Relations Among Complexity Measures

NICHOLAS PIPPENGER

IBM Thomas J Watson Research Center, Yorktown Heights, New York

AND

MICHAEL J. FISCHER

University of Washington, Seattle, Washington

ABSTRACT Various computational models (such as machines and combinational logic networks) induce various and, in general, different computational complexity measures Relations among these measures are established by studying the ways in which one model can "simulate" another It is shown that a machine with k-dimensional storage tapes (respectively, with tree-structured storage media) can be simulated on-line by a machine with one-dimensional storage tapes in time $O(n^{2-1/k})$ (respectively, in time $O(n^2/\log n)$) An *oblivious* machine is defined to be one whose head positions, as functions of time, are independent of the input, and it is shown that any machine with one-dimensional tapes can be simulated on-line by an oblivious machine with two one-dimensional tapes in time $O(n \log n)$ All of these results are the best possible, at least insofar as on-line simulation is concerned. By similar methods it is shown that n steps of the computation of an arbitrary machine with one-dimensional tapes can be performed by a combinational logic network of cost $O(n \log n)$ and delay $O(n)$

KEY WORDS AND PHRASES complexity, simulation, automata, Turing machine, time, circuit, network, size, cost

CR CATEGORIES 5 22, 5 25, 5 26, 6 1

1. *Introduction*

Since the appearance of Turing's fundamental paper [11], numerous variants of his machine-oriented computational model have been proposed. Many of these variants have been inspired by the observation that sequential access to one-dimensional tapes is not always appropriate in a computational model. They thus replace one-dimensional tapes by multidimensional tapes, tree-structured storage media, or other nonsequential storage structures. While these machines compute the same functions as those with one-dimensional tapes, they may compute them faster. One purpose of this paper is to study the dependence of computation time on storage structure. A second purpose is to study the relationship between the time required by a machine and the cost and delay required by a combinational logic network.

In Section 2 we show how machines with nonsequential storage structures can be simulated by machines with one-dimensional tapes. In particular, we show that any computation that can be performed in n steps by a machine with k-dimensional tapes can

The results in this paper were announced at the M I.T Workshop on Computational Complexity, Endicott House, Dedham, Massachusetts, August 6–10, 1973

This research was supported in part by the National Science Foundation under Research Grant GJ-34671 to M I T Project MAC

Authors' addresses: N Pippenger, Mathematical Sciences Department, P O Box 218, IBM Thomas J Watson Research Center, Yorktown Heights, NY 10598, M J Fischer, Department of Computer Science, University of Washington, Seattle, WA 98195.

be performed in $O(n^{2-1/k})$ steps by a machine with one-dimensional tapes. Hennie [3] has given an example of a computation for which $cn^{2-1/k}$ steps (for some $c > 0$) are necessary if the computation is to be performed on-line. That $O(n^{2-1/k} \log n)$ steps are sufficient was shown by Stoss [10]. (Stoss's simulation requires only two tapes; so it includes the result of Hennie and Stearns [4] as the special case $k = 1$.) We show that any computation that can be performed in n steps by a machine with tree-structured storage media can be performed in $O(n^2/\log n)$ steps by a machine with one-dimensional tapes. In this case, Hennie [3] has given an example for which $cn^2/(\log n)^2$ steps are required for on-line performance. We shall give an improved version of this example which shows that $cn^2/\log n$ steps are required.

In Section 3 we study the relationship between the time required by a machine and the cost and delay required by a combinational logic network. To do this, we use an auxiliary notion of independent interest: that of an oblivious machine. An *oblivious* machine is one whose head movements are fixed functions of time, independent of the inputs to the machine. We first show that any computation that can be performed in n steps by a machine with one-dimensional tapes can be performed in $O(n \log n)$ steps by an oblivious machine with two one-dimensional tapes. We also give an example for which $cn \log n$ steps are necessary for on-line performance. A similar simulation result, omitting only the qualification "oblivious," was given by Hennie and Stearns [4]; it is unknown whether this weaker problem requires $cn \log n$ steps for on-line performance.

The methods we use for oblivious machines can also be applied to combinational logic networks. We show that a computation that can be performed in n steps by a machine with one-dimensional tapes can be implemented by a network with cost $O(n \log n)$ and depth $O(n)$. That cost $O(n^2)$ and depth $O(n)$ are sufficient was shown by Savage [8] and Schnorr [unpublished].

More recently, Schnorr [9] has refined the techniques used in the present paper to take into account the program length and space requirements, as well as the time requirements of the machine.

2. *Simulation of Machines with Nonsequential Storage Structures*

The machines we shall consider will be bounded-activity machines, as introduced by Cook and Aanderaa [1]. Such a machine has a one-way read-only *input tape* (from which it reads a sequence of *input symbols*), a one-way write-only *output tape* (on which it writes a sequence of *output symbols*), a *control* (which can assume any of a finite number of *control states*), and *storage*.

Storage consists of a finite number of *storage media* and a greater or equal number of *storage heads*, one or more for each storage medium. Each storage medium consists of a countably infinite number of *cells* (each of which can contain any of a finite number of *storage symbols*). Each storage head will always be located at some cell of its storage medium and will be able to execute any of a finite number of *shifts* (which may be thought of as mappings from the set of cells of the medium into itself).

At the outset the control will be in a prescribed initial control state; all the cells of storage will contain a prescribed initial storage symbol (the *blank symbol*); and all the storage heads will be at prescribed initial locations (the *origins* of their respective storage media).

The operation of a machine will take place in a sequence of *steps*. At each step a machine will observe its control state and the symbols scanned by the heads on its input tape and storage media, and on the basis of this information it may change its control state, change the symbols scanned by the heads on its output tape and storage media, and shift any of its heads. This action will be specified by a deterministic transition function.

The principal differences among machines inhere in their storage. We shall consider two classes of storage: *k-dimensional tapes* and *k-regular trees*.

In a k-dimensional tape, the cells correspond to the elements of a free commutative group on k generators. There are $2k$ shifts, which correspond to addition of a generator or

its negative. It will often be convenient to include a *null shift*, which corresponds to the identity mapping from the cells to themselves.

In a k-regular tree, the cells correspond to the elements of a free group on k generators. There are $2k$ shifts, which correspond to premultiplication by a generator or its inverse. Again it will often be convenient to include a null shift. (Paterson, Fischer, and Meyer [7] have used k-regular trees as storage media with a restricted form of random access.)

For simplicity, we shall always assume that all the storage media of a machine are of the same type. Thus we shall speak of a k-dimensional machine (one having some number of k-dimensional tapes) or a k-regular machine (one having some number of k-regular trees). Note that a one-dimensional tape is equivalent to a one-regular tree, both being *sequential* storage media. A machine having some number of sequential storage media with just one head per medium will be called an *elementary machine.*

We shall say that two machines *simulate* each other if when they are started with the same string of symbols on their input tapes, they produce the same string of symbols on their output tapes. Clearly, simulation is an equivalence relation. We shall say that two machines that simulate each other do so *on-line* if the shifts of the input and output heads occur in the same order (but not necessarily at the same steps) for both machines This is also an equivalence relation. Finally, we shall say that one machine that simulates another does so in time $T(n)$ if, for every n, the one machine shifts its output head at least as many times during its first $T(n)$ steps as the other machine does during its first n steps.

THEOREM 1. *If $\mathscr{M}$ is a machine with k-dimensional tapes, there is an elementary machine $\mathscr{M}'$ that simulates $\mathscr{M}$ on-line in time $O(n^{2-1/k})$.*

PROOF. Suppose for now that $\mathscr{M}$ has just one k-dimensional storage tape $\mathscr{T}$, with just one head $\mathscr{H}$. The general case, with more than one storage tape and more than one head per tape, will be discussed later.

The construction of the machine $\mathscr{M}'$ will involve a number of different ideas. In order to introduce these ideas gradually and to show their interrelationships more clearly, we shall present three versions of $\mathscr{M}'$ of increasing intricacy, only the last of which will simulate $\mathscr{M}$ in time $O(n^{2-1/k})$. In the first two versions some things will be done by means that may seem more complicated than is necessary. These means have been chosen to simplify the transitions to later versions, and their justification should become apparent on further reading.

In order to further clarify the exposition, rather than describe $\mathscr{M}'$ formally in terms of "states" and "transition functions," we shall describe informally how a "general-purpose" machine $\mathscr{M}'$ with sufficiently many one-dimensional tapes can be "programmed" to simulate $\mathscr{M}$. In this description we shall use terms such as "subroutine" and "stack," which are common in programming. It should be clear how these informal ideas could be implemented in a formal model.

