A linear lower bound for incrementing a space-optimal integer representation in the bit-probe model

Michael Raskin*

raskin@{cs.au.dk,mccme.ru}

Computer Science department, Aarhus University

July 4, 2016

Abstract

I consider the problem of representing integers in close to optimal number of bits to support increment and decrement operations efficiently as described in [BGPS2014]. The problem is studied in the bit probe model. I analyse the number of bits read to perform the operation in the worst case.

Such representations together with the corresponding increment and decrement algorithms are called counters.

The article [BGPS2014] contains two basic constructions: one uses $n$ bits to represent exactly $2^n$ values and requires $n - 1$ reads in the worst case, the other one is redundant (represents less than $2^n$ values), but requires only a logarithmic number of reads for incrementing.

The only known lower bound for number of reads to increment a non-redundant counter was logarithmic. Reducing the gap between the linear upper bound and logarithmic lower bound remained an open problem.

I present a proof of a linear lower bound for the number of bits read in the worst case.

Keywords: binary counter, data structure, integer representation, bit-probe model, lower bound.

Basic definitions

The algorithms operate on bit strings with random access.

The basic input/output operations are: reading a bit with specified number and setting a bit with a specified number to a specified state.

The cost of an algorithm is described by the number of reads it does in the worst case.

Every such algorithm specifies a function from bit strings to bit strings of the same length. Repeated application of such a function will always lead to a cycle.

An algorithm implements a space-optimal binary counter of length $n$ if the algorithm’s iterated application to any bit string of length $n$ yields all the possible bit strings of length $n$ in some order.

A binary counter that is not space-optimal is called redundant.

*The author acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation and from the Center for Research in Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council.
Previously known examples

There is a space-optimal counter presented in [BGPS2014] that never reads all the \( n \) bits, but has to read \( n - 1 \) bits in the worst case. The only known lower bound was logarithmic.

There are also redundant counters using \( n \) bits to represent numbers in a range greater than \( \{0, \ldots , 2^n - 1\} \) while reading \( (\log n) \times (1 + o(1)) \) bits in the worst case. In this case the logarithmic lower bound turned out to be tight.

The bound

The main result of the present paper is: a space-optimal counter on \( n \) bits must have the worst-case number of reads greater or equal to \( \lfloor \frac{n}{2} \rfloor \) (half of all the bits rounded down).

In other words, it is impossible to have a space-optimal counter with complexity equal to \( F(n) := \lfloor \frac{n}{2} \rfloor - 1 \) reads.

The counter with \( n - 1 \) reads

To illustrate some of the notation, we will use the space-optimal counter from [BGPS2014]. It was initially presented as a decision tree (the inner nodes contain variables read, the algorithm moves down-left on reading 0 and down-right on reading 1):

![Decision Tree Diagram]

This algorithm corresponds to the following encoding of integers in the range \( \{0, \ldots , 15\} \):

| code | \( enc^{-1}(code) \) | code | \( enc^{-1}(code) \) | code | \( enc^{-1}(code) \) | code | \( enc^{-1}(code) \) |
|------|-----------------|------|-----------------|------|-----------------|------|-----------------|
| 0000 | 0               | 0100 | 6               | 1000 | 8               | 0010 | 2               |
| 0001 | 1               | 1001 | 9               | 0111 | 7               | 1010 | 13              |
| 0100 | 4               | 1100 | 12              | 1111 | 15              | 0010 | 2               |
| 0101 | 5               | 1110 | 14              | 1011 | 11              | 0011 | 3               |

In the present article the encoding function will be ignored in favour of the increment function that the algorithm has to implement. The increment function is related to the encoding function by the following equality: \( inc(c) = enc(enc^{-1}(c) + 1) \).

Iterating the increment function should give us a cycle of length \( 2^n \). In the example from [BGPS2014] the cycle is:

| code | inc(code) | code | inc(code) | code | inc(code) | code | inc(code) |
|------|-----------|------|-----------|------|-----------|------|-----------|
| 0000 | 0001      | 1001 | 9         | 0110 | 6         | 1000 | 8         |
| 0001 | 1         | 1001 | 9         | 0111 | 7         | 1111 | 15        |
| 0100 | 4         | 1100 | 12        | 1111 | 15        | 0010 | 2         |
| 0101 | 5         | 1110 | 14        | 1011 | 11        | 1000 | 8         |
|      |           |      |           |      |           |      |           |
We can write the same table as the table of function values (sorted by argument):

| code | inc(code) | code | inc(code) | code | inc(code) | code | inc(code) |
|------|----------|------|----------|------|----------|------|----------|
| 0000 | 0001     | 0001 | 0100     | 0101 | 1000     | 1010 | 1110     |
| 0110 | 0111     | 1100 | 0010     | 0011 | 1101     | 1100 | 1111     |
| 1000 | 1011     | 1001 | 0100     | 0101 | 1101     | 1100 | 1111     |
| 1110 | 1011     | 1001 | 0010     | 0011 | 1111     | 1110 | 1111     |

Binary counter as hypercube shuffling

Hypercube

All the codes used by an $n$-bit binary counter can be seen as vertices of an $n$-dimensional hypercube. We will identify vertices with the codes.

