Trans-dichotomous algorithms without multiplication - some upper and lower bounds

Andrej Brodnik[1]⋆. Peter Bro Miltersen[2]⋆⋆ J. Ian Munro[3]⋆⋆⋆

[1] Institute of Mathematics, Physics and Mechanics, Ljubljana, Slovenia and Luleå University, Luleå, Sweden. Email Andrej.Brodnik@IMFM.Uni-Lj.SI.
[2] BRICS, Centre of the Danish National Research Foundation, University of Aarhus, Denmark. Email bromille@brics.dk.
[3] Department of Computer Science, University of Waterloo, Canada. Email imunro@uwaterloo.ca.

Abstract. We show that on a RAM with addition, subtraction, bitwise Boolean operations and shifts, but no multiplication, there is a trans-dichotomous solution to the static dictionary problem using linear space and with query time $\sqrt{\log n (\log\log n)^{1+o(1)}}$. On the way, we show that two w-bit words can be multiplied in time $(\log w)^{1+o(1)}$ and that time $\Omega(\log w)$ is necessary, and that $\Theta(\log\log w)$ time is necessary and sufficient for identifying the least significant set bit of a word.

1 Introduction

Consider a problem (like sorting or searching) whose instances consists of collections of members of the universe $U = \{0,1\}^w$ of w-bit bit strings (or numbers between 0 and $2^w - 1$). An increasingly popular theoretical model for studying such problems is the *trans-dichotomous* model of computation [13, 14, 1, 7, 8, 3, 2, 20, 18, 9, 4, 21, 6], where one assumes a random access machine where each register is capable of holding exactly one element of the universe, i.e. we assume that the word size of the machine matches the "word size" of the universe. We can operate on the registers using at least a *basic instruction set* consisting of: Direct and indirect addressing, conditional jump, and a number of computational instructions, including addition, subtraction, bitwise Boolean operations and left and right shifts. All operations are unit cost. An important ingredient of many trans-dichotomous algorithms is exploiting the unit cost assumption by doing parallel operations on the word level. It is well known that such "tricks" often speed programs up in practice, and we can view trans-dichotomous algorithms as theory's contribution to the understanding of this phenomenon.

In this paper, we consider trans-dichotomous algorithms for the static dictionary problem: Given a set of keys $S \subseteq \{0,1\}^w$, construct a data structure using $O(n)$ registers, with $n = |S|$, so that membership queries "Is x in S?" can be answered efficiently, and, if $x \in S$, some information associated with x can be retrived.

⋆ Some of the results of this paper appeared in the PhD thesis of this author

⋆⋆ Supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT). Part of this work was done while the author was at the University of Toronto.

⋆⋆⋆ This research was supported by a grant from the Natural Science and Engineering Council of Canada.

It is well known that if the computational instructions also include multiplication, the static dictionary problem can be solved using only $O(1)$ query time [12, 11] and even in space within an additive low order term of the information theoretic minimum [8]. However, in many architectures encountered in practice, multiplication is more expensive to perform than the basic operations mentioned above, so it seems natural to disallow it as a unit cost operation in our theoretical model. Indeed, this approach is taken in [2, 18, 21], and we shall also take it here.

Andersson *et al* [4] show that if only the basic instruction set is used, worst case query time $\Omega(\sqrt{\log n/\log\log n})$ is necessary. This lower bound is also valid under extended computational instruction sets, as long as all instructions can be computed by AC^0 circuits (multiplication can not [15]). Andersson *et al* show that a matching upper bound $O(\sqrt{\log n/\log\log n})$ can be obtained, if various exotic AC^0 instructions (*clustering functions* and *cluster busters*) are allowed.

One of the main results of the present paper is that with only the basic instruction set of addition, subtraction, bitwise Boolean operations, and shifts, worst case query time $\sqrt{\log n(\log\log n)^{1+o(1)}}$ is possible. This leaves only a $(\log\log n)^{1+o(1)}$ gap to the lower bound.

The technique is basically a variation of the technique of Andersson *et al* [4], combining range reduction with packed B-trees. In order to carry this approach through in the setting of the basic instruction set, we introduce some techniques and subroutines which may be useful for trans-dichotomous computation in general, namely *bit permutation*, *circuit evaluation*, and *locating the least significant bit of a word*. We also show lower bounds for these problems, in some cases establishing optimality of our subroutines. Thus, we note that any fixed permutation of the bits of a word can be realized in time $O(\log w)$ and that reversing a word requires time $\Omega(\log w)$. We show that multiplying two words can be done in time $(\log w)^{1+o(1)}$ and that $\Omega(\log w)$ is a lower bound. We show that finding the least significant 1-bit in a word can be done in time $O(\log\log w)$ and that this is optimal. The lower bounds hold, even if a precomputed table of size $O(2^{w^\epsilon})$ for a small constant $\epsilon > 0$ is allowed. Our lower bound technique is essentially an extension of the *locality*-technique of [17].

The existence of a static dictionary with sublogarithmic query time on a RAM with the basic instruction set disproves a conjecture of the second author [17]. It is, however, *not* a practical solution, and is intended only to demonstrate an asymptotic upper bound; having disallowed unit cost multiplication from the instruction set, we base our upper bound on reintroducing it as a subroutine on subwords: An essential part of the algorithm is the parallel execution of several instances of Schönhage and Strassen's multiplication algorithm on parts of a word using word level parallelism.

The paper is structured as follows: in Section 2, we construct our general subroutines. In Section 3, we show how to use them to solve the static dictionary problem, and in Section 4, we show the lower bounds.

