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

MATTHEW BENTLEY

ABSTRACT. An O(log n) 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 of standard bitset limitations

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_one. 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 'lower' bitsets in the stack represent some state of words in 'higher' bitsets.

Layer: A bitset which is part of a stack. The highest layer is the initial bitset whose values we are predominantly concerned with ie. the actual data set. Every lower 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 additional memory cost of bitset stacking for a 64-bit platform is always 1.6% of a non-stacked bitset, 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 bottom 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 bottom 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 bottom 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 bottom 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 non-zero/zero status of each of those words. Specifically, a 0 bit in the second layer means all bits in the word it represents in the first layer are 0. If it is 1, not all of the bits in that word are 0. The same follows for any lower bitset layers - the third must represent the non-zero/zero status of words in the second bitset, and so on. So we can scan the lowest layer for the first non-0 word, and use the index of the first 1 within that word to find the word containing a 1 on the next layer up, repeating this until we reach the top layer. We can see from this that a single-bit bitset stack designed for detecting the first 0 cannot also be used to detect the first 1, as effectively 3 states (all bits in the word are 0, some bits are 0 and some are 1, all bits are 1) would be needed for the second layer instead of 2. A non-binary stack will be explored later.

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

To scan for the last 0 or 1 in the first bitset, we just scan bitset words from back to front instead of from front to back, starting in the lowest 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 bottom layer 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 below, and repeat until the bottom 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. Non-binary stacks

Considering the case where, in a 2-tier stack we want to be searching for either first/last zeroes *or* ones at different times, we could maintain two separate 2nd-tier layers - one for whether the 1st layer's word is zero/non-zero, and one for whether the 1st layer's words are maximum-value/non-maximum-value. However this complicates updates, increases the number of potential allocations, and gets worse the more layers you add.

An alternative is to store a singular 2nd-tier layer, but have it store 2-bit properties for each 1st layer word - specifically, two zeroes for the 1st layer's word being zero, two ones for the 1st layer's word being maximum-value, and a one and a zero for the 1st layer's word being neither zero nor maximum-value. By doing so we maintain the positive qualities of both a zero/non-zero-indicating layer, and a maximum-value/non-maximum-value-indicating layer.

That is to say, if we use 64-bit words, then the first word in the 2nd layer will represent the status of the first 32 words in the 1st layer. If that first 2nd layer word is at maximum-value, then all 32 words in the 1st layer are at maximum value. If it is zero, all 32 words in the 1st layer are zero. If it is neither zero nor maximum value, then some of the 32 words in the 1st layer are zero, some are at maximum value.

This means that if we are searching for the first 0 in the 1st layer's bitset, we can scan at the word level in the 2nd layer until we find the first word which isn't at maximum value. If we're searching for the first 1, we can scan at the word level in the 2nd layer until we find the first word which isn't zero. Hence our optimization of scanning at the word level for the 2nd layer is retained regardless of whether we're searching for zeroes or ones.

The 2nd layer has four potential values per 1st layer word, but we are only using 3. Perhaps there is a fourth scenario we can indicate with the fourth value (0 followed by 1) but I fail to imagine that.

We now need to make changes to the dual-bit values in the 2nd layer whenever the following occur in the first layer:

(1) A word changes from zero to non-zero.
(2) A word changes from non-zero to zero.
(3) A word changes from non-maximum-value to maximum-value.
(4) A word changes from maximum-value to non-maximum-value.

In the above list 1 and 3 could occur within the same action, as could 2 and 4 (we could have a word go from zero to maximum-value or vice-versa, essentially, if multiple bits are changed at once). 1/3 and 2/4 are however mutually-exclusive.

10. Summary of operations

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

10.1.1. *Search for first/last 0.*

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

10.1.2. *Search for next/previous 0 from a given index x.*

(1) Divide x by word bitdepth to find the word index m containing x.
(2) Perform modolu on x by word bitdepth to obtain the sub-word bit index s within word m.
(3) Scan the bits after/before s within word m to see if a 0 is found. If it is, operation is finished. If it is not, continue.
(4) Divide m by word bitdepth to find the word index n containing the bit representing m, in the layer below.
(5) Perform modolu by word bitdepth on m to obtain the sub-word bit index t within word n.

