SIAM J. COMPUT.
Vol. 17, No. 1, February 1988

005

FAST SIMULATION OF TURING MACHINES BY RANDOM ACCESS MACHINES*

JYRKI KATAJAINEN†, JAN VAN LEEUWEN‡ AND MARTTI PENTTONEN†

Abstract. We prove that a $T(n)$ time-bounded, $S(n)$ space-bounded and $U(n)$ output-length-bounded Turing machine can be simulated in $O(T(n)+(n+U(n))\log\log S(n))$ time by a random access machine (with no multiplication or division instructions) under the logarithmic cost criterion.

Key words. Turing machine, random access machine, simulation, computational complexity

1. Introduction. In the theory of computation one uses both the Turing Machine (TM) and the Random Access Machine (RAM) as standard models of effective computing (see e.g., [1]). Whereas the models are vastly different in detail, it is well known that the machines are "equivalent" in computational strength. More precisely, one can show that the machines are polynomially related in the sense of computational complexity theory (see [1, § 1.7] or [2]): a TM can simulate a RAM in $O(T(n)^2)$ time and a RAM can simulate a TM in $O(T(n)\log T(n))$ time, where $T(n)$ is the time complexity of the simulated machine and RAMs are assumed to use the so-called logarithmic cost criterion. In the result, RAMs are assumed without explicit "single" multiplication or division instructions in their instruction set. Slot and van Emde Boas [15] have shown that TMs and RAMs can simulate one another within only a constant factor of extra space.

When we speak of the simulation of one machine by another, we require that on the same input the latter machine produce the same output as the former one. In general, the simulating machine passes through an analogous computation, but it may also contain some auxiliary computations intermixed. These auxiliary steps are, of course, included in the complexity of the simulating machine.

Several studies have attempted to refine or lower the simulation costs between the two models, especially for the case of simulating RAMs by TMs (see e.g., Wiedermann [16] for some recent results). In this paper we consider the efficient simulation of TMs by RAMs. Let $T(n)$ denote the time complexity, $S(n)$ the space complexity and $U(n)$ an upperbound on the length of the longest output on inputs of length n. The following results are known.

THEOREM A (Folklore, see e.g., [1, § 1.7]). *A* TM *can be simulated in* $O(T(n)\log S(n))$ *time by a* RAM *(with no multiplication or division instructions) under the logarithmic cost criterion.*

THEOREM B (Paul [11, § 3.3]). *A* TM *can be simulated in* $O(T(n)+n\log n+U(n)\log T(n))$ *time by a* RAM *(with no multiplication or division instructions) under the logarithmic cost criterion.*

Theorem B shows that TMs can be simulated by RAMs with no essential time-loss provided $T(n)\geqq n\cdot\log n$ and $U(n)\leqq T(n)/\log T(n)$. Note, however, that [11, § 3.3] assumes stronger RAMs with shift instructions, that is (multiplication and) division by 2. Related results are also found in literature: Dymond and Tompa [4] proved that

* Received by the editors March 5, 1986; accepted for publication (in revised form) April 16, 1987.

† Department of Computer Science, University of Turku, 20500 Turku, Finland.

‡ Department of Computer Science, University of Utrecht, P.O. Box 80012, 3508 TA Utrecht, The Netherlands.

a TM can be simulated in time $\sqrt{T(n)}$ by a parallel RAM. Hopcroft, Paul and Valiant [7] simulate a TM in time $T(n)/\log T(n)$ by a unit-cost RAM. Robson [13] speeds up a TM computation by a probabilistic RAM.

In this paper we improve Theorem A to Theorem C as follows.

THEOREM C. *A* TM *can be simulated in* $O(T(n) \log\log S(n))$ *time by a* RAM *(with no multiplication or division instructions) under the logarithmic cost criterion.*

We also can improve Theorem B to Theorem D as follows.

THEOREM D. *A* TM *can be simulated in* $O(T(n)+(n+U(n)) \log\log S(n))$ *time by a* RAM *(with no multiplication, division or shift instructions) under the logarithmic cost criterion.*

This is, indeed, an improvement of Theorem B because in the case $\log n < \log\log S(n)$, $T(n)$ is the dominating term. This theorem improves also Theorem C, because $T(n) \geqq n$ and $T(n) \geqq U(n)$. None of the results assume that $T(n)$, $S(n)$ or $U(n)$ are constructible.

As an example of the use of Theorem C we mention the following corollary.

COROLLARY E. *Any linear time* TM *can be simulated in* $O(n \log\log n)$ *logarithmic time by a* RAM *(with no multiplication or division instructions).*

It follows that e.g. the reversal of a string of n inputs can be output by a RAM in $O(n \log\log n)$ units of logarithmic time. We can apply the above corollary also to the string-matching problem, where the task is to find all occurrences of a given pattern of length m from the text of length n, $m \leqq n$. The string-matching can be done in $O(n)$ time on a TM as shown by Fischer and Paterson [5] (see also [6]), and therefore in $O(n \log\log n)$ units of logarithmic time on a RAM.