Notation

When x and y are bit strings of equal length, we denote by x AND y, x OR y, x NAND y, NOT x, and x XOR y the bitwise Boolean operations on x and y. $x[i]$ denotes the i'th bit of x from the left. Words are considered bit strings of length w. Note that when a word x is interpreted as an unsigned integer, the least significant bit is $x[w]$ and the most significant bit is $x[1]$. This might be slightly confusing, but most often we view words as strings, rather than integers. If $I = [i, j] = \{i, i+1, \ldots, j\}$ is an interval of bits, $x[I]$ denotes $x[i]x[i+1] \ldots x[j]$. When x is a bit string, and i a positive integer, we denote by $x \uparrow i$ the result of shifting x i positions to the left (padding the rightmost part of the result with zeros, and killing off the i leftmost positions of x). Similarly, $x \downarrow i$ denotes shifting x i positions to the right (padding the leftmost part of the result with zeros, and killing off the i rightmost positions of x). If i is negative, $x \uparrow i = x \downarrow -i$ and vice versa. We denote by w the word size of the machine. We shall assume that w is a power of two. We will assume that bit strings are stored packed, i.e. a bit string of length m is stored in $\lceil m/w \rceil$ machine words. Note that if we want to do the above mentioned operations on strings of length m, we can do so in time $O(\lceil m/w \rceil)$ by using the basic bit manipulation instructions of our machine.

2 Useful subroutines

Permuting and circuit evaluation

Proposition 1. *Consider a fixed permutation π on m symbols. Given a bit vector $x[1]x[2] \ldots x[m]$, we can compute its permutation*

$$\mathrm{perm}_\pi(x) = x[\pi(1)]x[\pi(2)]x[\pi(3)] \ldots x[\pi(m)]$$

in time $O(\lceil m/w \rceil \log m)$.

Proof. Assume without loss of generality that m is a power of two. It is well known that a *butterfly network* with m sources and m sinks is a permutation network for m packages. Given the permutation π, in this graph we can find edge disjoint paths from source $\pi(i)$ to sink i. Given an input $x \in \{0,1\}^m$, mark all nodes along the path from $\pi(i)$ to i with label $x[\pi(i)]$. Now, each column in the graph is marked with a bit vector in $\{0,1\}^m$. The first column is marked with the input, the last column with the desired output. It is easily checked that given the mark of one column, we can compute the mark of the next in time $O(\lceil m/w \rceil)$, using only shifts and bitwise Boolean operations (bitwise AND to mask out bits, and bitwise OR to combine masks).

Proposition 2. *Let m, m' be given, and let $i_1 + i_2 + \cdots + i_m = m'$. Consider a map* expand $: \{0,1\}^m \to \{0,1\}^{m'}$*, defined by*

$$\mathrm{expand}(x[1]x[2] \ldots x[m]) = x[1]^{i_1} x[2]^{i_2} \ldots x[m]^{i_m}.$$

Then, expand(x) *can be computed in time $O(\lceil (m+m')/w \rceil \log(m+m'))$.*

Proof. For convenience of notation, we assume that $i_j > 0$ for all j and hence $m' \geq m$; the general case is similar. We also assume that m is even. First compute, using Proposition 1, $x' = \mathrm{perm}_\pi(x0^{m'-m}) = 0^{i_1-1}x[1]0^{i_2-1}x[2]\ldots 0^{i_m-1}x[m]$ in time $O(\lceil m'/w \rceil \log m')$. Let $y = 1^{i_1}0^{i_2}1^{i_3}\ldots 1^{i_{m-1}}0^{i_m}$. Now, compute

$z_1 = y$ AND NOT$[(y$ AND $x') + y)]$,
$z_2 =$ (NOT y) AND NOT$[(($NOT $y)$ AND $x') +$ (NOT $y)]$,
$z = z_1$ OR z_2

Then z is $x[1]^{i_1}x[2]^{i_2}\ldots x[m]^{i_m}$, as desired.

Proposition 3. *Let m', m be given. A monotone projection $f : \{0,1\}^m \to \{0,1\}^{m'}$ is a map of the form $f(x)[i] = \rho(i)$, with $\rho(i) \in \{0, 1, x[1], x[2], \ldots, x[m]\}$. Any monotone projection can be computed in time $O(\lceil (m'+m)/w \rceil \log(m'+m))$.*

Proof. Write the projection as a composition of an expansion, a permutation, and bitwise Boolean operations with masks, and use Propositions 1 and 2.

Lemma 4. *Let C be a Boolean circuit of depth d, fan-in 2, and size $s \geq m, m'$, computing a map $\{0,1\}^m \to \{0,1\}^{m'}$. Given x, $C(x)$ can be computed in time $O(\lceil s/w \rceil d \log s)$.*

Proof. Convert C to C' of the following normal form:

- All negations of C' occur at inputs.
- C' is constructed on levels $0..2d+1$, with and-gates on even levels and or-gates on odd levels. The inputs to gates at level l are either constants or outputs of gates at level $l-1$. The inputs and negations to inputs are regarded as level 0 gates, and the outputs as level $2d+1$ gates. There are exactly s gates on each level, except level 0, with $2m$ gates, and level $2d+1$, with m' gates.

Note that the connection between two levels can be described as a monotone projection. Given input $x \in \{0,1\}^m$, we can now compute $C(x)$ by first computing the concatenation of x and NOT x and then computing d blocks of operations, each consisting of a monotone projection, a bitwise Boolean OR, a monotone projection, and a bitwise Boolean AND. Finally, we compute a last projection. Total time is $O(\lceil s/w \rceil d \log s)$.