Bit writes as block moves

If we have read some bits (but not all of them), we have found out that the current vertex lies inside some block (a hypercube of lower dimension). If we change some of the opened bits in the counter, this is the same as if we do a parallel translation of the block.

If we read no more than $n - 3$ bits for each increment (and $\forall n : F(n) < max(0, n - 3)$), the blocks are at least $2 \times 2 \times 2$ cubes.

An example visualisation

The 4-bit counter needing 3 reads for an increment corresponds to a 4-dimensional hypercube. It is more convenient to draw it as two 3-dimensional cubes side by side (so the extra coordinate is projected to the vector proportionate to the projection of the first coordinate).
Both pictures use the same set of arrows to represent the movement of the blocks. The top picture shows the blocks before the moves, and the bottom picture shows the blocks after the moves.

We can list the vertices by block:

| block | vertices | vertices (decimal) | block | vertices | vertices (decimal) |
|-------|----------|-------------------|-------|----------|-------------------|
| a     | 0000 and 0100 | 0 and 4         | e     | 0001 and 1001 | 1 and 9           |
| b     | 1000 and 1100 | 8 and 12        | f     | 0111 and 1011 | 3 and 11          |
| c     | 0010 and 0110 | 2 and 6         | g     | 0101 and 0111 | 5 and 7           |
| d     | 1010 and 1110 | 10 and 14       | h     | 1101 and 1111 | 13 and 15         |

and after the blocks are shifted we get a new split:

| block | vertices | vertices (decimal) | block | vertices | vertices (decimal) |
|-------|----------|-------------------|-------|----------|-------------------|
| a     | 0001 and 0101 | 1 and 5         | e     | 0100 and 1100 | 4 and 12          |
| b     | 1010 and 1110 | 10 and 14       | f     | 0000 and 1000 | 0 and 8           |
| c     | 0011 and 0111 | 3 and 7         | g     | 1101 and 1111 | 13 and 15         |
| d     | 0010 and 0110 | 2 and 6         | h     | 1001 and 1011 | 9 and 11          |

**Specifying a block**

Each block can be fully described by specifying which bits are fixed and which are variables, and additionally specifying the values of all the fixed bits. For example, the block «a» in the example is defined by the mask «0*00».
Hypercube shuffling as a permutation

The increment algorithm describes a function. If the counter is space-optimal, all the possible bit strings represent different integers and should have different images. This means that the algorithm describes a permutation \( inc \) on the set of the vertices of the hypercube of all the possible bit strings.

This permutation has to be a cycle of length \( 2^n \). Any such cycle is an odd permutation.

It suffices to prove that every permutation described by shifting blocks of at least \( n - F(n) \) dimensions will be an even permutation.

Ordering

It will be convenient to have a standard order of enumerating the vertices of the hypercube. We will enumerate the codes in the standard lexicographical order of representing bit strings. The «first» bit from the algorithm’s explanation is used as the least significant bit.

In this ordering each block is defined by fixing some subset of bits in the binary representation. We can consider three different enumerations of the vertices in the chosen order (we can enumerate the vertices according to the position of the blocks before or after the shift corresponding to running a single increment step).

Let’s call \( \text{Before}(k) \) the code of the \( k \)-th vertex in the enumeration corresponding to the block placement before the shifts. Its image under increment is \( \text{After}(k) \), i.e. the \( k \)-th vertex in the enumeration corresponding to the block placement after the shifts (it is the vertex with the same relative placement in the same block, but after the shift, i.e. the image). Obviously, the functions \( \text{Before} \) and \( \text{After} \) are permutations. We want to prove that \( \text{inc} = \text{After} \circ \text{Before}^{-1} \) is an even permutation. It is enough to show that \( \text{Before} \) and \( \text{After} \) are even permutations when we read fewer than \( \lfloor \frac{n}{2} \rfloor \) bits.

For the example algorithm the \( \text{Before}(-) \) numbering is:

\[
\begin{align*}
(0000 & 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111) \\
(0000 & 0100 1000 1100 1100 0010 0110 1010 1110 0001 1001 0011 1001 0101 0111 1101 1111)
\end{align*}
\]

in binary, or \((0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)\) in decimal notation. The \( \text{After} \) numbering is

\[
\begin{align*}
(0000 & 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111) \\
(0001 & 0101 1010 1110 0011 0111 0010 0110 1100 0000 1000 1101 1111 1001 1011)
\end{align*}
\]

or \((0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)\). We first enumerate the two vertices in the \( a \) block, then the two vertices in the \( b \) block, etc. using the positions of the blocks before and after the move, respectively.
We can see that the Before permutation is odd and the After permutation is even (the example algorithm reads more than a half of all the bits, so this is not a counterexample).