It will be necessary for $\mathscr{M}'$ to maintain a representation of the configuration of $\mathscr{M}$. Some aspects of this representation are obvious: $\mathscr{M}'$ will use its input and output tapes just as $\mathscr{M}$ uses its input and output tapes, and $\mathscr{M}'$ will use its control to simulate the control of $\mathscr{M}$. It remains for $\mathscr{M}'$ to maintain representations of the position of $\mathscr{H}$ and the condition of $\mathscr{T}$, and it is on these aspects of the representation that the remainder of the proof will focus.

REPRESENTING THE POSITION OF $\mathscr{H}$. The set of *displacements* of $\mathscr{H}$ forms a group in a natural way. This is a free commutative group on k generators, where the generators and their negatives correspond to the $2k$ shifts that $\mathscr{H}$ might execute An element of this group can be thought of as a list of k integers that gives the number (positive, negative, or zero) of steps to be taken in each of the k coordinate directions.

The position of $\mathscr{H}$ will be represented by representing the displacement from the *origin* (the initial location of $\mathscr{H}$) to the present location of $\mathscr{H}$. This displacement will in turn be represented by representing k integers. These integers will be represented on k tapes, called the *$\mathscr{H}$-position tapes*, according to the following scheme.

A *counter* is a stack which may be empty (to represent the integer zero), which may

contain some number of "+" symbols (to represent a positive integer), or which may contain some number of "−" symbols (to represent a negative integer). A counter may be *incremented* by pushing a "+" symbol onto the stack, unless there is a "−" symbol on the top of the stack, in which case this symbol is popped off the stack. Similarly, a counter may be *decremented*.

The position of $\mathscr{H}$ in the initial configuration of $\mathscr{M}$ will be represented by the null displacement, which in turn will be represented by empty stacks on the $\mathscr{H}$-position tapes.

THE SHIFT SUBROUTINE. Suppose that the position of $\mathscr{H}$ is represented on the $\mathscr{H}$-position tapes as described above. The task of the SHIFT subroutine will be to update this representation to reflect a shift of $\mathscr{H}$. It will do this by incrementing or decrementing the appropriate counter.

REPRESENTING THE CONDITION OF $\mathscr{T}$. The condition of $\mathscr{T}$ will be represented by means of a string, called a *path string*, which will be kept on a tape called the *$\mathscr{T}$-condition tape*. A path string will contain two types of symbols: There will be $2k$ symbols, called *shift symbols*, that represent the possible shifts that $\mathscr{H}$ might execute, and a set of symbols, called *metasymbols*, that represent the various symbols that might be written on a cell of $\mathscr{T}$.

The occurrence of a symbol in a path string will be said to *visit* a cell of $\mathscr{T}$ if by starting at the origin of $\mathscr{T}$ and by executing in sequence the shifts represented by all the shift symbols preceding the given occurrence, one arrives at the given cell.

A path string will represent the condition of $\mathscr{T}$ in the following way. The cell of $\mathscr{T}$ visited by an occurrence of a metasymbol is understood to contain the symbol represented by that metasymbol. (Each cell of $\mathscr{T}$ will be visited by at most one occurrence of a metasymbol.) A cell of $\mathscr{T}$ whose content is not represented in this way is understood to contain a blank.

The condition of $\mathscr{T}$ in the initial configuration of $\mathscr{M}$ will be represented by a path string consisting of a single metasymbol which visits the origin and represents a blank.

THE READ SUBROUTINE. Suppose that the position of $\mathscr{H}$ is represented by the $\mathscr{H}$-position tapes and the condition of $\mathscr{T}$ is represented on the $\mathscr{T}$-condition tape. The task of the READ subroutine is to determine the symbol scanned by $\mathscr{H}$.

The READ subroutine works as follows: It starts with the head on the $\mathscr{T}$-condition tape at the first symbol of the path string and moves the head over each symbol of the path string in sequence. During this process it maintains the validity of the following assertion: The $\mathscr{H}$-position tapes represent the displacement between the cell of $\mathscr{T}$ visited by the symbol of the path string currently scanned by the head on the $\mathscr{T}$-condition tape and the cell of $\mathscr{T}$ at which $\mathscr{H}$ is located. The assertion is true at the outset, since the first symbol of a path string visits the origin, and the $\mathscr{H}$-position tapes represent the displacement between the origin and the cell of $\mathscr{T}$ at which $\mathscr{H}$ is located. Furthermore, its validity will be maintained if as each shift symbol in the path string is passed, the negative of the shift it represents is applied to the $\mathscr{H}$-position tapes by means of the SHIFT subroutine.

If, during this process, a metasymbol is encountered when the $\mathscr{H}$-position tapes represent the null displacement (all stacks empty), this metasymbol represents the symbol scanned by $\mathscr{H}$. If no such metasymbol is encountered, a blank is scanned by $\mathscr{H}$. In either case, after the process is complete, it can be reversed to restore the $\mathscr{H}$-position tapes to their condition at the outset: The head on the $\mathscr{T}$-condition tape is moved backward over each symbol of the path string, and as each shift symbol is passed, the shift it represents is applied to the $\mathscr{H}$-position tapes.

THE WRITE SUBROUTINE. Suppose again that the position of $\mathscr{H}$ is represented on the $\mathscr{H}$-position tapes and the condition of $\mathscr{T}$ is represented on the $\mathscr{T}$-condition tape. The task of the WRITE subroutine is to update the path string to reflect a change in the symbol scanned by $\mathscr{H}$.

The WRITE subroutine works as follows: It begins by performing a forward-then-backward pass over the path string, similar to the pass performed by the READ subroutine. If during this pass a metasymbol is encountered when the $\mathscr{H}$-position tapes represent the

null displacement, changing this metasymbol will reflect the change in the symbol scanned by $\mathscr{H}$, and the task of the WRITE subroutine will be complete after this first pass. If no such metasymbol is encountered, a second forward-then-backward pass will be performed to insert one.

During this second pass the WRITE subroutine will search for a metasymbol that visits a cell of $\mathscr{T}$ that is *adjacent* to the cell at which $\mathscr{H}$ is located. (One cell is adjacent to another if $\mathscr{H}$ can move from one to the other by a single shift.) To do this, it searches for a situation in which the $\mathscr{H}$-position tapes represent a unit displacement (all stacks empty but one, which contains a single "+" or "−"). In such a situation, the unit displacement specifies the shift that would move $\mathscr{H}$ from the adjacent cell to the cell at which $\mathscr{H}$ is located. When it finds such a symbol, it inserts immediately after it a string of three symbols: a shift symbol that represents the shift that would move $\mathscr{H}$ from the adjacent cell to its current location, a metasymbol representing the symbol to be written at the current location, and a shift symbol representing the shift that would move $\mathscr{H}$ from its current location back to the adjacent cell.

It should be clear that if either the first pass finds a metasymbol that visits the current location or the second pass finds a symbol that visits a cell adjacent to the current location, then the path string on the $\mathscr{T}$-condition tape will be updated to reflect the change in the symbol scanned by $\mathscr{H}$. We cannot argue that either the first or the second pass will succeed until we have presented other parts of the program of $\mathscr{M}'$, but the argument will depend on only one fact: The program for $\mathscr{M}'$ will use the WRITE subroutine to update cells in the same order in which they are scanned by $\mathscr{H}$. Thus if the cell currently scanned by $\mathscr{H}$ has not previously been scanned (so that the first pass does not succeed), it must be adjacent to one that has (so that the second pass will).

THE STEP SUBROUTINE. At this point we can describe a subroutine, the STEP subroutine, that simulates a complete step of the computation of $\mathscr{M}$. As before, we asume that the position of $\mathscr{H}$ and the condition of $\mathscr{T}$ are represented on the $\mathscr{H}$-position and $\mathscr{T}$-condition tapes, respectively.

The STEP subroutine works as follows: It uses the READ subroutine to determine the symbol scanned by $\mathscr{H}$ and reads a symbol from the input tape (if necessary). It uses this information, together with the control state of $\mathscr{M}$, to determine (according to the transition function of $\mathscr{M}$) the new control state of $\mathscr{M}$ and the other actions to be taken. It then writes a symbol on the output tape (if necessary) and uses the WRITE subroutine to update the symbol scanned by $\mathscr{H}$. Finally, it uses the SHIFT subroutine to effect the appropriate shift of $\mathscr{H}$.

In all versions of $\mathscr{M}'$, all manipulation of the input and output tapes will be done by the STEP subroutine. For this reason, a simulation will always be an on-line simulation.

VERSION I OF $\mathscr{M}'$. At this point the main program for Version I of $\mathscr{M}'$ can be expressed as follows:

```
MAIN
    repeat indefinitely
      call STEP
```

That this program simulates $\mathscr{M}$ follows by induction on the following hypothesis:

After n invocations of the STEP subroutine,

(1) the counters on the $\mathscr{H}$-position tapes represent the position of $\mathscr{H}$ after n steps by $\mathscr{M}$;

(2) the path string on the $\mathscr{T}$-condition tape visits all and only those cells of $\mathscr{T}$ scanned by $\mathscr{H}$ during the first n steps by $\mathscr{M}$ (except possibly the last);

(3) the path string on the $\mathscr{T}$-condition tape represents the condition of $\mathscr{T}$ after n steps by $\mathscr{M}$.

