the first 0.

(3) Calculate the word index C for the next layer down via the following: C = A * word-bitdepth + B.
(4) Move down to the next bitset layer in the stack.
(5) If this is not the bottommost bitset, make A = C then go to step 2.
(6) If this is the bottommost bitset, find the first 0 in word C.

9.1.2. *Change bit from 1 to 0.* :

(1) Check the status of the word which the bit is within, to see whether it is currently at maximum value (all 1's). Store this status as boolean k. Set the bit to 0.
(2) If k is true, find the bit s in the bitset above which corresponds to the changed word in this bitset. If the word containing s is not at maximum value, set k to false. Move to that layer and set s to 0. If there is a layer above this layer, repeat step 2.

9.1.3. *Change bit from 1 to 0 (branch-free alternative).* :

(1) Store the maximum-value status of the word which the bit is within as boolean k. Set the bit to 0.
(2) Repeat the following for all but the top-most layer: find the bit s in the bitset above which corresponds to the changed word in this bitset. Store the maximum-value status of the word containing s as boolean l. Set s to k, then set k to l. Move to that layer.

9.1.4. *Change bit from 0 to 1.* :

(1) Set the bit to 1. Check the status of the word which the bit is within, to see whether it is now at maximum value.
(2) If the word is at maximum value, set the corresponding bit in the layer above to one. Move to that layer. If there is a layer above this layer, check to see if the word containing the recently-set bit is now at maximum value and repeat step 2.

9.1.5. *Change bit from 0 to 1 (branch-free alternative).* :

(1) Set the bit to 1.
(2) Repeat the following for all but the top-most layer: Store the following status as k: whether the word which the bit was contained within is now at maximum value. Set the corresponding bit in the layer above to k. Move to that layer.

9.2. **When stack is set up for searching for first/last 1.**

9.2.1. *Search for first/last 1.*

(1) Scan words of topmost bitset layer until a non-zero word A is found - from front to back if searching for first 1, from back to front if searching for last.
(2) Examine that word to find the sub-word index B of the first 1.
(3) Calculate the word index C for the next layer down via the following: C = A * word-bitdepth + B.
(4) Move down to the next bitset layer in the stack.
(5) If this is not the bottommost bitset, make A = C then go to step 2.
(6) If this is the bottommost bitset, find the first 1 in word C.

9.2.2. *Change bit from 1 to 0.* :

(1) Set the bit to 0. Check the status of the word which the bit is within, to see whether it is now at 0.
(2) If the word is zero, set the corresponding bit in the layer above to zero. Move to that layer. If there is a layer above this layer, check to see if the word containing the recently-set bit is now zero and repeat step 2.

9.2.3. *Change bit from 1 to 0 (branch-free alternative).* :

(1) Set the bit to 0.
(2) Repeat the following for all but the top-most layer: Store the following status as k: whether the word which the bit was contained within is now equal to zero. Set the corresponding bit in the layer above to k. Move to that layer.

9.2.4. *Change bit from 0 to 1.* :

(1) Check the status of the word which the bit is within, to see whether it is currently zero. Store this status as boolean k. Set the bit to 1.
(2) If k is true, find the bit s in the bitset above which corresponds to the changed word in this bitset. If the word containing s is non-zero, set k to false. Move to that layer and set s to 1. If there is a layer above this layer, repeat step 2.

9.2.5. *Change bit from 0 to 1 (branch-free alternative).* :

(1) Store the non-zero status of the word which the bit is within as boolean k. Set the bit to 1.
(2) Repeat the following for all but the top-most layer: find the bit s in the bitset above which corresponds to the changed word in this bitset. Store the non-zero status of the word containing s as boolean l. Set s to k, then set k to l. Move to that layer.

10. Conclusion

By exerting minimal additional structuring with no additional allocation (if concatenating the multiple bitsets in the same memory chunk) and a very low additional memory cost (1/word-bitwidth %) we can significantly decrease the cost of searching for either the first or last 1/0 in a given bitset. It is unknown to me whether or not this technique can be used to increase the performance of other common bitset functionality, but I will leave that to others to digress upon.