Corollary 5. *Multiplying two w-bit words can be done in time $(\log w)^{3+o(1)}$.*

Proof. Apply Lemma 4 to Schönhage and Strassen's multiplication circuit [19] of size $w(\log w)(\log\log w)$ and depth $(\log w)(\log\log w)$.

Since multiplication is of primary importance, we want to optimize the above corollary a bit. Using the general circuit simulation algorithm is a bit of an overkill. Schönhage and Strassen have two multiplication circuits, one of size $w(\log w)(\log\log w)(\log\log\log w)\ldots$, based on the Fourier transform over the complex numbers, and one of size $w(\log w)(\log\log w)$, based on the Fourier transform over finite rings. By examining the first algorithm in details (which we won't go into here) one finds that it *word parallelizes* perfectly, i.e. it can be implemented using bitwise Boolean operations and shifts (and no additions) with only a constant factor of overhead, i.e. we get the following proposition:

Proposition 6. *Multiplying two w-bit words can be done in time* $O((\log w)(\log\log w)(\log\log\log w)\ldots)$.

The second multiplication circuit does *not* seem to word parallelize perfectly, the obstacle being that different parts of a word must be shifted by different amounts, so we don't know if we can improve this to $O((\log w)(\log\log w))$.

We need Fact 6 to get the $\sqrt{\log n(\log\log n)^{1+o(1)}}$ bound for the static dictionary problem in the next section. However, readers who prefer a presentation without gaps can use Fact 5 instead and still get a $\sqrt{\log n(\log\log n)^{O(1)}}$ solution.

Note that all of the above upper algorithms use various word-sized constants containing masks depending on the word size w. We cannot afford to compute these constants at run-time, so we assume that they have been determined and hard-wired into the algorithm at compile-time. Following the terminology of Ben-Amram and Galil [5], such solutions are called *weakly non-uniform*. Fredman and Willard's fusion tree [13] is another example of a weakly non-uniform algorithm. We do not consider weak non-uniformity as a serious deficiency of the algorithms; computing useful constants at compile-time is hardly an esoteric phenomenon. Note that in our solution to the static dictionary where the subroutines will be used, we can eliminate the non-uniformity by simply making the constants part of the static data structure instead of the query program.

Simulating circuits in trans-dichotomous algorithms is not a new idea; indeed, the word merging algorithm of Albers and Hagerup [1] which is often used in trans-dichotomous algorithms is essentially a simulation of Batcher's bitonic merging network. Thorup, in his paper on sorting with the basic instruction set [21], shows a different simulation result:

Theorem 7 (Thorup). *Given a Boolean circuit C, mapping w bits to w bits. Suppose w instances x_1, x_2, ..., x_w are given. The sequence $C(x_1), C(x_2), \ldots, C(x_w)$ can be computed in time $O(s + \log w)$.*

Thus, by applying "mass production" and performing several evaluations simultaneously, Thorup avoids the multiplicative $d \log s$ penalty we have to pay. However, for our purposes, we have to consider the evaluation of a single instance.

The rightmost set bit of a word

Consider the function RightmostOne, giving the position of the rightmost set bit of a word. For example, RightmostOne(0010001010000000) = 9.

Lemma 8. RightmostOne(x) *can be computed in time* $O(\log\log w)$

Proof. 1. Given a word x with least significant 1-bit in position i. Let 1^w be the all-1 string and consider the two words $x + 1^w$ and x XOR 1^w. It is easily seen that they are different on bits $1, 2, \ldots, i$, but the same on bits $i+1, i+2, \ldots, w$. Thus, if we let $y = (x + 1^w)$ XOR $(x$ XOR $1^w)$ and let $z = y$ XOR $(y \uparrow 1)$, z will be the word with a 1 in position i and 0 elsewhere.

2. Do a binary search for the single 1 bit in z. Stop after $\log\log w$ steps. We now have an integer r, so that we know the 1-bit is in the subword $x' = x[r \ldots r + s - 1]$ with $s = \lceil w/\log w \rceil$.

3. Move x' to the leftmost end of the word, i.e. perform $x := x \uparrow (r-1)$ Note that we now have a word where the least significant bit is between 1 and s. If we find the position and add $r-1$, we are done.
4. Make a bit string x'' containing $j = \lceil \log s \rceil$ consecutive copies of x'. This can be done in time $O(\log s) = O(\log \log w)$ [2]. The bit string x'' can be stored in ≤ 2 words, so we can operate on it with unit cost.
5. To avoid cumbersome notation, the next step is most easily explained by example. Suppose that $w = 64$, then $s = 11$ and $j = 4$. Suppose $x' = 00001000000$. We want to return 5. We have
   $x'' = 00001000000|00001000000|00001000000|00001000000$.
   Let p be the constant bit string
   $p = 00000001111|00011110000|01100110011|10101010101$.
   In general, each block of p contains a sequence of alternating blocks of 0's and 1's, in the i'th block from the right, the blocks have size 2^i. The first 0 of each block is chopped off. Let $g = x''$ AND p. Then
   $g = 00000000000|00001000000|00000000000|00001000000$.
   Now, the i'th block from the right of g contains a 1, if and only if the i'th least significant bit of the answer is a 1. We want to move these 1's to specified positions.
6. Let $h = g + a$, where
   $a = 01111111111|01111111111|01111111111|01111111111$.
   Now, for $i = 0 \ldots j-1$, position $is+1$ of h is a 1 if and only if the i'th most significant bit of the desired answer is a 1. But using time $O(\log \log w)$ time, we can move position $is+1$ to position $w-(j-1)+i$ for all i, and we are done.