THE RUNNING TIME OF VERSION I. We shall show that Version I of $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^2)$.[1]

[1] A word on our use of $O(\cdots)$ is in order. The constants implicit in this notation are to be understood to be uniform in time-dependent parameters such as n (and later m, j, and t), but not in machine-dependent parameters, such as k.

Consider the length of the path string that represents the condition of $\mathcal{T}$. Its length is initially one, and its length is extended by at most three for each execution of the WRITE subroutine, and hence for each execution of the STEP subroutine. Thus when $\mathcal{M}'$ has simulated n steps of $\mathcal{M}$, the path string will have length at most $3n + 1 = O(n)$.

The time taken by the nth execution of the STEP subroutine is dominated by the time taken by the READ and WRITE subroutines. This in turn is dominated by the time taken by the forward-then-backward passes over the path string, which is $O(n)$. From this, it follows that the time taken by the first n executions of the STEP subroutine is $O(n^2)$, so that $\mathcal{M}'$ simulates $\mathcal{M}$ in time $O(n^2)$.

THE STRATEGY OF VERSION II. Version I of $\mathcal{M}'$ runs as slowly as it does because as the path string that represents the condition of $\mathcal{T}$ grows longer, the time taken by the READ and WRITE subroutines, and hence by the STEP subroutine, increases. In order to mitigate this effect, Version II will exploit the following principle: At any point in the computation of $\mathcal{M}$, only the contents of those cells of $\mathcal{T}$ that $\mathcal{H}$ can reach by at most m shifts from its current position are relevant to the next m steps.

Consider the situation after n steps by $\mathcal{M}$. Let c denote the cell of $\mathcal{T}$ at which $\mathcal{H}$ is located and let m be a positive integer. There is a natural metric on the cells of $\mathcal{T}$: The distance between two cells is the minimum number of shifts sufficient to carry $\mathcal{H}$ from one of the cells to the other. Let $\mathcal{B}$ denote the *ball* with center c and radius m (this is the set of cells whose distance from c is at most m) and let $\mathcal{S}$ denote the *sphere* with center c and radius $m + 1$ (this is the set of cells whose distance from c is exactly $m + 1$). It is easy to show that the number of cells in $\mathcal{B}$ is $O(m^k)$, while the number of cells in $\mathcal{S}$ is $O(m^{k-1})$.

Suppose that $\mathcal{M}'$ has simulated the first n steps by $\mathcal{M}$, and consider how it might simulate the next m steps. Version I did this by applying the STEP subroutine m times to the path string that represents the condition of $\mathcal{T}$. Version II will use the following technique. By means of a subroutine called the EXTRACT subroutine, it will create from the path string that represents the condition of $\mathcal{T}$ two new path strings, one representing the condition of $\mathcal{B}$ and another representing the condition of $\mathcal{T} - \mathcal{B}$. It will then apply the STEP subroutine m times to the path string that represents the condition of $\mathcal{B}$. Finally, by means of a subroutine called the MERGE subroutine, it will combine the updated path string representing the condition of $\mathcal{B}$ with the path string representing the condition of $\mathcal{T} - \mathcal{B}$ to obtain an updated path string representing the condition of $\mathcal{T}$. The total effect of this will be the same as that of applying the STEP subroutine m times to the path string that represents the condition of $\mathcal{T}$.

If m is not too large, the path string representing the condition of $\mathcal{B}$ will be shorter than the path string representing the condition of $\mathcal{T}$, and the time taken by the m executions of the STEP subroutine will be reduced. If m is not too small, this reduction will outweigh the time taken by the EXTRACT and MERGE subroutines, yielding a net reduction in the time taken to simulate m steps.

To show how Version II of $\mathcal{M}'$ will use the technique just described to perform a complete simulation of $\mathcal{M}$, we shall give a "top-down" description of the process. The infinitely many steps by $\mathcal{M}$ will be divided into *phases* of increasing length. The zeroth phase will consist of the first step, the first phase will consist of the next 2^{k+1} steps, and so forth, with the jth phase consisting of $2^{j(k+1)}$ steps. Each phase will in turn be divided into *subphases* of equal length. The zeroth phase will be divided into one subphase of one step, the first phase will be divided into 2^k subphases of two steps each, and so forth, with the jth phase being divided into 2^{jk} subphases of 2^j steps each. The steps of each subphase will be simulated by the technique described previously, so that $m = 2^j$ for each subphase of the jth phase.

THE LOCAL COUNTERS. The path strings used to represent the condition of $\mathcal{T}$ do so by "relative addressing" with respect to the origin, in the sense that one starts from the origin to determine which cell of $\mathcal{T}$ a symbol visits. In order to ensure their shortness, the path strings used to represent the condition of $\mathcal{B}$ will do so by relative addressing with respect

to c, the center of $\mathscr{B}$. The path strings used to represent the condition of $\mathscr{T} - \mathscr{B}$, like those used to represent the condition of $\mathscr{T}$, will use relative addressing with respect to the origin

During a subphase the STEP subroutine must have access not only to the path string representing the condition of $\mathscr{B}$, but also to the counters representing the position of $\mathscr{H}$. Since the path string representing the condition of $\mathscr{B}$ uses relative addressing with respect to c, these counters must represent the displacement from c to the position of $\mathscr{H}$, not from the origin to the position of $\mathscr{H}$. To meet this need, a new set of counters will be introduced.

At the outset of a subphase, the EXTRACT subroutine will create a set of "local" counters that will represent the position of $\mathscr{H}$ relative to c. These counters will be initialized to represent the null displacement, since at the outset of the subphase, $\mathscr{H}$ is located at c. During the subphase the STEP subroutine will manipulate these local counters, not the "global" ones which represent the position of $\mathscr{H}$ relative to the origin. At the conclusion of the subphase, the MERGE subroutine will add the contents of the local counters into the global counters, thus updating the latter to reflect the shifts made by $\mathscr{H}$ during the subphase. The MERGE subroutine will then destroy the local counters.

THE SUBPHASE STACKS. During a subphase, Version II of $\mathscr{M}'$ must maintain two path strings and two sets of counters. To store these conveniently, the $\mathscr{T}$-condition tape and $\mathscr{H}$-position tapes will be used as stacks. The $\mathscr{T}$-condition tape will contain a stack of path strings: between subphases this stack will contain only the path string representing the condition of $\mathscr{T}$; at the outset of each subphase, the EXTRACT subroutine will replace this with the path strings representing the condition of $\mathscr{B}$ and the condition of $\mathscr{T} - \mathscr{B}$ (with the former at the top of the stack); at the end of each subphase the MERGE subroutine will replace these with the updated path string representing the condition of $\mathscr{T}$.

Similarly, each $\mathscr{H}$-position tape will contain a stack of counters[2] between subphases these stacks will contain only the global counters representing the position of $\mathscr{H}$ relative to the origin; at the outset of each subphase the local counters representing the position of $\mathscr{H}$ relative to c will be pushed onto the stacks by the EXTRACT subroutine; at the end of each subphase the local counters will be popped off the stacks by the MERGE subroutine. In this way, the path string and counters to be manipulated by the STEP subroutine will always be at the top of the stack

THE LENGTH COUNTERS. Just as there is a natural metric on the cells of $\mathscr{T}$, there is a natural norm on the group of displacements: The *length* of a displacement is the minimum number of shifts sufficient to effect it.

It will be convenient to associate with each set of counters that represents a displacement an additional counter that represents the length of the displacement. These counters will be kept on a new tape called the *length tape*, and when several displacements are stacked on the $\mathscr{H}$-position tapes, their lengths will be stacked (in the same order) on the length tape.

It is easy to see how the counters on the length tape can be maintained by the subroutines that manipulate the counters on the $\mathscr{H}$-position tapes. The SHIFT subroutine will increment a length counter whenever it pushes a "+" or "−" symbol onto one of the stacks that implement the associated displacement counters, and will decrement the length counter whenever it pops a "+" or "−" symbol off. When the EXTRACT subroutine creates the local displacement counters and initializes them to the null displacement, it will create a local length counter and initialize it to zero. Finally, when the MERGE subroutine adds the contents of the local displacement counters into the global displacement counters, it will use the SHIFT subroutine to manipulate the latter and thus will automatically update the global length counter.[3]

THE EXTRACT SUBROUTINE. Suppose that the condition of $\mathscr{T}$ is represented by a path string on the $\mathscr{T}$-condition tape, the position of $\mathscr{H}$ is represented by a displacement in

[2] Do not be confused by the fact that each counter is itself implemented as a stack

[3] Note that it would not be correct to add the local length counter into the global length counter, since the triangle inequality might not be an equality

the counters on the $\mathscr{H}$-position tapes, and the length of this displacement is represented in the counter on the length tape. Suppose further that the positive integer m is represented in a counter on yet another tape. The number m, together with the position of $\mathscr{H}$, determines the ball $\mathscr{B}$ and the sphere $\mathscr{S}$. The task of the EXTRACT subroutine is to replace the path string representing the condition of $\mathscr{T}$ with two path strings describing the conditions of $\mathscr{B}$ and $\mathscr{T} - \mathscr{B}$, and to create and initialize the new counters on the $\mathscr{H}$-position and length tapes.