(6) Scan the bits after/before t within word n to see if a 0 is found.
(7) If it is not, and there is a layer below, copy n to m and t to s, move to the layer below and go back to step 5.
(8) If it is not and there is no layer below, check all words after/before (depending on whether we're searching for a next/previous 1) n until a non-maximum-value word is found. If no non-maximum-value words are found, the operation is finished - a next 0 cannot be found. If a non-maximum-value word is found, store it's index as n and scan the bits within to find the first/last 0.
(9) If a 0 is found, record this as t.
(10) Multiply n by word bitdepth and add t to find the index of the word containing a 0 on the next layer down. Store this as n, then scan that word to find the first/last 0. Store this as t. If this is the bottom layer, multiply n by word bitdepth and add t - this is the index of the next/previous 0 on the bottom layer. Operation is finished. If it is not the bottom layer, repeat this step.

10.1.3. *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 false, the update is finished. If k is true, find the bit s in the bitset below 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 below this layer, repeat step 2.

10.1.4. *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 bottom layer: find the bit s in the bitset below 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.

10.1.5. *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 not at maximum value, the update is finished. If the word *is* at maximum value, set the corresponding bit in the layer below to one. Move to that layer. If there is a layer below this layer, check to see if the word containing the recently-set bit is now at maximum value and repeat step 2.

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

(1) Set the bit to 1.
(2) Repeat the following for all but the bottom 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 below to k. Move to that layer.

10.1.7. *Change bit to either 1 or 0 (boolean v).* : // TODO

(1) Check to see if the word containing the bit is non-zero, store the result as boolean k. Store the prior maximum-value status of the word as l. Set the bit to v. Store the non-zero status of the changed word as p, and the maximum-value status as q.
(2) If $k == p$ and $l == q$, the update is finished. Otherwise continue.
(3) Find the 2-bit value in the layer below which corresponds to the changed bit and store this as s. Find the word containing s and store this as m. Store the non-zero status of m as k and the maximum-value status of m as l. Set the first bit of s to k and the second to l. If there is no layer below this layer, the update is finished. Otherwise continue.
(4) Store the non-zero status of the changed word as p, and the maximum-value status as q. Move up a layer and go back to step 2.

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

10.2.1. *Search for first/last 1.*

(1) Scan words of bottom 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/last 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/last 1 in word c.

10.2.2. *Search for next/previous 1 from a given index x.*

(1) Divide x by word bitdepth to find the word index m containing x.
(2) Perform modolu on x by word bitdepth to obtain the sub-word bit index s within word m.
(3) Scan the bits after/before (depending on whether we're searching for a next/previous 1) s within word m to see if a 1 is found. If it is, operation is finished. If it is not, continue.
(4) Divide m by word bitdepth to find the word index n containing the bit representing m, in the layer below.
(5) Perform modolu by word bitdepth on m to obtain the sub-word bit index t within word n.
(6) Scan the bits after/before t within word n to see if a 1 is found.
(7) If it is not, and there is a layer below, copy n to m and t to s, move to the layer below and go back to step 5.
(8) If it is not and there is no layer below, check all words after/before$n$ until a non-zero word is found. If no non-zero words are found, the operation is finished - a next 1 cannot be found. If a non-zero word is found, store it's index as n and scan the bits within to find the first/last 1.
(9) If a 1 is found, record this as t.
(10) Multiply n by word bitdepth and add t to find the index of the word containing a 1 on the next layer down. Store this as n, then scan that word to find the first/last 1. Store this as t. If this is the bottom layer, multiply n by word bitdepth and add t - this is the index of the next/previous 1 on the bottom layer. Operation is finished. If it is not the bottom layer, repeat this step.

10.2.3. *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 currently zero.
(2) If it's not, the update is finished. If it is, set the corresponding bit in the layer below to zero. Move to that layer. If there is a layer below that layer, check to see if the word containing the bit we just set is now zero and repeat step 2.