In language acceptance the size of the output is constant. Hence, by Theorem D we also have

COROLLARY F. *Any language accepted by a* $T(n)$ *time-bounded,* $S(n)$ *space-bounded* TM *can be accepted in* $O(T(n)+n \log\log S(n))$ *time on a* RAM.

The paper is organized as follows. In § 2 we recapitulate some basic definitions. In § 3 we prove Theorem C and Theorem D. Finally, in § 4 we discuss how these improvements were achieved and how the results could probably be improved further.

2. Machine models. We define TMs and RAMs such that they appear as instances of the same abstract model, following the guidelines of [14]. The machines have very similar input, output and control structures, but differ in the structure and the use of the memory. The definition of TMs and RAMs is included to fix the particular instruction sets.

2.1. Turing machines. We describe the "parts" of a Turing machine without much formal notation. We assume that the input, output and work-tape alphabet is $\{0, 1\}$ and refer to the individual symbols as bits. A (multitape) TM consists of the following parts (compare [1, § 1.6]):

(i) *a* one-way read-only *input tape*, containing a bit string followed by an endmarker #.

(ii) *a* one-way write-only *output tape*, where a bit string will be written.

(iii) k two-way read-write *work-tapes* ("memory"), containing bits in successive memory cells. The tapes are two-way infinite. On each tape there is a separate read-write head that can be activated for reading, writing or moving one tape-cell to the left or to the right.

(iv) *a* TM *program*, which is a finite sequence of labelled or unlabelled instructions from a fixed instruction set (see below). No two instructions should carry the same label.

The instruction set of a TM contains eight instruction types:

(1) **input** $\lambda_0, \lambda_1, \lambda_\#$: causes a "next" input symbol α to be read, and the input head moves one cell to the right (except on #). Depending on whether α is 0, 1 or #, control is transferred to the instruction with label $\lambda_0, \lambda_1, \lambda_\#$.

(2) **output** β: causes a bit β to be output, and the output head moves one cell to the right.

(3) **jump** λ: transfers control to the instruction with label λ.

(4) **halt**: halts the program.

(5) **head** i: activates the read-write head on the ith work-tape ($1 \leqq i \leqq k$). Only one read-write head will be active at a time.

(6) **write** β: causes a bit β to be written in the tape-cell designated by the active read-write head.

(7) **branch** λ_0, λ_1: causes the bit β to be read from the tape-cell designated by the active head. Depending on whether β is 0 or 1 control is transferred to the instruction with label λ_0 or λ_1.

(8) **move** δ (with $\delta \in \{L, R\}$): moves the active read-write head one cell to the left or to the right depending on whether δ is L or R.

We assume that initially all work-tapes contain 0 in every cell, and that head 1 is active. The computation starts from the first instruction and thereafter the instructions of a program are executed in their successive order unless a jump instruction orders otherwise.

The *time complexity* $T(n)$ of a TM is the largest number of instructions executed in halting computations on inputs of length n. The *space complexity* $S(n)$ is the largest number of cells occupied on any work-tape in halting computations on inputs of length n. The *output complexity* $U(n)$ is the length of the longest output produced in halting computations on inputs of length n.

Because a TM with a two-way infinite tape can be simulated by a TM with a one-way infinite tape in real time (see e.g.,[8, § 7.5]), we shall assume that the work-tapes of a TM are one-way infinite, say infinite to the right. Initially all read-write heads are positioned on the leftmost cell of their work-tape. By the standard construction used in the above simulation [8, § 7.5], we can further assume that a read-write head is never moved off the left end of the work-tape. (Thus the computation is stopped by a halt instruction, not by the fall of a read-write head.) Although in the construction the tape alphabet is enlarged, it is straightforward to return into the binary alphabet (see also [8, § 7.8]).

2.2. Random access machines. In describing the random access machine, we only emphasize the parts that are different from those of a TM. Parts (i) and (ii) are very similar for a RAM but instead of (iii) one has the following set-up (compare [1, § 1.2]):

(iii′) a special register called the *accumulator* (AC) and a countable sequence of ordinary *registers* ("memory") indexed by the nonnegative integers (used as addresses). Each register can hold an arbitrary nonnegative integer in binary notation. Only data stored in the AC can be operated upon.

A RAM *program* is defined as in (iv), but the instruction set of a RAM differs from the instruction set of a TM. In instructions, the contents of register j are denoted by $\langle j \rangle$. The instruction set of a RAM contains twelve instruction types:

(1′)-(4′): similar to the instructions (1)-(4) of a TM.

(5′) **jzero** λ: transfers control to the instruction with label λ if $\langle \mathrm{AC} \rangle = 0$, and continues to next instruction otherwise.

(6′) **load** $=j$: loads the integer j into AC.

(7′) **load** j: loads $\langle j \rangle$ into AC.