The EXTRACT subroutine works as follows: During a forward-then-backward pass (similar to those used by the READ and WRITE subroutines) over the path string that represents the condition of $\mathscr{T}$, it creates the path strings that represent the conditions of $\mathscr{B}$ and $\mathscr{T} - \mathscr{B}$ on two work tapes. It then copies these onto the $\mathscr{T}$-condition tape in the appropriate order. The new counters are created on the $\mathscr{H}$-position and length tapes simply by writing new bottom-of-stack symbols on these tapes.

Let the path string that represents the condition of $\mathscr{T}$ be divided into "segments" (substrings) of two types: *inner segments*, consisting of symbols that visit cells in $\mathscr{B}$, and *outer segments*, consisting of symbols that visit cells in $\mathscr{T} - \mathscr{B}$. These two types of segments occur alternately. During the pass over the path string, the inner segments are copied onto one work tape, to form the path string representing the condition of $\mathscr{B}$, and the outer segments are copied onto another work tape, to form the path string representing the condition of $\mathscr{T} - \mathscr{B}$.

The inner and outer segments copied onto the work tapes do not by themselves form complete path strings, since the last symbol of a given segment need not visit the same (or an adjacent) cell as the first symbol of the next segment of the same type. To deal with this situation, we shall precede each such segment (except the first outer segment) and follow each such segment (except the last outer segment) with a sequence of shift symbols, which is called a *linking segment*, that would carry $\mathscr{H}$ from c to the first cell visited by the segment, or from the last cell visited by the segment to c. Since transitions between inner and outer segments occur when $\mathscr{H}$ moves between $\mathscr{B}$ and $\mathscr{T} - \mathscr{B}$ (which always occurs at distance m from c), no linking segment need have length greater than $m + 1$. It will be convenient to ensure that inner segments, outer segments, segments that link inner segments, and segments that link outer segments are mutually distinguishable. This can be done by a fourfold expansion of the alphabet of the $\mathscr{T}$-condition tape.

To carry out its operations, the EXTRACT subroutine needs only two abilities that we have not yet discussed: During its pass over the path string that represents the condition of $\mathscr{T}$, it must be able to recognize when a transition between an inner and an outer segment occurs, and it must be able to generate the appropriate linking segments. To obtain these abilities, it uses the length tape.

By using the same technique as was used by the READ and WRITE subroutines, the EXTRACT subroutine can ensure that during its pass over the path string that represents the condition of $\mathscr{T}$, the displacement from the cell being visited to the position of $\mathscr{H}$ is represented on the $\mathscr{H}$-position tapes, and thus that the length of this displacement is represented on the length tape. If before this pass is made, the number m is subtracted from the counter on the length tape, then during the pass the counter will alternate between positive and nonpositive values whenever the cell being visited alternates between $\mathscr{T} - \mathscr{B}$ and $\mathscr{B}$. In this way the EXTRACT subroutine can recognize transitions between inner and outer segments. (After the pass, the number m can be added back into the counter on the length tape.) Furthermore, whenever a transition occurs, the $\mathscr{H}$-position tapes will represent the displacement from the cell being visited to c. From this, the EXTRACT subroutine can generate the required linking segments.

THE MERGE SUBROUTINE. The task of the MERGE subroutine is the inverse of that of the EXTRACT subroutine. It copies the path string that represents the condition of $\mathscr{B}$ from the $\mathscr{T}$-condition tape onto a work tape, and the path string that represents the condition of $\mathscr{T} - \mathscr{B}$ from the $\mathscr{T}$-condition tape onto another tape. It then scans both work

tapes, copying the inner and outer segments alternately onto the $\mathscr{T}$-condition tape to form a path string that represents the condition of $\mathscr{T}$, and discarding the linking segments.

In addition, it merges the counters on the $\mathscr{H}$-position and length tapes. It adds the displacement from c to the position of $\mathscr{H}$ to the displacement from the origin to c, to obtain the displacement from the origin to the position of $\mathscr{H}$. As it does so, it updates the length of the latter displacement.

VERSION II OF $\mathscr{M}'$. At this point, the program for Version II of $\mathscr{M}'$ can be expressed as follows:

```
MAIN.
    repeat indefinitely for j = 0, 1, 2,
      call PHASE (j)
PHASE(j).
    repeat 2^(jk) times
      call SUBPHASE(j)
    return
SUBPHASE(j)
    call EXTRACT with m = 2^j
    repeat 2^j times
      call STEP
    call MERGE
    return
```

That this program simulates $\mathscr{M}$ can be confirmed by verifying the following hypothesis.

The execution of a subphase of the jth phase is equivalent in effect to execution of the STEP subroutine 2^j times.

THE RUNNING TIME OF VERSION II. We shall show that Version II of $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^{2-1/(k+1)})$. We begin with a crucial observation: The path string that represents the condition of $\mathscr{T}$ visits any cell at most $2k + 2$ times. This is verified by considering how the path string is developed by the WRITE subroutine. A cell is visited twice when it is first scanned by $\mathscr{H}$: once by a metasymbol and once by a shift symbol. After this, it is revisited (by a shift symbol) only when an adjacent cell is first scanned by $\mathscr{H}$. Since there are only $2k$ adjacent cells, there can be at most $2k + 2$ visits. This observation also applies to the inner segments of the path string that represents the condition of $\mathscr{B}$, since they are developed in the same way.

At the end of the jth phase, the STEP subroutine will have been executed

$$1 + 2^{k+1} + \cdots + 2^{j(k+1)} = O(2^{j(k+1)})$$

times. Thus the path string that represents the condition of $\mathscr{T}$ will have length $O(2^{j(k+1)})$ until that time.

During any subphase of the jth phase, $m = 2^j$, so $\mathscr{B}$ contains $O(m^k) = O(2^{jk})$ cells. Each of these cells is visited at most $2k + 2$ times by the inner segments of the path string that represents the condition of $\mathscr{B}$, so the total length of these inner segments is at most $O(2^{jk})$. Furthermore, each linking segment in the path string that represents the condition of $\mathscr{B}$ was generated because of a visit to a cell in $\mathscr{S}$ by the path string that represents the condition of $\mathscr{T}$. $\mathscr{S}$ contains $O(m^{k-1}) = O(2^{j(k-1)})$ cells, and each of these is visited at most $2k + 2$ times; so there are at most $O(2^{j(k-1)})$ linking segments. Since each linking segment has length at most $m + 1 = O(2^j)$, the total length of these linking segments is at most $O(2^{jk})$. Thus during the jth phase, the path string that represents the condition of $\mathscr{B}$ will have length $O(2^{jk})$.

Consider the time taken by a subphase of the jth phase. The time taken by the EXTRACT subroutine is dominated by the time taken to pass over the path string that represents the condition of $\mathscr{T}$, which is $O(2^{j(k+1)})$. The time taken by each execution of the STEP subroutine is dominated by the time taken by the READ and WRITE subroutines to pass over the path string that represents the condition of $\mathscr{B}$, which is $O(2^{jk})$. Thus the

time taken by all 2^j executions of the STEP subroutine is $O(2^{j(k+1)})$. The time taken by the MERGE subroutine is dominated by the time taken to regenerate the path string that represents the condition of $\mathcal{T}$, which is $O(2^{j(k+1)})$. Thus the total time taken by a subphase of the jth phase is $O(2^{j(k+1)})$.

The jth phase consists of 2^{jk} subphases; so the time taken by the jth phase is $O(2^{j(2k+1)})$. From this it follows that the time taken by the first $j + 1$ phases is also $O(2^{j(2k+1)})$. The nth step of $\mathcal{M}$ is simulated during the first $j + 1$ phases, where $j = \lceil(\log_2 n)/(k + 1)\rceil$, since $2^{j(k+1)} \geq n$ steps of $\mathcal{M}$ are simulated in the jth phase alone. From this it follows that $\mathcal{M}'$ simulates $\mathcal{M}$ in time $O(n^{(2k+1)/(k+1)}) = O(n^{2-1/(k+1)})$.

THE STRATEGY OF VERSION III. Version II of $\mathcal{M}'$ runs as slowly as it does because the path strings manipulated by the STEP subroutine still grow, though not as rapidly as in Version I. To further mitigate this effect and, in fact, obtain the best possible running time, Version III will use the techniques of Version II recursively, in such a way that the path strings manipulated by the STEP subroutine are uniformly bounded in length.