We shall actually use the following slight generalisation, which is a corollary of the proof:

Corollary 9. *Let* RightmostField(x, d) *be the function which looks at bits* $1, d+1, 2d+1, \ldots,$ *in the word* x*, and returns the largest* i*, for which bit* id *is* 1. RightmostField(x, d) *can be computed in time* $O(\log \log(w/d))$.

Remarks: Gerth Brodal (personal communication) has noted that a slight modification of the above algorithm can also be used to find the *most* significant set bit of a word. When multiplication is allowed, both operations can be done in constant time [13, 7, 3].

3 The static dictionary problem

We shall use the subroutine $u \leftarrow$ InterleavedMult(x, y, l, m). Here, l and m are positive integers, and x and y are two bit strings (actually single words) of length at most $lm \leq w$; x and y are interpreted as consisting of the binary notation of m positive integers, each containing at most l bits, stored in *interleaved* fashion (see figure 1). The subroutine returns all of the m pairwise products stored in a bit string of length $2lm$. Using the general circuit simulation lemma, we get

Lemma 10. InterleavedMult(x, y, l, m) *can be computed in time* $O((\log l)^{3+o(1)})$.

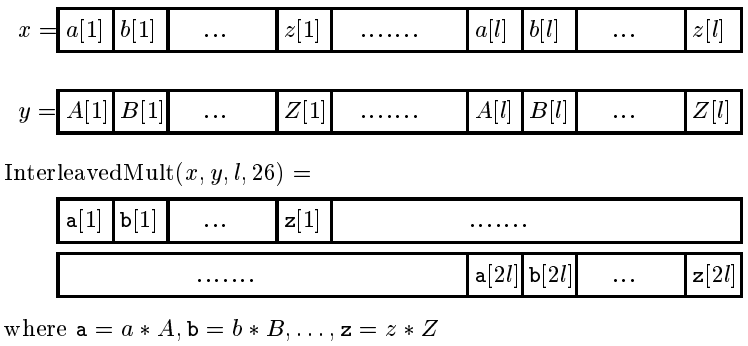

Fig. 1. The InterleavedMult function

Readers preferring a presentation without gaps can use this bound in Theorem 13 and obtain a slightly weaker bound than stated there. For the best bound we know, we need

Lemma 11. InterleavedMult(x, y, l, m) *can be computed in time* $O((\log l)^{1+o(1)})$.

Proof. We use Fact 6. Since the algorithm proving this fact can be implemented without additions, we can execute m instances of the algorithm, with the instances given in interleaved form, without any overhead.

Lemma 12. *Let* $S \subseteq \{0,1\}^b$ *be a set of* b*-bit keys of size* n *with* $n \geq 4\lceil \log n \rceil^2$. *There is an interval* $I \subseteq \{0, \ldots, b-1\}$ *of size at most* $b/\log n$, *so that for some* $a \in \{0,1\}^b$, $f_a(x) = \text{InterleavedMult}(x, a, 4\lceil \log n \rceil^2, \lceil b/4\lceil \log n \rceil^2 \rceil)[I]$ *defines a 1-1 function on* S. *(Recall that* $x[I]$ *selects the bits marked by* I *from* x*).*

Proof. We shall use the following fact, due to Dietzfelbinger *et al* [11]: The class of hash functions $\mathcal{H}_{k,l} = \{h_a : \{0, \ldots, 2^k - 1\} \to \{0, \ldots, 2^l - 1\} | 0 < a < 2^k \text{ and } a \text{ odd}\}$ given by $h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}$ is *nearly universal*, i.e. for fixed x and y with $0 \leq x, y < 2^k$, if a is chosen at random, then $\Pr[h_a(x) = h_a(y)] \leq 2/2^l$. A value $a \in \{0,1\}^b$ can be interpreted as $\lceil b/4\lceil \log n \rceil^2 \rceil$ $4\lceil \log n \rceil^2$-bit words stored in interleaved fashion. The InterleavedMult function multiplies each of these with the corresponding segment in x. If each of these functions are injective on S (restricted to the corresponding segment), the entire function will be injective. In general, if we are given a set S of size n taken from some universe, and a family of nearly universal hash functions mapping the universe to a set of size n^2, some member of the family will be injective on S [10]. Thus, we have that for each segment, there is a value of the restriction of a to that segment, so that when the multiplication is performed, and a certain sub-segment of length $2\lceil \log n \rceil$ of the result is extracted, this will be an injective function on S, restricted to the corresponding segment. But since we store the segments in interleaved fashion, the union of all the subsegments is an interval of size

$$2\lceil \log n \rceil \lceil b/4\lceil \log n \rceil^2 \rceil \leq 2\lceil \log n \rceil \left(\frac{b}{4\lceil \log n \rceil^2} + 1\right) \leq \frac{b}{2\log n} + 2\lceil \log n \rceil \leq b/\log n.$$

This is the interval I, and the proof of the lemma is complete.

Note why we multiply segments stored in interleaved fashion, rather than concatenated fashion: we could also have implemented a BlockMult, multiplying a number of continuous blocks with the same time bound, but we would have no way of collecting the various subsegments from each block into a single interval within a sufficiently small time bound; this can be shown using our lower bounds techniques of Section 4. However, Thorup [21] has found an application for such a BlockMult subroutine.