(8′) **load** $*j$: loads $\langle\langle j\rangle\rangle$ into AC ("indirect addressing").
(9′) **store** j: stores $\langle \mathrm{AC}\rangle$ into register j.
(10′) **store** $*j$: stores $\langle \mathrm{AC}\rangle$ into register $\langle j\rangle$.
(11′) **add** j: adds $\langle j\rangle$ to the current value in AC.
(12′) **sub** j: subtracts $\langle j\rangle$ from the current value in AC. In order to keep the contents of the AC nonnegative, we assume that subtraction is proper, i.e., the result is 0 whenever $\langle \mathrm{AC}\rangle \leqq \langle j\rangle$.

We assume that all registers, including the AC, initially contain 0. Memory need not be used contiguously.

We do not simply count the number of instructions executed in a RAM program but use the so-called *logarithmic cost criterion*: the "time" charged for an instruction is equal to the sum of the sizes (in bits) of the integers (addresses and data) involved in its execution. Note that the size of a positive integer m is $\lceil \log (m+1)\rceil \sim \log m$, and the size of zero is 1. The *time complexity* $T(n)$ of a RAM is the largest amount of time, measured according to the logarithmic cost criterion, used in halting computations on inputs of length n. See Slot and van Emde Boas [15] for notions of space complexity for RAMs.

It will be convenient to use various extensions to the basic RAM instruction set, provided that the execution time is adequately measured by the logarithmic cost criterion. By using subtractions and a trick introduced in [13, pp. 495–496], one can easily show that this is the case for comparison instructions. It should be noted, however, that the properness of the subtraction operation is not needed anywhere in the subsequent proofs because we always know which of the two operands is greater. Also in some algorithms it is convenient to have a RAM with k separate *memories* (or *arrays* as called by Cook and Reckhow [2]), $k>1$, each consisting of a countable sequence of registers indexed $0, 1, 2, \cdots$. We call this a "multimemory" RAM.

LEMMA 2.2.1. *Every $T(n)$ time-bounded multimemory* RAM *can be simulated in $O(T(n))$ time by an ordinary* RAM.

Proof. The technique was essentially given in [2]. The idea is simply to interleave the RAM memories into one, using addresses $i+kj-1$ for register j of the ith memory $(1 \leqq i \leqq k, j \geqq 0)$. Those addresses can be computed in $O(k \log j)$ time, which multiplies the time bound by a constant factor. □

It is important to state explicitly the basic instruction set of the RAM. However, for the sake of the readability, we extend it with some Pascal-like control structures that have obvious translations to the basic RAM instructions.

3. The simulation of a TM by a RAM.

Consider a $T(n)$ time-bounded, $S(n)$ space-bounded TM. The simple idea underlying Theorem A is to represent the cells of the work-tapes in consecutive registers of a RAM, with additional registers containing current read-write head positions. Every step of the TM is easily simulated in $O(\log S(n))$ time on a RAM, assuming the logarithmic cost criterion. In the simulation underlying Theorem B, a saving in the cost per step is achieved by precomputing in a table the action of the TM on all blocks of a suitable size. A subcomputation corresponding to the size of the block reduces to a table look-up.

We will also use blocking to balance the costs of the address and the contents of a memory location. At first we keep the idea of step-wise simulation. We use blocking merely to localize the active region of the work-tape during a time-interval. For step-wise simulation, the active block is swapped to a low-indexed region of the memory in order to save in the access time under logarithmic cost criterion. (It is interesting to compare this technique to the swapping of pages to and from disk in paged virtual memory operating systems, see e.g., Deitel [3]. Another analogy is a hardware cache. The cache idea was also used by Loui in [9].)

A serious problem is determining the optimal block size together with efficient algorithms for unpacking and packing blocks. A further problem is that $T(n)$ and $S(n)$ need not be constructible. For simplicity, we denote these values by T and S. In § 3.1 we assume that n and $S(n)$ are known at the beginning of the computation, and determine the optimal block size in this case. In § 3.2 we remove this assumption and notice that the same time bound holds even though the block size is determined dynamically during the computation. Ultimately, in § 3.3 we improve the table look-up method of [11]. Interestingly, the packing and unpacking techniques developed in §§ 3.1 and 3.2 will now be useful in reading input blocks and writing output blocks.

3.1. The static step-wise simulation using blocked memory. We assume now that n and $S(n)$ are known in advance. We will begin the basic simulation algorithm together with the necessary blocking and deblocking algorithms. By the time analysis of this algorithm we determine the optimal block size. For simplicity, we speak of one TM tape only; if there are many work-tapes, they are treated analogously, independently.