To show how Version III of $\mathcal{M}'$ works, we shall again give a top-down description. The infinitely many steps by $\mathcal{M}$ will be divided into *phases* of increasing length. The zeroth phase will consist of the first step, the first phase will consist of the next 2^k steps, and so forth, with the jth phase consisting of 2^{jk} steps. Each phase will in turn be divided into *zeroth-level subphases* of equal length. The zeroth phase will be divided into one zeroth-level subphase of one step, the first phase will be divided into 2^{k-1} zeroth-level subphases of two steps each, and so forth, with the jth phase being divided into $2^{j(k-1)}$ zeroth-level subphases of 2^j steps each. Each zeroth-level subphase (except for the one of the zeroth phase) will be divided into two *first-level subphases*, each having half as many steps. This process of subdivision will continue, with each ith-level subphase (except for those in the ith phase) being divided into two $i + 1$st-*level subphases*, each having half as many steps. In the jth phase, this process terminates with the jth-level subphases, which have one step each.

RECURSIVE APPLICATION OF THE EXTRACT AND MERGE SUBROUTINES. In a zeroth-level subphase of the jth phase ($j > 0$), the EXTRACT subroutine will "split" the path string representing the condition of $\mathcal{T}$ into path strings representing the conditions of $\mathcal{B}_0$ and $\mathcal{T} - \mathcal{B}_0$, where $\mathcal{B}_0$ is a ball of radius 2^j whose center is the position of $\mathcal{H}$ at the outset of the subphase. This activity is the same as that performed by Version II.

In an ith-level subphase of the jth phase ($0 < i < j$), the EXTRACT subroutine will split the path string representing the condition of $\mathcal{B}_{i-1}$ into path strings representing the conditions of $\mathcal{B}_i$ and $\mathcal{B}_{i-1} - \mathcal{B}_i$, where $\mathcal{B}_i$ is a ball of radius 2^{j-i} whose center is the position of $\mathcal{H}$ at the outset of the subphase. It is easy to see that $\mathcal{B}_i$ is contained in $\mathcal{B}_{i-1}$. The EXTRACT subroutine will form the path string that represents the condition of $\mathcal{B}_i$ by taking selected portions of the inner segments from the path string that represents the condition of $\mathcal{B}_{i-1}$ and joining them together with linking segments of the type that were used to join inner segments at the zeroth level. It will form the path string that represents the condition of $\mathcal{B}_{i-1} - \mathcal{B}_i$ by filling the gaps between the selected portions with linking segments of the type that were used to join outer segments at the zeroth level.

As before, the task of the MERGE subroutine will be the inverse of that of the EXTRACT subroutine. It will combine the updated information in the path string representing the condition of $\mathcal{B}_i$ with the information in the path string representing the condition of $\mathcal{B}_{i-1} - \mathcal{B}_i$ to obtain an updated path string representing the condition of $\mathcal{B}_{i-1}$. It is easy to see that since the various types of segments are distinguishable, the MERGE subroutine can discard the appropriate linking segments and correctly interleave the segments they separate.

To keep track of the various path strings, the EXTRACT and MERGE subroutines will use the $\mathcal{T}$-condition tape as a stack, as previously described. The EXTRACT subroutine will replace the path string on top of the stack by the two path strings it produces, while the MERGE subroutine will replace the two path strings on top of the stack by the one it

produces. As before, each path string on the $\mathscr{T}$-condition tape will be associated with a set of counters on the $\mathscr{H}$-position tapes and a counter on the length tape, and the path string and counters to be manipulated by the STEP subroutine will always be on the top of the stack.

VERSION III OF $\mathscr{M}'$. At this point, the program for Version III of $\mathscr{M}'$ can be expressed as follows:

```
MAIN
    repeat indefinitely for j = 0, 1, 2,
      call PHASE(j)
PHASE(j)
    repeat 2^{j(k-1)} times
      call SUBPHASE (0, j)
    return
SUBPHASE (i, j)
    if i = j then call STEP
          else begin
              call EXTRACT with m = 2^{j-i}
              repeat 2 times
                call SUBPHASE (i + 1, j)
              call MERGE
              end
    return
```

That this program simulates $\mathscr{M}$ can be confirmed by verifying the following hypothesis, using induction on i (starting at $i = j$ and working down to $i = 0$).

The execution of an ith-level subphase of the jth phase is equivalent in effect to execution of the STEP subroutine 2^{j-i} times.

THE RUNNING TIME OF VERSION III. We shall show that Version III of $\mathscr{M}'$ simulates $7\mathscr{M}$ in time $O(n^{2-1/k})$.

At the end of the jth phase, the STEP subroutine will have been executed

$$1 + 2^k + \cdots + 2^{jk} = O(2^{jk})$$

times. Thus the path string that represents the condition of $\mathscr{T}$ will have length $O(2^{jk})$ until that time.

During an ith-level subphase of the jth phase, $m = 2^{j-i}$; so the path string that represents the condition of $\mathscr{B}_i$ will have length $O(2^{(j-i)k})$. (This is obtained by bounding the lengths of the inner segments and linking segments as before.)

Consider the execution of a zeroth-level subphase of the jth phase, together with all its recursive subphases. There will be 2^i ith-level subphases: The number of subphases increases by a factor of 2 at each successive level. We have just seen, however, that the length of the path strings decreases by a factor of $2^k \geq 4$ at each level. Thus the total time taken by executions of the EXTRACT and MERGE subroutines is dominated by the time taken at the zeroth level, which is $O(2^{jk})$. Aside from the executions of the EXTRACT and MERGE subroutines, there are 2^j executions of the STEP subroutine These take time $O(2^j)$, since they all occur at the jth level, where the path strings have length $O(1)$. Thus the total time taken by a zeroth-level subphase of the jth phase is $O(2^{jk})$.

The jth phase consists of $2^{j(k-1)}$ zeroth-level subphases, so the time taken by the jth phase is $O(2^{j(2k-1)})$ From this it follows that the time taken by the first $j + 1$ phases is also $O(2^{j(2k-1)})$. The nth step of $\mathscr{M}$ is simulated during the first $j + 1$ phase, where $j = \lceil(\log_2 n)/k\rceil$, since $2^{jk} \geq n$ steps of $\mathscr{M}$ are simulated in the jth phase alone From this it follows that $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^{(2k-1)/k}) = O(n^{2-1/k})$.

If $\mathscr{M}$ has more than one tape, the machinery described above can be replicated for each tape, and the activities described can be carried out for all the tapes simultaneously. If $\mathscr{M}$ has more than one head per tape, it can be replaced without loss of time by a machine with a larger number of tapes, but only one head per tape (see Leong and Seiferas [5]).

This completes the proof of Theorem 1.

In [3], Hennie gives (for each $k \geq 2$) a k-dimensional machine $\mathcal{M}$ with the following property: Any elementary machine $\mathcal{M}'$ that simulates $\mathcal{M}$ on-line takes time at least proportional to $n^{2-1/k}$ to do so. This shows that Theorem 1 gives the best possible result for on-line simulation.

THEOREM 2. *If $\mathcal{M}$ is a k-regular machine, there is an elementary machine $\mathcal{M}'$ that simulates $\mathcal{M}$ on-line in time $O(n^2/\log n)$.*

PROOF. As in the proof of Theorem 1, we suppose that $\mathcal{M}$ has just one k-regular tree $\mathcal{T}$, with just one head $\mathcal{H}$. If $\mathcal{M}$ has more than one tree or more than one head per tree, this can be dealt with as in Theorem 1 (the results of Leong and Seiferas [5] for multidimensional tapes apply to trees as well; it may be necessary to increase k, but this will not affect our result).

The machine that we construct to prove this theorem will be very similar to Version II of the machine constructed to prove Theorem 1 and will differ only in the following two respects:

(1) The representation of the position of $\mathcal{H}$ and the SHIFT subroutine will be different.

(2) The division of the steps into phases and subphases will be different.

Only these differences will be dealt with explicitly.

REPRESENTING THE POSITION OF $\mathcal{H}$. Whereas the group of displacements on a k-dimensional tape was a free commutative group on k generators, the group of displacements on a k-regular tree-structured storage medium is a free group on k generators, without the commutation relations. Again the generators and their inverses correspond to the $2k$ shifts that $\mathcal{H}$ might execute. An element of this group can be thought of as a word, over an alphabet of $2k$ symbols representing these shifts, that specifies the sequence of shifts to be taken.

The position of $\mathcal{H}$ will be represented by the displacement from the origin to the present location of $\mathcal{H}$. This displacement will in turn be represented by its word, on a tape called the *$\mathcal{H}$-position tape*. This word can be thought of as forming a stack, with the first symbol (representing the first shift away from the origin) at the bottom of the stack and the last symbol at the top of the stack.