We are now ready for the main theorem of this section.

Theorem 13. *There is a solution to the static dictionary problem for sets $S \subseteq \{0,1\}^w$ of size n using $n + o(n)$ memory cells, each containing w bits and with worst case query time $\sqrt{\log n (\log\log n)^{1+o(1)}}$ with a query algorithm only using the basic instruction set.*

Proof. Assume first that $w \geq 4\lceil \log n \rceil^2$. Using Lemma 12 with $b = w$, we choose a word a such that f_a is 1-1 on A. We store a in our data structure and use f_a to hash our elements, thereby reducing them to size $b' \leq w/\log n$. By Lemma 11, computing the hash function on a single element requires time $(\log\log n)^{1+o(1)}$. Now, let $b = b'$ and repeat. In general, after t iterations, we have reduced our keys to word size less than $\leq w/(\log n)^t$, *unless* we arrive at a key size which is less than $4\lceil \log n \rceil^2$. If we do arrive at such a word size, we stop. Otherwise we continue for $t = \sqrt{\log n / \log\log n}$ iterations. In the first case we have found a sequence of hash functions whose composition is an injective function mapping S to $\{0,1\}^{4\lceil \log n \rceil^2}$. In the latter case, we have found a sequence of hash functions whose composition is an injective function mapping S to the domain $\{0,1\}^{\lfloor w/(\log n)^t \rfloor}$.

In the first case, we store the reduced elements in a two level hash table, following Fredman, Komlos, and Szemeredi [12], except that we use the hash function of Dietzfelbinger *et al* (the analysis of [12] goes through for any nearly universal family). The multiplicative hash function can be evaluated using lemma 6, using time $(\log\log n)^{1+o(1)}$. The search time is $O((t+1)(\log\log n)^{1+o(1)}) \leq \sqrt{\log n (\log\log n)^{1+o(1)}}$, as desired.

In the second case, we store the reduced elements in a packed B-tree of degree $D \geq \lfloor (\log n)^t \rfloor$, following Andersson [2]. In Andersson's paper, it was shown how to search a packed B-tree in time $O(\log_D n)$ using the basic RAM instruction set. However, he had to use a huge table in order to determine the rank of a given element in a B-tree node and we want linear space. Examining his algorithm, we find that the obstacle is precisely the computation of the RightmostField function, or, to be completely precise, the analogous LeftmostField function. By storing each B-tree node in reverse sorted order, we can use the RightmostField subroutine instead, and search the B-tree in time $O((\log\log D)(\log_D n))$.

We now clearly have a data structure of size $n + o(n)$. The search time is

$$t(\log\log n)^{1+o(1)} + O(\frac{\log n}{\log((\log n)^t)}(\log\log((\log n)^t))) = \sqrt{\log n (\log\log n)^{1+o(1)}},$$

as desired.

Remark: Combining the above techniques with techniques of the first and third author [8], it is possible to improve the space bound in Theorem 13 to $B + o(B)$ bits, where B is the information theoretic minimum $\log \binom{2^w}{n}$, in the case where we only want to test membership and not retrieve associated information.

4 Lower bounds

In this section, we show lower bounds for some of our subroutines; namely reversing a word, multiplying two words, and locating the least (or most) significant bit of a word, using the basic instruction set. As mentioned in the introduction, a lower bound for the static dictionary problem *itself* is shown in [4]. Note that since the problems only involve $O(1)$ words, they can be solved in constant time if a precomputed array containing a *table* of the function is allowed, but this table will be very large. Our lower bounds hold even in the presence of precomputed tables, as long as the precomputed table has size less than 2^{w^ϵ} for certain constants $\epsilon > 0$.

For the lower bounds, we need to specify and simplify our model slightly: we have a RAM with a certain fixed number of CPU-registers, say registers A, ..., Z, and a random access memory which can be accessed from the CPU using direct and indirect reads and writes. We have conditional jump (if A $= 0$ then branch) and the following computational instructions operating on CPU-registers: $+$, NAND (this is sufficient to simulate all the bitwise Boolean operations) and $\uparrow$.

As an aid in showing our lower bounds, we shall consider non-Boolean *word circuits* computing on words. Each wire of such a circuit C holds a word $x \in W = \{0,1\}^w$. It may contain three kinds of gates: $+$-gates, NAND-gates, and $\uparrow_r$-gates for various constants r, with $\uparrow_r(x) = x \uparrow r$. Arbitrary constants $c \in W$ may also be fed into the circuit. The size of a circuit is the number of gates it contains. A circuit with one source and one sink computes a function $W \to W$ in the natural way.

The following general lemma is the key to all lower bounds.

Lemma 14. *Let C be a word circuit of size s. Let $u \in W$ be a word with a single block of ones in the left end, i.e. $u = 1^k 0^{w-k}$ for some k.*

Then, there is a bit string v with the following properties:

- *v has length $3w$ and will be indexed $v[-w] \ldots v[-1]v[1]v[2] \ldots v[2w]$, so when we do a bitwise* AND *with v and a word, only the middle part of v will matter.*
- *v has Hamming weight at most $2^s k$*
- *For any i between 0 and $w-k$, if x varies, but the value of x* AND *$(v \downarrow i)$ is fixed (i.e. we only vary x on the bits where $v \downarrow i$ is 0), $C(x)$* AND *$(u \downarrow i)$ takes on at most $2^{2^{2s}}$ values.*

Proof. The proof is an induction in s. For $s = 0$ (i.e. $C(x) = x$ or $C(x) = c$), the claim clearly holds, if we let $v = u$, padded with 0's.