The basic idea of the simulation is to divide the tape into S/b blocks of b cells. Hence, b successive cells of the TM are represented by a number (in the range $0, \cdots, 2^b-1$) in a register of the RAM. By Lemma 2.2.1 we can assume that the RAM keeps the "blocked" representation of the work-tape in a separate memory. The position of the tape head of the TM is indicated by an address to the active block (a number in the range $1, \cdots, S/b$) together with an address within a block $(0, \cdots, b-1)$, both stored in fixed RAM registers. One simulation cycle, corresponding to b steps of the TM, consists of accessing the active block with two neighbours, unpacking them to low-indexed registers, directly simulating the next b steps of the TM, and packing the updated bits back to the same registers. The neighbouring blocks are taken along in order to guarantee that in all cases b simulation steps can be taken staying in the unpacked zone. This unpacked $3b$ bit zone, kept in a separate memory, is called the *window*.

The simulation of the single TM instructions is quite obvious; it is done as in the proof of Theorem A. It is easy to construct a procedure simulating an instruction of the TM, updating the contents of the window, the head address and the current instruction label. Now we can represent the simulation in the form of a RAM program as follows:

```
procedure simulate
{Suppose that the block size $b$ is given.}
activeblock := 1
{the first block is active, with empty left neighbour}
head := the first address of the middle block of the window
loop {until a halt instruction in the simulation}
    loadwindow(activeblock, $b$)
    for $b$ times do simulate an instruction
    storewindow(activeblock, $b$)
    if head moved to neighbour then
        update head and activeblock addresses
```

The procedure loadwindow fetches the contents of the active block and its neighbours into low-indexed registers, and unpacks the b-bit integers. The procedure storewindow packs the window blocks, and stores them by overwriting their older copies.

We shall now attack the problem of packing and unpacking the blocks efficiently. As our RAM model does not include division or shift instructions, we have to invent another method for finding the bit representation of a number and vice versa. We will see that unpacking and packing can be done efficiently with precomputed tables.

As a first attempt one could decode numbers to bit-strings by building a table that gives the decoding directly. For example the table could contain the b bits of a number n ($<2^b$) in the registers nb, $nb+1, \cdots, nb+b-1$. A disadvantage of this method is that, while bits are obtained directly, the access of them may cost $O(\log n)$. For this reason loading a window takes $O(b^2)$ time. By a similar analysis as what follows, one can see that this would give $O(T\sqrt{\log S})$ simulation algorithm. However, we can do unpacking and packing in $O(b \log b)$ time.

The efficient decoding of a number to its bit representation and vice versa is based on a divide-and-conquer strategy with precomputed shift tables. We will first build the necessary tables and then give the unpacking and packing algorithms.

We assume that each table is stored in its own memory. We will need tables *lshift*, *rshift*, *origin* and *power*. By $l$shift(i), $r$shift(i), $\cdots$, we denote the contents of the register i reserved for $l$shift, $r$shift, $\cdots$.

The tables $l$shift and $r$shift in Fig. 1 contain as subtables shift tables for 1-bit numbers, 2-bit numbers, 4-bit numbers etc. The divide-and-conquer strategy implies that b-bit numbers are shifted $b/2$ bits to the right or b bits to the left. The entries of the tables are numbers rather than bit strings. Thus, for example, the number $75 = 01001011_2$ has the right shift $4 = 0100_2$ and $4 = 0100_2$ has the left shift $64 = 01000000_2$. The origin table expresses where subtables begin: The shift tables for 2^i-bit numbers begin at origin(i).

origin:

register	0	1	2	3	4	$\cdots$	i
contents	0	2	6	22	278		$2^{2^0}+2^{2^1}+\cdots+2^{2^{i-1}}$

rshift:

register	0	1	2	3	4	5	6	7	8	$\cdots$	21	$\cdots$	origin(i)+j
block size	1	bit	2	bits				4	bits				2^i bits
block value	0	1	0	1	2	3	0	1	2		15		j
contents	0	0	0	0	1	1	0	0	0		3		j div $2^{2^{i-1}}$

lshift:

register	0	1	2	3	4	5	6	7	8	$\cdots$	21	$\cdots$	origin(i)+j
block size	1	bit	2	bits				4	bits				2^i bits
block value	0	1	0	1	2	3	0	1	2		15		j
contents	0	2	0	4	8	12	0	16	32		240		$j \cdot 2^{2^i}$

FIG. 1. *The origin, rshift and lshift tables.*

We have to first analyze how much the building of the tables costs.

LEMMA 3.1.1. *The tables origin, rshift and lshift up to block size b ($=2^k$) can be built in logarithmic time $O(b2^b)$.*

Proof. Assuming that the values origin($i-2$) and origin($i-1$) are already computed, the following program will compute the ith origin value, $i \geqq 2$.

```
**procedure** buildorigin($i$)
$t :=$ origin($i-1$) $-$ origin($i-2$)   $\{t := 2^{2^{i-2}}\}$
origin($i$) := origin($i-1$)
**for** $t$ times **do** origin($i$) := origin($i$) + $t$
```

Clearly, its time complexity is $O(2^{2^{i-2}} \cdot 2^{i-1})$.