The position of $\mathcal{H}$ in the initial configuration will be represented by the identity displacement, which in turn will be represented by the null word, which in turn will be represented by an empty stack on the $\mathcal{H}$-position tape.

THE SHIFT SUBROUTINE. Suppose that the position of $\mathcal{H}$ is represented on the $\mathcal{H}$-position tape as described above. The task of the SHIFT subroutine will be to update this representation to reflect a shift of $\mathcal{H}$. It will do this by pushing a symbol representing this shift onto the stack, unless there is a symbol representing the inverse shift on the top of the stack, in which case this symbol is popped off the stack.

THE READ, WRITE, EXTRACT, AND MERGE SUBROUTINES. Only minor changes are mandated by our change in the representation of the position of $\mathcal{H}$.

In the READ, WRITE, and EXTRACT subroutines, the $\mathcal{H}$-position tape is used to keep track of the displacement from the cell of $\mathcal{T}$ visited by the symbol of the path string currently scanned by the head on the $\mathcal{T}$-condition tape and the cell of $\mathcal{T}$ at which $\mathcal{H}$ is located. This can be done by pushing symbols onto and popping them off the stack if the word on the stack is reversed during the scan over the path string. This has no effect on our time bounds, since it is easy to obtain bounds on the length of these words that are as tight as the bounds on the length of the corresponding path strings.

At the conclusion of a subphase, the MERGE subroutine will "multiply" the "local word" representing the displacement from c to the position of $\mathcal{H}$ with the "global word" representing the displacement from the origin to c, to obtain the displacement from the origin to the position of $\mathcal{H}$. This is done simply by concatenation and cancellation.

THE OVERALL ORGANIZATION OF $\mathcal{M}'$. The top-down description of the process is as follows. The infinitely many steps by $\mathcal{M}$ will be divided into *phases* of increasing length.

The first phase will consist of the first $2k$ steps, the second phase will consist of the next $2(2k)^2$ steps, and so forth, with the jth phase consisting of $j(2k)^j$ steps. Each phase will in turn be divided into *subphases* of equal length. The first phase will be divided into $2k$ subphases of one step each, the second phase will be divided into $(2k)^2$ subphases of two steps each, and so forth, with the jth phase being divided into $(2k)^j$ subphases of j steps each.

Given the changes to the STEP subroutine (and its constituent subroutines) already described, the program for $\mathscr{M}'$ can be expressed as follows:

```
MAIN
    repeat indefinitely for j = 1, 2, 3,
      call PHASE(j)
PHASE(j)
    repeat (2k)^j times
      call SUBPHASE(j)
    return
SUBPHASE(j)
    call EXTRACT with m = j
    repeat j times
      call STEP
    call MERGE
    return
```

THE RUNNING TIME OF $\mathscr{M}'$. We shall show that $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^2/\log n)$. The argument closely parallels that used for Version II of $\mathscr{M}'$ in Theorem 1, with the following important difference: The number of cells in $\mathscr{B}$ and $\mathscr{S}$ each are $O((2k-1)^m)$.

At the end of the jth phase, the STEP subroutine will have been executed

$$2k + 2(2k)^2 + \cdots + j(2k)^j = O(j(2k)^j)$$

times. Thus the path string that represents the condition of $\mathscr{T}$ will have length $O(j(2k)^j)$ until that time.

During any subphase of the jth phase, $m = j$, so $\mathscr{B}$ contains $O((2k-1)^m) = O((2k-1)^j)$ cells. Each of these cells is visited at most $2k+2$ times by the inner segments of the path string that represents the condition of $\mathscr{B}$, so the total length of these segments is at most $O((2k-1)^j)$. Furthermore, since $\mathscr{S}$ contains $O((2k-1)^j)$ cells and each of these cells is visited at most $2k+2$ times by the path string that represents the condition of $\mathscr{T}$, there are at most $O((2k-1)^j)$ linking segments. Since each linking segment has length at most $m+1 = O(j)$, the total length of these linking segments is at most $O(j(2k-1)^j)$. Thus during the jth phase, the path string that represents the condition of $\mathscr{B}$ will have length $O(j(2k-1)^j)$.

Consider the time taken by a subphase of the jth phase. The time taken by the EXTRACT subroutine is dominated by the time taken to pass over the path string that represents the condition of $\mathscr{T}$, which is $O(j(2k)^j)$. The time taken by each execution of the STEP subroutine is dominated by the time taken to pass over the path string that represents the condition of $\mathscr{B}$, which is $O(j(2k-1)^j)$. Thus the time taken by all j executions of the STEP subroutine is $O(j^2(2k-1)^j)$, which is $O(j(2k)^j)$. The time taken by the MERGE subroutine is dominated by the time taken to regenerate the path string that represents the condition of $\mathscr{T}$, which is $O(j(2k)^j)$. Thus the total time taken by a subphase of the jth phase is $O(j(2k)^j)$.

The jth phase consists of $(2k)^j$ subphases, so the time taken by the jth phase is $O(j(2k)^{2j})$. From this it follows that the time taken by the first j phases is also $O(j(2k)^{2j})$. The nth step of $\mathscr{M}$ is simulated during the first j phases where $j = \lceil \log_{2k}(n/\log_{2k}(n/\log_{2k} n))\rceil$, since $j(2k)^j \geq n$ steps of $\mathscr{M}$ are simulated during the jth phase alone. From this it follows that $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^2/\log n)$.

This completes the proof of Theorem 2.

A simple modification of an example given by Hennie in [3] gives (for any $k \geq 2$) a k-regular machine $\mathcal{M}$ with the following property: Any elementary machine $\mathcal{M}'$ that simulates $\mathcal{M}$ on-line takes time at least proportional to $n^2/\log n$ to do so. We shall describe $\mathcal{M}$ and sketch a proof of this property. A rigorous proof can be obtained by the methods of [3].

$\mathcal{M}$ will have one k-regular tree $\mathcal{T}$ with head $\mathcal{H}$. It will have $2k + 3$ input symbols ($2k$ *shift symbols* corresponding to the shifts $\mathcal{H}$ might execute, together with 0, 1, and blank), three output symbols (0, 1, and blank), and a trivial control (having but a single state and thus no "memory"). At each step $\mathcal{M}$ reads an input.

(1) If the input is a shift symbol, $\mathcal{T}$ will be left as it is, $\mathcal{H}$ will execute the corresponding shift, and a blank output will be written.

(2) If the input is a 0 or 1, this symbol will be written on $\mathcal{T}$ (in the cell at which $\mathcal{H}$ is located), $\mathcal{H}$ will be left where it is, and a blank output will be written.

(3) If the input is a blank, $\mathcal{T}$ will be left as it is, $\mathcal{H}$ will be left where it is, and the symbol scanned by $\mathcal{H}$ will be written as output.

$\mathcal{M}$ is a machine that "exercises" its storage medium in a general way.

To see that any elementary machine $\mathcal{M}'$ that simulates $\mathcal{M}$ on-line takes time at least proportional to $n^2/\log n$, consider the following ensemble of input sequences. Each input sequence will be divided into two parts of approximately equal length. The first part will be devoted to shifting and writing (that is, will consist of shift symbols, 0's, and 1's), while the second part will be devoted to shifting and reading (that is, will consist of shift symbols and blanks).

During the first part, $\mathcal{H}$ will scan in turn all the cells of some ball centered at the origin, returning to the origin at the conclusion of its tour. As it scans each cell for the first time, it will write either a 0 or a 1 in it. If the input is of length n, the number of different cells scanned will be at least proportional to n, and the distance from the origin of each cell scanned will be at most proportional to $\log n$.

The second part of each input sequence will consist of a number of "questions" about the contents of the cells scanned during the first part. Each question will consist of a sequence of shift symbols that carries $\mathcal{H}$ to one of these cells, a blank (which "reads out" the content of the cell), and a sequence of shift symbols that carries $\mathcal{H}$ back to the origin. Each question will have length at most proportional to $\log n$; so there will be time for a number of questions at least proportional to $n/\log n$.

Now let $\mathcal{M}'$ be an elementary machine that simulates $\mathcal{M}$ on-line. When confronted with input sequences from the ensemble described above, $\mathcal{M}'$ must represent the condition of $\mathcal{T}$ at the conclusion of the first part. This representation will occupy a number of cells at least proportional to n on the one-dimensional tapes of $\mathcal{M}'$. Thus during the second part, each question in turn can be chosen so that $\mathcal{M}'$ must spend time at least proportional to n to find the answer. Since the number of questions is at least proportional to $n/\log n$, $\mathcal{M}'$ must spend time at least proportional to $n^2/\log n$ to find the answers. This shows that Theorem 2 gives the best possible result for on-line simulation.

3. *Simulation by Oblivious Machines and Combinational Logic Networks*

Our goal in this section is to study the relationship between the time required by a machine and the cost and delay required by a combinational logic network. We begin, however, with the study of a specially restricted machine model.