Suppose the top gate of C is a $+$-gate, i.e. $C(x) = C_1(x) + C_2(x)$, where C_1 and C_2 are smaller circuits than C. By induction, let v_1 and v_2 be given, and let $v = v_1$ OR v_2. Now, if the bits of x AND $(v \downarrow i)$ are fixed, $C_1(x)$ AND $(u \downarrow i)$ and

$C_2(x)$ AND $(u \downarrow i)$ each take on at most $2^{2^{2(s-1)}}$ different values. Furthermore, since the word $u \downarrow i$ is a word with only a single interval of set bits, we have that if $C_1(x)$ AND $(u \downarrow i)$ and $C_2(x)$ AND $(u \downarrow i)$ are fixed, $C(x)$ AND $(u \downarrow i)$ take on at most 2 different values, corresponding to whether a carry bit is propagated to the interval or not. Thus, $C(x)$ AND $(u \downarrow i)$ takes on at most $2^{2^{2s-2}} \cdot 2^{2^{2s-2}} \cdot 2 \leq 2^{2^{2s}}$ different values when the bits of x AND $(v \downarrow i)$ are fixed.

The case where the top gate of C is a NAND-gate is handled similarly to the +-gate case.

Now suppose the top gate of C is an $\uparrow_r$ gate with input C_1. By induction, let v_1 be given, and let $v = v_1 \downarrow r$. If x AND $(v \downarrow i) = x$ AND $(v_1 \downarrow (i + r))$ is fixed, $C_1(x)$ AND $(u \downarrow (i + r))$ takes on at most $2^{2^{(s-1)}}$ values, and $C_1(x)$ AND $(u \downarrow (i + r))$ determines $C(x)$ AND $(u \downarrow i)$.

Theorem 15. *Any RAM program computing the rightmost (or leftmost) set bit of a word uses time $\Omega(\log \log w)$. This is true, even if a precomputed table of size $O(2^{w^{1-\epsilon}})$ is allowed, for any constant $\epsilon > 0$.*

Proof. Let w be sufficiently large. Let $e_i = 0^{i-1}10^{w-i}$. Let $U = \{e_1, e_2, \ldots, e_w\}$, i.e., the set of words with Hamming weight one. Given e_i in U, a least significant bit algorithm returns i. Suppose an algorithm exists which gives this answer in less than $(\log \log w)/10$ steps. We shall show that it gives the same answer on $i, j \in U$ with $i \neq j$ and thus cannot be correct.

Consider executing the algorithm on each $x \in U$. After t steps, the random access machine is in some configuration $K(x, t)$.

We shall show that for any $t < \log \log w/10$, we can find a subset $U_t \subseteq U$ and word circuits $A_1, C_1, A_2, C_2, \ldots, A_t, C_t, D_{\mathrm{A}}, \ldots, D_{\mathrm{Z}}$, so that

- $|U_t| \geq |U|/2^{2^{4t}}$,
- the size of each A_i, C_i, and D_i is at most t.
- For all $x \in U_t$, in all configurations $K(x, t)$, the program counter is at the same location, and furthermore, the memory has the same appearance as originally, except that in memory register $A_i(x)$, the new value $C_i(x)$ can be found. If for more than one value $i_1 < i_2 < \cdots < i_r$, we have $A_{i_1}(x) = A_{i_2}(x) = \cdots = A_{i_r}(x)$, the value of the register is $C_{i_r}(x)$, i.e. the largest index "wins". The value of CPU-register i is $D_i(x)$.

Let us first see that if we can show that this invariant can be upheld for any algorithm, we get the theorem. We can assume, without loss of generality, that the value is returned in CPU-register A, i.e. for members of U_T, with $T = (\log \log w)/10$, the returned value is $D_{\mathrm{A}}(x)$. Note that all bits of the result are zero, except the least significant $\log w$. According to Lemma 14, there is a bit string v of Hamming weight at most $(\log w)^2$, so that if the bits of v are all zero, the result is member of a set of size $2^{2^{\log \log w/5}}$. This means that for $x \in U_T - B$, for a certain set B of size $(\log w)^2$, only $2^{2^{\log \log w/5}}$ different answers are obtained. Since $U_T - B \gg 2^{2^{\log \log w/5}}$, two elements return the same answer and the algorithm is incorrect.

We should now check that the invariant can be upheld. The cases of direct and indirect write, conditional jump, and computation using NAND and + are all straightforward. The interesting cases are *read* and *computation using* $\uparrow$. Suppose we do an indirect read, e.g., let CPU register A be equal to memory cell u, where u is the value of CPU register B. The value of u for members $x \in U_t$ is described by a word circuit $D_{\mathrm{B}}(x)$. For each member x of U_t, it is either the case that $D_{\mathrm{B}}(x)$ is the address of one of the undisturbed values in the precomputed table, it is one of the previously written addresses $A_1(x), \ldots, A_t(x)$, or it is neither. Thus, we get the set U_t partitioned into $t+2$ subset. Take the largest of these, V. If V corresponds to one of the $t+1$ last cases, we can let $U_{t+1} = V$ and easily uphold the invariant. In the first case, we read some precomputed value. Only the $w^{1-\epsilon}$ least significant bits of the address are non-zero. By Lemma 14, there is a bit string v of Hamming weight at most $(\log w)w^{1-\epsilon}$, so that if x AND $v = 0$, the result is member of a set of size $2^{2^{wt}}$. This means that for $x \in V - B$, for a certain set B of size $(\log w)w^{1-\epsilon}$, only $2^{2^{2t}}$ different addresses are read. We get $V - B$ partitioned into $2^{2^{2t}}$ subsets. Let U_{t+1} be the largest of these. The invariant can now easily be upheld.