When constructing the ith $r$shift and $l$shift subtables we can use the origin values for $i-1$, i and $i+1$.

```
procedure buildrshift(i)
$j :=$ origin($i$); $x := 0$; $t :=$ origin($i$) − origin($i-1$)   $\{t := 2^{2^{i-1}}\}$
for $t$ times do
    for $t$ times do rshift($j$) $:= x$; $j := j+1$
    $x := x+1$

procedure buildlshift(i)
$j :=$ origin($i$); $x := 0$; $t :=$ origin($i+1$) − origin($i$)   $\{t := 2^{2^i}\}$
for $t$ times do lshift($j$) $:= x$; $j := j+1$; $x := x+t$
```

The execution of both procedures requires $O(2^{2^i} \cdot 2^i)$ time. Thus the tables up to k can be constructed in time $O(\sum_{i=1}^{k} 2^{2^i} \cdot 2^i) = O(2^k \cdot 2^{2^k}) = O(b2^b)$. □

We also need powers of 2 for unpacking numbers to bit strings, and for packing bit strings to numbers. It is useful to precompute them, too, in the table *power*.

LEMMA 3.1.2. *The table power$(i) = 2^i$ up to kth power can be built in $O(k^2)$ logarithmic time.*

Proof. A new power can be computed by doubling the previous one by addition. This method gives the time bound $O(k^2)$. □

Now we are ready to present the unpacking and packing algorithms.

LEMMA 3.1.3. *Assuming that the tables lshift, rshift, origin and power up to the block size b are available, it is possible to compute the b-bit representation of an integer $n < 2^b$, and the numeric value of a b-bit string, both in $O(b \cdot \log b)$ time.*

Proof. The procedure unpack(n, j, a) unpacks a number $n < 2^{2^j}$ to its 2^j-bit representation beginning at the ath register of the window. The procedure is as follows:

```
procedure unpack($n, j, a$)
if $j = 0$ then window($a$) $:= n$
else $n_1 :=$ rshift(origin($j$) + $n$); $n_2 := n -$ lshift(origin($j-1$) + $n_1$)
    unpack($n_1, j-1, a$); unpack($n_2, j-1, a+$power($j-1$))
```

For clarity, we have written the algorithm in recursive form. The recursion can be eliminated by using one memory as a stack where the second recursive call is stored while the first is executed. While unpacking a number $n < 2^j$ there are never more than $\log j$ calls in the stack. In order to balance access cost it is economical to initiate the stack at the address $\log b$ and let it grow downwards. If we denote by $t(x)$ the logarithmic time of unpacking an x-bit number, by analyzing the program we get

$$t(1) = k_1 \log b,$$
$$t(x) = 2t(x/2) + k_2 x + k_3 \log b$$

which gives $t(b) = O(b \cdot \log b)$.

The procedure pack(a, j, n) computes the numeric value of the bit string window(a), window$(a+1), \cdots$, window$(a+2^j-1)$. Also it is written recursively:

```
procedure pack($a, j, n$)
if $j = 0$ then $n :=$ window($a$)
else pack($a, j-1, n_1$); pack($a+$power($j-1$), $j-1, n_2$)
    $n :=$ lshift(origin($j-1$) + $n_1$) + $n_2$
```

The recursion is controlled as in unpacking. Also the time analysis is analogous. □

We can now obtain a preliminary version of Theorem C:

THEOREM 3.1.4. *Assuming that n and $S(n)$ are known, a $T(n)$ time-bounded, $S(n)$ space-bounded* TM *can be simulated in $O(T(n) \log \log S(n))$ logarithmic time by a* RAM.

Proof. The total time of the static simulation is bounded by

$$T_{\mathrm{RAM}} = b2^b + T/b(\log S + b\cdot\log b + b\cdot\log b + b\cdot\log b + \log S)$$

where $b2^b$ is needed for the construction of the tables, T/b is the number of cycles in the simulation loop, $\log S$ is the cost of loading and storing the blocks, $b\cdot\log b$ is the cost of unpacking and packing the blocks of the window, and another $b\cdot\log b$ is needed for the elementary simulation steps. By choosing $b = c\cdot\log S$, $c<1$, we get $T_{\mathrm{RAM}} = O(T\cdot\log\log S)$. □

3.2. The dynamization. Until now we have assumed that the space requirement S of the TM on an input is known in advance, which is not the case. Now we shall remove this assumption by using the well-known technique (see, e.g., [10]) which consists of using one space bound while sufficient and increasing it when necessary. As long as the space bound

$$S_0, S_1, \cdots, S_i = 2^{2^{i+1}}, \cdots$$

is sufficient, the block size $b_0, b_1, \cdots, b_i = 2^i, \cdots$, is used respectively. Every time a space bound is exceeded, the simulating memory must be reorganized by combining the blocks pairwise. As the size of the blocks grows also the tables must be grown accordingly. It is worth noting that, unlike in [11], input head need not be reset and the computation be restarted from the beginning when the block's size is increased.