OBLIVIOUS MACHINES. Consider a machine with a one-dimensional read-only input tape, a one-dimensional write-only output tape, and some number of one-dimensional read/write storage tapes. We shall say that such a machine is *oblivious* if the movements of the input, output, and storage heads are fixed functions of time, independent of the input to the machine. (One may think of the head movements as being controlled by a second, autonomous machine which has one-dimensional storage tapes but no input or output tapes. Susceptibility to such a decomposition could be made the basis of an alternative definition of obliviousness which would be structural rather than behavioral.)

Although we have defined obliviousness only for machines with one-dimensional tapes, the same definition can obviously be applied to machines having nonsequential storage structures, such as those considered in Section 2. Indeed, a similar definition could be framed for random-access machines, requiring that the sequence of instructions followed and the sequence of storage locations accessed each be independent of the input.

The notion of an oblivious machine is of interest for a number of reasons. First, just as a machine model provides a certain formalization of the idea of an algorithm, the notation of an oblivious machine provides a certain formalization of the idea of an oblivious algorithm. For example, a table look-up by sequential search on a random-access machine may be programmed obliviously (if the search does not stop when the desired record is found but continues through the entire table), but a binary search cannot be, since the number of records examined is small compared to the entire table, and which records are examined depends on which record is sought. Second, when obtaining lower bounds on the complexity of functions, it is easier to consider oblivious machines than nonoblivious machines, since their behavior is simpler. With the aid of an efficient procedure whereby an oblivious machine can simulate a nonoblivious one, lower bounds on the complexity of functions for nonoblivious machines can be obtained. Third, the computations of oblivious machines can be implemented by combinational logic networks of low cost: A computation that can be performed in n steps by an oblivious machine can be implemented by a network with cost $O(n)$. Conversely, there are fast oblivious machines corresponding to many network constructions. Statements about oblivious machines, however, involve a uniformity over input lengths that is not present in the network context.

TRANSDUCERS. We shall say that a machine (not necessarily oblivious) is a *transducer* if it reads an input and writes an output at every step. Any machine can be replaced without loss of time by a transducer that performs essentially the same computation (this may involve simulating a queue to buffer the inputs [2, 5] and doing some trivial recoding of the outputs). Since transducers are already oblivious with respect to input–output behavior, the task of simulating them by oblivious machines will be technically simplified.

Results such as those we prove below can be obtained for machines other than transducers The results for transducers display all the essential ideas, however, and once these ideas are mastered, the other variants can be derived at will.

THEOREM 3. *If $\mathscr{M}$ is a transducer with one-dimensional tapes, there is an oblivious machine $\mathscr{M}'$ with two one-dimensional tapes that simulates $\mathscr{M}$ on-line in time $O(n \log n)$.*

PROOF. As in the proof of Theorem 1, we suppose that $\mathscr{M}$ has just one storage tape $\mathscr{T}$, with just one head $\mathscr{H}$. The general case will be discussed later.

We shall present two versions of $\mathscr{M}'$, only the second of which will simulate $\mathscr{M}$ in time $O(n \log n)$.

REPRESENTING THE CONDITION OF $\mathscr{M}$ AND THE POSITION OF $\mathscr{H}$. $\mathscr{M}'$ will use one storage tape $\mathscr{T}'$ with one storage head $\mathscr{H}'$ to maintain its representation of the condition of $\mathscr{T}$, the position of $\mathscr{H}$, and the number of steps by $\mathscr{M}$ that have been simulated. $\mathscr{T}'$ will contain, cell for cell, the same sequence of symbols as $\mathscr{T}$. In addition, through an enlargement of the alphabet of $\mathscr{T}'$ over that of $\mathscr{T}$, the positions of the following markers will be maintained:

(1) There will be an *origin marker* at the origin, this marker will never be moved.

(2) There will be a *head position marker* at the cell corresponding to the cell of $\mathscr{T}$ at which $\mathscr{H}$ is positioned.

(3) There will be *left* and *right end markers*, if n steps by $\mathscr{M}$ have been simulated, these markers will be positioned $n + 1$ cells to the left and right, respectively, of the origin.

THE STEP SUBROUTINE Suppose that the condition of $\mathscr{T}$, the position of $\mathscr{H}$, and the number of steps by $\mathscr{M}$ that have been simulated are represented on $\mathscr{T}'$ as described above. The task of the STEP subroutine will be to simulate the $n + 1$st step by $\mathscr{M}$ and to update $\mathscr{T}'$ to maintain this representation.

We shall assume that $\mathscr{M}'$ remembers, in its control, the symbol in the cell of $\mathscr{T}'$ at which

the head position marker is located; this remembered symbol is updated every time $\mathscr{H}'$ scans the head position marker.

The STEP subroutine works as follows: An input symbol is read from the input tape; this, together with the remembered symbol, determines the output symbol to be written on the output tape, the storage symbol to be written by $\mathscr{H}$ on $\mathscr{T}$, and the shift that $\mathscr{H}$ is to undergo. Then $\mathscr{H}'$ moves left from the origin marker to the left end marker, which it moves one position further to the left, then moves right past the origin marker to the right end marker, which it moves one position further to the right, and finally moves left again to the origin marker. During this excursion, $\mathscr{H}'$ scans the head position marker twice. once while passing from left to right and once while passing from right to left. Thus $\mathscr{M}'$ is able to update the symbol in the cell at which the head position marker is located and to shift the head position marker left or right, as appropriate.

VERSION I OF $\mathscr{M}'$. At this point the main program for Version I of $\mathscr{M}'$ can be expressed as follows:

```
MAIN
    repeat indefinitely
      call STEP
```

That this program simulates $\mathscr{M}$ follows by a simple induction.

THE RUNNING TIME OF VERSION I. We shall show that Version I of $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^2)$.

Consider the distance between the two end markers. After n steps by $\mathscr{M}$ have been simulated, the distance is $2n + 1 = O(n)$.

The time taken by the nth execution of the STEP subroutine is dominated by the time taken by the two excursions to the end markers, which is $O(n)$. From this it follows that the time taken by the first n executions of the STEP subroutine is $O(n^2)$, so that $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n^2)$.

THE STRATEGY OF VERSION II. Version I of $\mathscr{M}'$ runs as slowly as it does because as the distance between the end markers grows, the time taken by the STEP subroutine to visit the head position marker, which may be anywhere between them, increases. In order to mitigate this effect, Version II will periodically reorganize the information on $\mathscr{T}'$ to ensure that whenever the STEP subroutine is executed, the end markers and the head position marker will be near the origin marker. Since $\mathscr{H}$ will not always be near the origin of $\mathscr{T}$, these reorganizations will have to shift the information in the neighborhood of the head position marker to the neighborhood of the origin marker, relying on the "translation invariance" of the action of $\mathscr{M}$ for justification.

The top-down description of the process is as follows: The infinitely many steps by $\mathscr{M}$ will be divided into *phases* of increasing length. The zeroth phase will consist of the first step, the first phase will consist of the next two steps, and so forth, with the jth phase consisting of 2^j steps. Each phase (except for the zeroth) will in turn be divided into two *zeroth-level subphases*, each having half as many steps. Each zeroth-level subphase (except for those of the first phase) will be divided into two *first-level subphases*, each having half as many steps. This process of subdivision will continue, with each ith-level subphase (except for those of the $i + 1$st phase) being divided into two $i + 1$st-*level subphases*, each having half as many steps. In the jth phase, this process terminates with the jth-level subphases, which have one step each.

THE AUXILIARY TAPE. In addition to $\mathscr{T}'$, Version II of $\mathscr{M}'$ will use an auxiliary storage tape $\mathscr{T}''$. This auxiliary tape will be used for three purposes.

First, we shall often want to move a marker on $\mathscr{T}'$ so as to double or halve its distance from another marker; here $\mathscr{T}''$ will serve as a measuring tape on which a distance (or, by counting two against one, half a distance) can be marked off. Second, we shall often want to copy strings of symbols from one part of $\mathscr{T}'$ to another, here $\mathscr{T}''$ will serve as a buffer, so that the copying operation can be performed in time proportional to the sum of the

length of the string and the distance it is moved (rather than their product, as would be the case without a buffer). Third, our simulation procedure will be recursive; $\mathscr{T}''$ will serve as a stack for preserving information across recursive invocations. Stacking and unstacking will never be called for during measuring and copying operations, so the latter can be performed on top of the stack.

THE COMPRESS(j) SUBROUTINE. Suppose that the condition of $\mathscr{T}$, the position of $\mathscr{H}$, and the number n of steps by $\mathscr{M}$ that have been simulated are represented on $\mathscr{T}'$, as described in Version I, and suppose further that $n = 2^j - 1$ for some natural number j. The task of the COMPRESS(j) subroutine will be to shift the information in the neighborhood of the head position marker nearer to the origin marker and to halve the distance between the end markers.