The case of a computation using $\uparrow$ is very similar. The reason that $\uparrow$ is more difficult to handle than the other computational instructions is that $\uparrow$ is a binary predicate, but in our word circuits, we only allow unary $\uparrow_r$- gates with fixed second argument r. But we can handle $\uparrow$ instructions similarly to reads, by noting that only the $\log w+1$ least significant bits of the second argument are relevant. By Lemma 14, we find a large subset of U_t for which the second argument is the same and we are done.

The lower bound for reverse follows a slightly different strategy. We consider reversing a subword of length $b \leq w$; $\mathrm{reverse}(x[1]x[2]\ldots x[b]) = x[b]\ldots x[2]x[1]$.

Lemma 16. *Let $b \leq w$. A word circuit of size $\leq \frac{1}{10}\log b$ correctly computes reverse on at most a $2^{-b^{1/4}}$ fraction of $\{0,1\}^b$.*

Proof. Let $k = \lfloor b^{1/3} \rfloor$. Let a word circuit C be given. By Lemma 14, we can find a word v of Hamming weight at most $b^{1/10}b^{1/3}$ so that the statement of the lemma holds for $u = 1^k 0^{w-k}$, i.e. if x AND $(v \downarrow i)$ is fixed, $C(x)$ AND $(u \downarrow i)$ takes on at most $2^{b^{1/5}}$ different values.

Let $T = \{0, 1, 2, \ldots, b - \lfloor b^{1/3} \rfloor - 1\}$. We claim that for some $i \in T$, $v \downarrow i$ and $\mathrm{reverse}(u \downarrow i)$ are *disjoint*, i.e. they do not contain set bits in overlapping positions. Note that $v \downarrow i$ and $\mathrm{reverse}(u \downarrow i)$ are disjoint if $b - u_1 - i \neq v_1 + i$ for all indices u_1, v_1 so that $u[u_1] = v[v_1] = 1$. This is equivalent to $i \neq (b - u_1 - v_1)/2$, i.e. for any particular (u_1, v_1) pair, at most one i goes wrong. Since there are only $|u|_h|v|_h \leq b^{1/3}b^{1/10}b^{1/3} < \#T$ pairs, some $i \in T$ is good for all pairs.

Now fix the bits of x which are 1 in $v \downarrow i$ to any value and select all other bits at random. We want to find the probability that $C(x) = \mathrm{reverse}(x)$. We know that $C(x)$ AND $(u \downarrow i)$ takes on at most $2^{b^{1/5}}$ different values. Note that $\mathrm{reverse}(x)$ AND $(u \downarrow i)$ is determined 1-1 by x AND $\mathrm{reverse}(u \downarrow i)$, and

since reverse($u \downarrow i$) is disjoint from the set of fixed bits of x, all $2^{\lfloor b^{1/3} \rfloor}$ possible values of reverse(x) AND ($u \downarrow i$) are equally likely. The probability that reverse(x) AND ($u \downarrow i$) has one of the different possible values of $C(x)$ AND ($u \downarrow i$) is thus at most $2^{b^{1/5}}/2^{\lfloor b^{1/3} \rfloor} < 2^{-\lfloor b^{1/3} \rfloor/2} < 2^{-b^{1/4}}$. This is also an upper bound on the probability that the circuit is correct for random x (since the bits of x, marked by $v \downarrow i$ were fixed arbitrarily).

Theorem 17. *Computing reverse of a b-bit string, $(\log w)^{10} \leq b \leq w$, requires time $\Omega(\log b)$, even when a precomputed table of size $2^{b^{1/10}}$ is given.*

Proof. The strategy is to find a small set of word circuits, so that for a non-negligible subset of the possible inputs, every value found in the random access memory is the value of one of the circuits on the input. Unlike the lower bound proof for the least significant bit problem, we only have to argue about the set of values in the random access memory, not where they are located.

We will show: For any $t > 0$, there is a subset of U_t of $U = \{0,1\}^b$ and an (unordered) set of word circuits $\mathcal{C}_t$ so that

- $\#\mathcal{C}_t \leq 2^{b^{1/10}} + t + 1$
- For each $C \in \mathcal{C}_t$, the size of C is at most t.
- $\#U_t \geq 2^b/(\#\mathcal{C}_t)^{2t}$.
- For any $x \in U_t$, if we run the algorithm on x for t steps, the set of non-zero values in the memory is a subset of $\mathcal{C}_t(x)$. Furthermore, the program counter points to the same location in the program for all $x \in U_t$.

We first show why this implies our theorem. After running $T = \frac{1}{10}\log b$ steps, we find slightly more than $2^{b^{1/10}}$ word circuits of size at most T and a set U_T of size at least $2^{b-o(b)}$. Each of these circuits are correct on at most a $2^{-b^{1/4}}$ fraction of U. Therefore on at most a $(\#\mathcal{C}_T) \cdot 2^{-b^{1/4}}$ fraction of $x \in U$ will the value of even one of the circuits be reverse(x). Since U_T forms a greater fraction of U than that, we have for some x in U_T, the value reverse(x) is not found in any memory location. Therefore the program is not correct.

The proof that we can maintain the invariant is similar to the proof for the least significant bit, only simpler. The proof is an induction in t. For $t = 0$ the lemma clearly holds, we just let the initial C_i's be trivial circuits containing the constants of the precomputed table, except for one circuit, the identity circuit mapping x to x. Also, $U_0 = U$.