The dynamic simulating algorithm gets the following form.

```
procedure simulate
activeblock := 1
head := the first address of the middle block of the window
$b := 1$; $S := 4$
Initialize tables lshift, rshift, origin and power;
loop {until a halt instruction in the simulation}
    while $\lceil\log(S+1)\rceil \leq 4b$ do
        loadwindow(activeblock, $b$)
        for $b$ times do
            Simulate an instruction
            if head on a previously unvisited cell then $S := S+1$
        storewindow(activeblock, $b$)
        if head moved to neighbour then
            update head and activeblock addresses
    $b := b+b$
    Combine successive blocks pairwise
    Update head and activeblock corresponding to the reorganization
    Update lshift, rshift, origin and power to the new block size
```

The analysis of the dynamic simulation is basically the same as in the previous section, but now there is the extra work of reorganizing the memory. In spite of using wrong block sizes at the beginning, the program still behaves asymptotically fast. Before going to the time analysis of the program, we prove a small counting lemma, also used in [2], [12].

LEMMA 3.2.1. *A binary counter of any event occurring t times during the computation can be maintained in $O(t)$ time. As a by-product we get easily the length of the binary representation of the count, which is $\lceil\log(t+1)\rceil$.*

Proof. A memory is reserved for the binary representation of the count. For each bit of the count (without leading zeros), two bits are used. The ith least significant bit 0 or 1 is represented by 00_2 or 01_2 in the register i. If the actual value of the count is t, $2^{j-1} \leqq t \leqq 2^j$, then 11_2 in the register j marks the end of the representation.

Each counting step is simulated by a sweep over the representation in the memory, beginning at the register 0. 01's are replaced by 00's, until 00 is met which becomes 01. However, if 11 is met instead of 00, it is replaced by 01 and 11 is stored in the next register. At any moment, the address of 11 corresponds to the logarithm of the count, and the registers below it correspond to the bit representation of the count.

The linear time bound is seen as follows. During a count up to $t = 2^j$, the length of the sweep is $t/2$ times 1, $t/4$ times 2, $t/8$ times 3 and so forth. Hence the total logarithmic time is proportional to

$$2^{j-1} \cdot 1 \cdot \log 1 + \cdots + 2^{j-i} \cdot i \cdot \log i + \cdots + 1 \cdot j \cdot \log j = O(t)$$

as can be seen by the quotient test applied to the infinite series $2^{-1} \cdot 1 \cdot \log 1 + \cdots 2^{-i} \cdot i \cdot \log i + \cdots$. □

The complexity analysis of the dynamic simulation is basically the same as in the previous subsection, but we have to take into account the following differences. A counter must be maintained in order to determine when block size must be increased. Whenever the block size is increased, the simulating memory must be reorganized by combining blocks pairwise. This is extra work, following from the fact that nonoptimal block size was used. Fortunately, the amount of extra work caused by reblockings and nonoptimal block size proves to be insignificant.

By Lemma 3.2.1, no more than $O(S)$ time is used in the computation of S. The time needed for the comparison $\lceil \log(S+1) \leqq 4b$ is $O(\log b)$ which is covered by the $O(b \cdot \log b)$ used in the simulation. Whenever the test fails, b is doubled and tables are updated. Note that by Lemmas 3.1.1 and 3.1.2 the total time needed in the construction of the tables is $O(b2^b) = O(S)$, because $2b \leqq \log S < 4b$. Each doubling of b is followed by reblocking. This is most easily done by the *l*shift table in $O((S_i/b_i) \cdot b_i)$ time. The superexponential growth of the series $\{S_i\}$ implies that all reblockings can be done in time $O(S)$.

For the final analysis of the simulation, assume that T_i steps of the TM are simulated using block size b_i. Hence the total time of the simulation is bounded by

$$\sum_{i=0}^{\log\log S} (T_i/b_i)(\log(S_i) + b_i \log b_i) = \sum_{i=0}^{\log\log S} T_i \log b_i$$

$$\leqq \log b \sum_{i=0}^{\log\log S} T_i = T \cdot \log b = T \cdot \log\log S.$$

Hence, the time bound $O(T \cdot \log\log S)$ holds also for the dynamic simulation. We have proved

THEOREM 3.2.1. *A $T(n)$ time-bounded and $S(n)$ space-bounded* TM *can be simulated in $O(T(n) \log\log S(n))$ time on a* RAM *without multiplication and division instructions.*

3.3. Simulation with precomputed transition tables. A further improvement in the simulation is achieved by combining the fast packing and unpacking algorithms of § 3.1 with the use of precomputed transition tables as in [7], [11].

As in § 3.1, we shall first assume that the optimal block size is known at the beginning of the simulation. The computation is speeded up by precomputing in a table all subcomputations of length b. For this purpose, the work-tape is divided into blocks of size b, as in previous constructions. Now, input and output must also be handled blockwise, because a subcomputation of length b may read b bits and write b bits.