As an illustration, 10 is the first vertex in the $d$ block before the shift, $\text{inc}(10) = 2$. The first vertex in the block $d$ (the fourth block) has number 6 in the Before ordering; we see that $\text{Before}^{-1}(10) = 6$. After the shift the first vertex in the block $d$ is 2. We see that $\text{After}(6) = 2$ and $\text{inc}(10) = \text{After}(\text{Before}^{-1}(10)) = (\text{After} \circ \text{Before}^{-1})(10) = 2$.

Both of the numberings Before and After are specified in the same way, by cutting the hypercube into blocks and enumerating the blocks. It is enough to show that any permutation specified in that way with blocks that have no more than $F(n)$ fixed coordinates is even.

The example algorithm has three fixed coordinates out of four for every block, so it is not a counterexample to the presented claim.

### Blocks and the inversions

The vertices inside each block are enumerated in a monotonously increasing order, so there are no inversions containing two vertices inside the same block. Two vertices from the different blocks constitute an inversion if the vertex from the block with a lower number has a higher number in the global lexicographical order. Therefore to find the parity of the number of inversions between the vertices in the given two blocks we need to find the parity of the number of pairs $(x, y)$ where $x > y$ and $x$ is taken from the first block and $y$ is taken from the second one. Note that since the blocks are disjoint and the total number of pairs of vertices from the two given blocks is even, the ordering of the blocks is not important.

We want to show that for every two blocks this number is even.

So let two blocks be specified by their free and fixed coordinates. We can write one on top of the other one. $\text{00}^* \cdots$

For each position there are the following possibilities:

- both blocks have a fixed coordinate, and they are equal;
- both blocks have a fixed coordinate, and they differ;
- one block has a fixed coordinate and the other block has a variable coordinate;
- both blocks have variable coordinates.

Note that $2F(n) \leq n - 2$ and each block can have only $F(n)$ fixed positions. Therefore there are at least two positions where both blocks have variable coordinates. Let us consider the most significant of all such positions and call it $p$.

The number of pairs of codes where the comparison can be made without considering bits on the position $p$ (and less significant positions) is even, because half of these codes has 0 in the first block on the position $p$ and the other half has 1.

If we have to consider the bits at the position $p$, we only consider the bits in the less significant positions if we find 0 or 1 at the position $p$. But these two cases are symmetrical and they cancel each other’s effect on parity.

If we have to consider the bits at the position $p$ but not any less significant bits, we ignore some of the variable bits at the positions less significant than $p$ (we know there is at least one more $*$ position), so we can only get an even number of pairs where $x > y$.

We have split all the pairs with $x > y$ into three even-sized sets (we have consider either only bits more significant that the position $p$ or the bits up to position $p$ or the less significant bits, too). Therefore the total number of such pairs is even.

This proof uses only the fact that two blocks have two common variable coordinates, so it is possible to illustrate this split with three-dimensional blocks in five-dimensional case. Consider the pair of blocks $0*0*1$. The pairs of codes of the first type satisfy the masks $0*1*0$ and $0*0*1$, it is clear there are $32 - 16 = 48$ such pairs and the first 32 of them provide an inversion as the upper code is less than the lower code. The pairs of the second type satisfy the masks $001*1$ and $000*1$, the
first 4 of them do not provide an inversion, but the second 4 of them do. The pairs of the third type are defined by masks \(001^*1\) and \(000^*0\), where inversions are provided only by the pairs \(00101\) and \(00010\).

As promised, the number of inversions is even.

This finishes the proof that the number of inversions containing two vertices in the two given blocks is even.

**Summarizing the inversion counts**

We have checked that for each block the number of inversions inside the block is zero, and for each pair of blocks the number of inversions between the vertices in these two blocks is even. Therefore the total number of inversions is even and the permutation is even.

This is applicable to both Before and After permutations, so the single increment step, After \(\circ\) Before\(^{-1}\) is an even permutation and therefore not an even-length cycle.

This proves the \(F(n) + 1 = \lfloor \frac{n}{2} \rfloor\) lower bound for the number of bits read during the worst increment step in the Arthur-Merlin setting.

**Acknowledgements**

I am grateful to Gerth Brodal for attracting attention to this problem. I am extremely grateful to Gudmund Frandsen for a lot of useful discussions and the efforts he has spent on reading the draft versions of this proof.

**References**

[BGPS2014] Gerth Stølting Brodal, Mark Greve, Vineet Pandey, Srinivasa Rao Satti. Integer representations towards efficient counting in the bit probe model. *J. Discrete Algorithms* 26: 34-44 (2014)