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

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

10.2.5. *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 false, the update is finished. If k is true, find the bit s in the bitset below 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 below this layer, repeat step 2.

10.2.6. *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 bottom layer: find the bit s in the bitset below 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.

10.3. **Non-binary stack.**

10.3.1. *Search for first/last 0.* Same as 10.1.1, but using 2-bit representations in all but the bottom layer.

10.3.2. *Search for first/last 1.* Same as 10.2.1, but using 2-bit representations in all but the bottom layer.

10.3.3. *Search for next/previous 0.* Same as 10.1.2, but using 2-bit representations in all but the bottom layer.

10.3.4. *Search for next/previous 1.* Same as 10.2.2, but using 2-bit representations in all but the bottom layer.

10.3.5. *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. Now check to see if the word is 0, and store this as l.
(2) If neither k nor l are true the update is finished. Otherwise continue.
(3) Find the 2-bit value s in the bitset layer below which corresponds to the changed word in this bitset.
(4) Find the word which s is within and store this location as m. Check to see if m is currently at maximum value and store this status as boolean n.

(5) If k is true, set s to 1. If k is not true but l is true, set s to 0.
(6) If there is no layer below this layer, the update is finished. If not, continue.
(7) Check to see if m is now 0, and store this status as l.
(8) Set k to n, and go back to step 2.

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

(1) Set the bit to 0. Find the word which the bit is within and store this as m.
(2) Repeat the following for all but the bottom layer: Store whether or not m is non-zero as boolean l. Find the 2-bit value s in the bitset layer below which corresponds to m. Set s to l. Set m to the word containing s. Move up a layer.

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

(1) Check to see if the word containing the bit is non-zero, store the result as boolean l. Set the bit to 1. Store the maximum-value status of the changed word as k.
(2) If neither l nor k are true the update is finished. Otherwise continue.
(3) Find the 2-bit value in the layer below which corresponds to the changed bit and store this as s. Find the word containing s and store this as m. Store the non-zero status of m as l. Set the first bit of s to 1 and the second to k. If there is no layer below this layer, the update is finished. Otherwise continue.
(4) Set the maximum-value status of m as k. Move up a layer and go back to step 2.

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

(1) Set the bit to 1. Find the word which the bit is within and store this as m.
(2) Repeat the following for all but the bottom layer: Store the maximum-value status of m as k. Find the 2-bit value in the layer below which corresponds to the changed bit and store this as s. Find the word containing s and store this as m. Set the first bit of s to 1 and the second to k.

10.3.9. *Change bit to either 1 or 0 (boolean v).* :

(1) Check to see if the word containing the bit is non-zero, store the result as boolean k. Store the prior maximum-value status of the word as l. Set the bit to v. Store the non-zero status of the changed word as p, and the maximum-value status as q.
(2) If $k == p$ and $l == q$, the update is finished. Otherwise continue.
(3) Find the 2-bit value in the layer below which corresponds to the changed bit and store this as s. Find the word containing s and store this as m. Store the non-zero status of m as k and the maximum-value status of m as l. Set the first bit of s to k and the second to l. If there is no layer below this layer, the update is finished. Otherwise continue.
(4) Store the non-zero status of the changed word as p, and the maximum-value status as q. Move up a layer and go back to step 2.

10.3.10. *Change bit to either 1 or 0 (boolean v) (branch-free alternative).* :

(1) Set the bit to v. Find the word which the bit is within and store this as m.
(2) Repeat the following for all but the bottom layer: Store the non-zero status of m as k and the maximum-value status of m as l. Find the 2-bit value in the layer below which corresponds to the

changed bit and store this as s. Find the word containing s and store this as m. Set the first bit of s to k and the second to l.

11. Conclusion

By exerting additional structuring with no additional numbers of allocations (if concatenating the multiple bitsets in the same memory chunk) and a very low additional memory cost (100/word-bitwidth %) we can significantly decrease the cost of searching for either the first or last 1/0 in a given bitset (or both). 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 think upon.