In the new construction, the simulation itself is simply table look-up and takes, as we will see, only $O(T(n))$ logarithmic time. Paul [11] organizes the table as a heap (like in heapsort) and uses shift operations in the calculation of the pointers. We store the table as a multidimensional array and calculate the addresses by precomputed shift tables, avoiding the use of shift operations. For input and output blocking, the packing and unpacking techniques of § 3.1 are used. We shall see that, if $U(n)$ is the maximum output length for inputs of length n, $O((n+U(n)) \log\log S(n))$ is sufficient for input and output, when it is done to and from low addresses, not higher than $O(\log S(n))$. Hence, the total time of the simulation is $O(T(n)+(n+U(n)) \log\log S(n))$.

In order to guarantee that large enough input blocks are always available, and there is not too frequent need for filling the blocks, we will use an input buffer of length $2b$. It is filled every time when more than half of the bits have been consumed. The same idea is useful for outputting.

We shall precompute the transition table

$$\text{transit}\,(i, x, l, p, v_{-1}, v_0, v_1, j, y) = (i', l', p', v'_{-1}, v'_0, v'_1, d, j', y')$$

where all entries are integers: $i, j, p, p' < b$; $i', j' < 2b$; $x, y, y' < 2^{2b}$; l, l' are program line numbers; $v_k, v'_k < 2^b$; and $d = -1, 0, 1$. The intuitive meaning of the table is the following: If i bits of the contents x of the input buffer have been consumed before, the TM is at its instruction l, the work-tape head is at the position p of the block v_0 whose neighbours are v_{-1} and v_1, and there are already j bits of y in the output buffer, then after b elementary steps of the TM, $i'(<2b)$ bits of x have been consumed, TM is at line l', the work-tape blocks have changed to v'_k and the head is at the position p' of v'_d, and there are j' bits of y' in the output buffer. With this notation, we can write the *static* simulating algorithm as follows:

```
procedure simulate
$i := 2b-1$; $j := 0$ {input and output buffers are empty}
$l := 1$; activeblock := 1; $p := 0$;
loop {until the simulation of a halt causes an exit}
    if $i \geqq b$ then fill input buffer to a new $x$ with $i = 0$;
    for $k = -1, 0, 1$ do $v_k :=$ work-tape(activeblock + $k$)
    $(i, l, p, v_{-1}, v_0, v_1, d, j, y) :=$ transit$(i, x, l, p, v_{-1}, v_0, v_1, j, y)$
    for $k = -1, 0, 1$ do work-tape(activeblock + $k$) := $v_k$
    activeblock := activeblock + $d$
    if $j \geqq b$ then output the $j$ bits of $y$ and make $j := 0$;
```

Some points of the program need closer examination.

First, as b successive steps of the TM are composed, the main loop is iterated T/b times.

If $i \geqq b$, the input buffer is filled as follows. First the $2b$ bits of x are unpacked by Lemma 3.1.3 in time $O(b \cdot \log b)$. Then the unused $2b - i$ bits are moved to positions $0, 1, \cdots$, followed by i new bits read from the input tape. This is easily done in $O(b \cdot \log b)$ time. Output is treated analogously.

Loading and storing work-tape blocks obviously takes $O(\log S)$ time. The transition table can easily be linearized to 2^{kb} registers, where k is a constant. By the shift tables similar to those in § 3.1, additions and subtractions, the address of a table entry can be computed, and the value can be decoded in $O(b)$ time.

We observe that input and output take $O(\log b)$ time for each bit, and hence $O((n+U(n))\log b)$ time in total. The cost of the simulation is of order $(T/b)(\log S + b)$. Hence, by choosing $b = O(\log S)$, we see that $O((n+U(n))\log\log S(n) + T(n))$ time is sufficient for simulation.

Finally, we shall show that the table can be built in $O(S)$ time. It can be done in the obvious way: For each table entry, simulate b steps of the TM and store the result in a register. The generation of the entries is controlled by nine nested loops, one for each argument of the table. To compute the value of each table entry, it is first unpacked to low-indexed registers in time $O(b\cdot\log b)$, then b steps of the TM are simulated in time $O(b\cdot\log b)$, and finally the result is packed in time $O(b\cdot\log b)$ and stored in time $O(b)$. Hence, the construction of the table takes $O(2^{kb}\cdot b\cdot\log b)$ time, which is $O(S)$, if $b = c\cdot\log S$ with a constant $c < 1/k$ is chosen.