A conditional jump divides U_t in two sets, of which we let U_{t+1} be the larger one. A direct or indirect read or write does not change the set of values. A computation combines two previously computed values. For addition and bitwise NAND, for each $x \in U_t$, the two arguments are described by two circuits from the set $\mathcal{C}_t$. Thus, U_t is partitioned into $(\#\mathcal{C}_t)^2$ subsets, of which we let U_{t+1} be the larger. For $\uparrow$, take the most commonly occurring circuit appearing as first argument, and the most commonly occurring integer between 0 and $w-1$ appearing as second argument, given the first argument. This reduces the set with a factor at most $w(\#\mathcal{C}_t)$. Thus, the invariant can be upheld and we are done.

The lower bound for multiplication is most conveniently shown by a reduction. The following lemma, which may be useful in other contexts, shows that if unit cost multiplication is allowed, we can reverse medium sized subwords in constant time.

Lemma 18. *On a RAM with unit cost multiplication in addition to the basic instruction set, reversing a string containing $s \leq \sqrt{w}$ bits can be done in constant time.*

Proof. Assume that the subword is in the right hand side of the word. The algorithm works in two steps. In the first step, it makes copies of the s least significant bits across the word and applies masks to get one copy of each of the bits of s in equidistant positions, but in reverse order. The second step gathers the bits together again. We use a result of Fredman and Willard [13] that guarantees that if y is a word with set bits on equidistant positions only, then for some w constants a and b, the expression $((a * y) \text{ AND } b) \downarrow w$ gathers these bits together on the least significant positions in a word in the same order as they were in y. The entire algorithm for reversing an s-bit string x is:

- $y = ((x \cdot d) \text{ AND } m) \downarrow (s-1)$
- RETURN $(a \cdot y) \downarrow ((s-1)s+1) \text{ AND } b$

where $a = \sum_{i=0}^{s-1} 2^{(s-i-1)s+(i+1)}$, $b = \sum_{i=0}^{s-1} 2^i$, $d = \sum_{i=0}^{s-1} 2^{i(s+1)}$ and $m = \sum_{i=0}^{s-1} 2^{i(s+1)+s-1-i}$.

Theorem 19. *Any RAM program multiplying two w-bit words using the basic instruction set uses time $\Omega(\log w)$. This is true, even if a precomputed table of size $O(2^{w^{1/20}})$ is allowed.*

Proof. Combine Lemma 18 and Theorem 17.

It is interesting to note that the following somewhat weaker lower bound follows more or less directly from Håstad's size-depth tradeoff for unbounded fan-in circuits computing multiplication [16]: time $\Omega(\log w / \log\log w)$ is necessary, even if a precomputed table of size $w^{O(1)}$ is allowed. This is *not* the case for the other two lower bounds, as reverse and least significant bit are AC^0 functions.

References

1. S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writting. In *3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 463–472, Orlando, Florida, 1992.
2. A. Andersson. Sublogarithmic searching without multiplications. In *36th IEEE Symposium on Foundations of Computer Science*, pages 655–663, 1995.
3. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *27th ACM Symposium on Theory of Computing*, pages 427–436, Las Vegas, Nevada, 1995.
4. A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n \log\log n})$ is necessary and sufficient. In *37th IEEE Symposium on Foundations of Computer Science*, pages 538–546, Burlington, Vermont, 1996.

5. A.M. Ben-Amram and Z. Galil. When can we sort in $o(n \log n)$ time? In *34$^{\text{th}}$ IEEE Symposium on Foundations of Computer Science*, pages 538–546, Palo Alto, California, 1993.
6. G.S. Brodal. Predecessor queries in dynamic integer sets. In *Proceedings 10$^{\text{th}}$ Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1997 (To appear).
7. A. Brodnik. Computation of the least significant set bit. In *Proceedings Electrotechnical and Computer Science Conference*, volume B, pages 7–10, Portorož, Slovenia, 1993.
8. A. Brodnik and J.I. Munro. Membership in a constant time and a minimum space. In *Proceedings 2$^{\text{nd}}$ European Symposium on Algorithms*, volume 855 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1994.
9. A. Brodnik and J.I. Munro. Neighbours on a grid. In *Proceedings 5$^{\text{th}}$ Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 307–320. Springer-Verlag, 1996.
10. J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.
11. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Technical Report 513, Fachbereich Informatik, Universität Dortmund, Dortmund, Germany, 1993.
12. M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
13. M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.
14. M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994.
15. M. Furst, J.B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, April 1984.
16. J. Håstad. Almost optimal lower bounds for small depth circuits. In *18$^{\text{th}}$ ACM Symposium on Theory of Computing*, pages 6–20, Berkeley, California, 1986.
17. P.B. Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *Proceedings 23$^{\text{rd}}$ International Colloquium on Automata, Languages and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 442–451. Springer-Verlag, 1996.
18. R. Raman. Priority queues: Small, monotone, and trans-dichotomus. In *Proceedings 4$^{\text{th}}$ European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer-Verlag, 1996.
19. A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
20. M. Thorup. On RAM priority queues. In *7$^{\text{th}}$ ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, Atlanta, Georgia, 1996.
21. M. Thorup. Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations. In *8$^{\text{th}}$ ACM-SIAM Symposium on Discrete Algorithms*, pages 352–359, New Orleans, Louisiana, 1997.

This article was processed using the LaTeX macro package with LLNCS style