Thus, we have proved Theorem D in the static case. As in § 3.2, the efficiency of the static construction can be achieved even though the optimal block size is not known in advance. We let the block size grow in the series $b_0, b_1, \cdots, b_i = 2^i, \cdots$, when the space requirement reaches the bounds

$$S_0, S_1, \cdots, S_i = 2^{2^{i+a}}, \cdots,$$

respectively. In S_i, the constant a is chosen such that $k < 2^a$. The *dynamic* simulating program is as follows.

```
procedure simulate
$i := 2b-1$; $j := 0$
$l := 1$; activeblock $:= 1$; $p := 0$;
$b := 1$; $S := 2^{2^a}$ {$a$ is a small constant}
Initialize lshift, rshift, origin, power and transit tables;
loop {until a halt in the simulation}
    while $\lceil\log(S+1)\rceil \leqq 2^{a+1}b$ do
        if $i \geqq b$ then fill input buffer;
        for $k = -1, 0, 1$ do $v_k :=$ work-tape(activeblock $+ k$)
        $(i, l, p, v_{-1}, v_0, v_1, d, j, y) :=$ transit$(i, x, l, p, v_{-1}, v_0, v_1, j, y)$
        Increase $S$ when necessary;
        for $k = -1, 0, 1$ do work-tape(activeblock $+ k$) $:= v_k$
        activeblock $:=$ activeblock $+ d$
        if $j \geqq b$ then flush the output buffer;
    $b := b + b$;
    Combine successive memory blocks pairwise;
    Update $p$ and activeblock corresponding to the reorganization;
    Update lshift, rshift, origin and power;
    Construct transit for the block size $b$;
```

There is very little new in the analysis of the program. During the whole simulation, $O(S)$ time is used in the construction of the tables *l*shift, *r*shift, origin and power. Every time when a new transit table is constructed, $O(S_i)$ time is spent, hence $\sum S_i = O(S)$ in total.

As the block size never exceeds $O(\log S)$, input and output packing, as well as unpacking, can be done in $O(\log\log S)$ time per bit, $O((n+U(n))\log\log S(n))$ time in total. As the simulation for each block size is linear, the total simulation is linear, too.

Hence, we have proved

THEOREM 3.3.1. *Any $T(n)$ time-bounded, $S(n)$ space-bounded and $U(n)$ output length-bounded* TM *can be simulated in $O(T(n) + (n+U(n))\log\log S(n))$ logarithmic time by a* RAM.

4. Discussion. We have improved the simulations of TMs by RAMs that can be found in [1] and [11]. We first improved the strategy of [1] by introducing blocking, a well-known technique used for speed-ups. Essential in our constructions was how to pack and unpack blocks efficiently. We did it with a precomputed table. The same tabulation technique was also used to improve the method of [11]. First, it allowed a natural look-up method, and second, with it input and output could be done more efficiently.

It is interesting to observe that in the simulation of Theorem D, faster input and output would improve the time bound. With a better input/output pattern the simulation could perhaps be sped up further. Whether such an improvement is possible remains as an open problem.

Acknowledgments. We are grateful to the referee for remarks and references that have improved the representation.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. A. COOK AND R. A. RECKHOW, *Time bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354-375.

[3] H. M. DEITEL, *An Introduction to Operating Systems*, Addison-Wesley, Reading, MA, 1984.

[4] P. W. DYMOND AND M. TOMPA, *Speedups of deterministic machines by synchronous parallel machines*, J. Comput. System Sci., 30 (1985), pp. 149-161.

[5] M. J. FISCHER AND M. S. PATERSON, *String-matching and other products*, in Complexity of Computation, R. M. Karp, ed., SIAM-AMS Proceedings 7, Amer. Math. Soc., Providence, RI, 1974, pp. 113-125.

[6] G. GALIL AND J. SEIFERAS, *Time-space-optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280-294.

[7] J. HOPCROFT, W. PAUL AND L. VALIANT, *On time versus space and related problems*, in Proc. 16th Annual IEEE Symposium on the Foundations of Computer Science, 1975, pp. 57-64.

[8] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[9] M. C. LOUI, *Minimizing pointers into trees and arrays*, J. Comput. System Sci., 28 (1984), pp. 359-378.

[10] M. H. OVERMARS, *The design of dynamic data structures*, Lecture Notes in Computer Science 156, Springer-Verlag, Berlin, 1983.

[11] W. J. PAUL, *Komplexitätstheorie*, Teubner Verlag, Stuttgart, 1978.

[12] M. PENTTONEN AND J. KATAJAINEN, *Notes on the complexity of sorting in abstract machines*, BIT, 25 (1985), pp. 611-622.

[13] J. M. ROBSON, *Fast probabilistic* RAM *simulation of single tape Turing machine computations*, Inform. and Control, 63 (1984), pp. 67-87.

[14] A. SCHÖNHAGE, *Storage modification machines*, this Journal, 9 (1980), pp. 490-508.

[15] C. SLOT AND P. VAN EMDE BOAS, *On tape versus core: an application of space efficient hash functions to the invariance of space*, in Proc. 16th Annual ACM Symposium on the Theory of Computing, ACM, New York, 1984, pp. 391-400.

[16] J. WIEDERMANN, *Deterministic and nondeterministic simulation of the* RAM *by the Turing machine*, in Proc. IFIP Congress 83, R. E. A. Mason, ed., North-Holland, Amsterdam, 1983, pp. 163-168.