The COMPRESS(j) subroutine works as follows: The contents of $\mathscr{T}'$ in the ball (interval) of radius $2^{j+1} - 1$ centered at the origin, including the symbols from the alphabet of $\mathscr{T}$ and the head position marker, but excluding the origin marker and end markers, are cyclically shifted 2^{j-1} positions. If the head position marker lies to the left of the origin marker, the information listed is cyclically shifted to the right; if the head position marker lies to right of the origin marker, the information listed is cyclically shifted to the left. In order to be oblivious, $\mathscr{M}'$ performs the head movements appropriate to both shifts, but only writes during one of them. A record of which shift was performed is placed on the stack. The left and right end markers are moved 2^{j-1} positions to the right and left, respectively. At this point the situation is analogous to the one when $n = 2^{j-1} - 1$, except that there is some information on $\mathscr{T}'$ lying beyond the end markers.

THE EXPAND(j) SUBROUTINE. The task of the EXPAND(j) subroutine is the inverse of that of the COMPRESS(j) subroutine. Using the record left on the stack by the COMPRESS(j) subroutine, it performs the reverse cyclic shift; the record is removed from the stack. The left and right end markers are moved 2^{j-1} positions to the left and right, respectively.

VERSION II OF $\mathscr{M}'$. At this point, the program for Version II of $\mathscr{M}'$ can be expressed as follows:

```
MAIN
    repeat indefinitely for j = 0, 1, 2,
      call PHASE(j)
PHASE(j)
    if j = 0 then call STEP
          else begin
               call COMPRESS(j)
               call PHASE(j - 1)
               call COMPRESS(j)
               call PHASE(j - 1)
               call EXPAND(j)
               call EXPAND(j)
               end
    return
```

That this program simulates $\mathscr{M}$ can be confirmed by verifying the following hypothesis, using induction on j:

If the end markers lie 2^j cells to either side of the origin marker, then the execution of the PHASE(j) subroutine, as regards the information between the end markers, is equivalent in effect to the execution of the STEP subroutine 2^j times. (In particular, PHASE(j) moves the end markers 2^j cells further from the origin marker.)

THE RUNNING TIME OF VERSION II. We shall show that Version II of $\mathscr{M}'$ simulates $\mathscr{M}$ in time $O(n \log n)$.

Whenever PHASE(j) is invoked, the end markers are at distance 2^j from the origin marker. The STEP subroutine is only invoked during executions of PHASE(0); so whenever the STEP subroutine is invoked, the end markers are at distance one from the

origin marker. Each execution of the STEP subroutine thus takes time $O(1)$.

The time taken by executions of the COMPRESS(j) and EXPAND(j) subroutines is dominated by the time taken to perform the cyclic shifts, each of which involves moving the contents of $2^{j+2} - 1$ cells. These executions thus take time $O(2^j)$.

Let $C(j)$ denote the time taken by an execution of PHASE(j). We have

$$C(0) = O(1)$$
$$C(j) = 2C(j-1) + O(2^j).$$

These equations imply

$$C(j) = O(j2^j).$$

Thus the time taken by the jth phase is $O(j2^j)$. It follows that the time taken by the first $j + 1$ phases is also $O(j2^j)$. The nth step of $\mathcal{M}$ is simulated during the first $j + 1$ phases, where $j = \lceil \log_2 n \rceil$, since $2^j \geq n$ steps of $\mathcal{M}$ are simulated in the jth phase alone. From this it follows that $\mathcal{M}'$ simulates $\mathcal{M}$ in time $O(n \log n)$.

If $\mathcal{M}$ has more than one tape, the tapes of $\mathcal{M}'$ can be divided into an equal number of tracks, with each track of $\mathcal{T}'$ having its own head position marker. Since $\mathcal{M}'$ is oblivious, all the tracks can be processed simultaneously. If $\mathcal{M}$ has more than one head per tape, it can be replaced without loss of time by a machine with a larger number of tapes, but only one head per tape (see Leong and Seiferas [5]).

This completes the proof of Theorem 3.

A simple application of the "overlap" argument introduced by Cook and Aanderaa [1] (see also Paterson, Fischer, and Meyer [7]) gives a transducer $\mathcal{M}$, with a single stack, having the following property: Any oblivious machine $\mathcal{M}'$ that simulates $\mathcal{M}$ on-line takes time at least proportional to n log n to do so. We shall describe $\mathcal{M}$ and sketch a proof of this property. A rigorous proof can be obtained by the methods of [1] and [7]

$\mathcal{M}$ will have a single stack, whose cells will contain either of two symbols: 0 or 1. Its input and output alphabets will contain three symbols (0, 1, and blank), and its control will have only a single state. At each step $\mathcal{M}$ reads an input.

(1) If the input is 0 or 1, this symbol is pushed onto the stack, and a blank output is written.

(2) If the input is blank and there is a 0 or 1 at the top of the stack, this symbol is popped off the stack and written as output.

(3) If the input is blank and the stack is empty, a blank output is written.

To see that any oblivious machine $\mathcal{M}'$ that simulates $\mathcal{M}$ on-line takes time at least proportional to n log n, consider the following ensemble of input sequences. Take $n = 2^j$. For each i ($0 \leq i \leq j - 1$) and each m ($0 \leq m \leq 2^{j-i} - 1$), let $I_{i,m}$ denote the interval of 2^i steps by $\mathcal{M}$, $m2^i$ through $(m + 1)2^i - 1$. Let $I'_{i,m}$ denote the interval of steps by $\mathcal{M}'$ that are devoted to simulating the steps of $I_{i,m}$. ($I'_{i,m}$ is well defined, since $\mathcal{M}'$ simulates $\mathcal{M}$ on-line.)

For each i and each even m in the range indicated above, consider the input sequences of length $n = 2^j$ that consist of 0's and 1's during $I_{i,m}$ and of blanks elsewhere. Since 2^i bits of information are read in during $I_{i,m}$ and written out again during $I_{i,m+1}$, the overlap between $I'_{i,m}$ and $I'_{i,m+1}$ must have size at least proportional to 2^i. The overlaps of all such pairs of intervals ($0 \leq i \leq j - 1$, $0 \leq m \leq 2^{j-i} - 1$, and m even) are disjoint and thus have sizes whose sum is at least proportional to $j2^{j-1}$, which is in turn proportional to n log n. Since $\mathcal{M}'$ is oblivious, the time it takes to simulate these n steps by $\mathcal{M}$ must be at least proportional to the overlap, which is in turn proportional to n log n. This shows that Theorem 3 gives the best possible result for on-line simulation.

COMBINATIONAL LOGIC NETWORKS. The networks we shall consider are acyclic interconnections of *gates* by means of *wires* that carry binary signals, as described by Muller [6]. It will be assumed that there are finitely many different types of gates available, and that these form a "universal" basis, so that any input-to-output function can be implemented by a suitable network. Each type of gate has a *cost* and a *delay*, which are positive

real numbers. The *cost* of a network is the sum of the costs of its gates. The depth of a network is the maximum overall input-to-output paths of the sum of the delays of the gates on that path.

The method used above for programming an oblivious machine can also be used to construct a combinational logic network that implements the first n steps in the computation of a transducer $\mathcal{M}$ with one-dimensional tapes. Such a network will have n inputs carrying suitable encodings of the symbols read from the input tape and n outputs carrying encodings of the symbols written onto the output tape. (If the input and output alphabets have more than two symbols, the inputs and outputs of the network will be "cables" of wires carrying binary signals.)

THEOREM 4. *If $\mathcal{M}$ is a transducer with one-dimensional tapes, there is a combinational logic network implementing n steps by $\mathcal{M}$ with cost $O(n \log n)$ and depth $O(n)$.*

PROOF. Only those cells of $\mathcal{T}'$ in a ball of radius 2 centered at the origin are relevant to the execution of the STEP subroutine, and only those in a ball of radius $2^{j+1} - 1$ are relevant to the executions of the COMPRESS(j) and EXPAND(j) subroutines. The computation of the STEP subroutine can thus be implemented by a "module" that has, in addition to one input and one output of the type mentioned above, five inputs and five outputs that carry suitable encodings of the contents of the relevant cells of $\mathcal{T}'$, and an input and an output that carry suitable encodings of the state of $\mathcal{M}'$. Similarly, the computations of the COMPRESS(j) and EXPAND(j) subroutines can be implemented by modules that have $2^{j+2} - 1$ inputs and $2^{j+2} - 1$ outputs that carry encodings of contents of cells of $\mathcal{T}'$. In addition, the COMPRESS(j) module will have a single output wire, and the EXPAND(j) module will have a single input wire, carrying the single bit of information that the COMPRESS(j) subroutine places on the stack and the EXPAND(j) subroutine removes from the stack.

The modules described above can be combined (as shown in Figure 1) to form a network

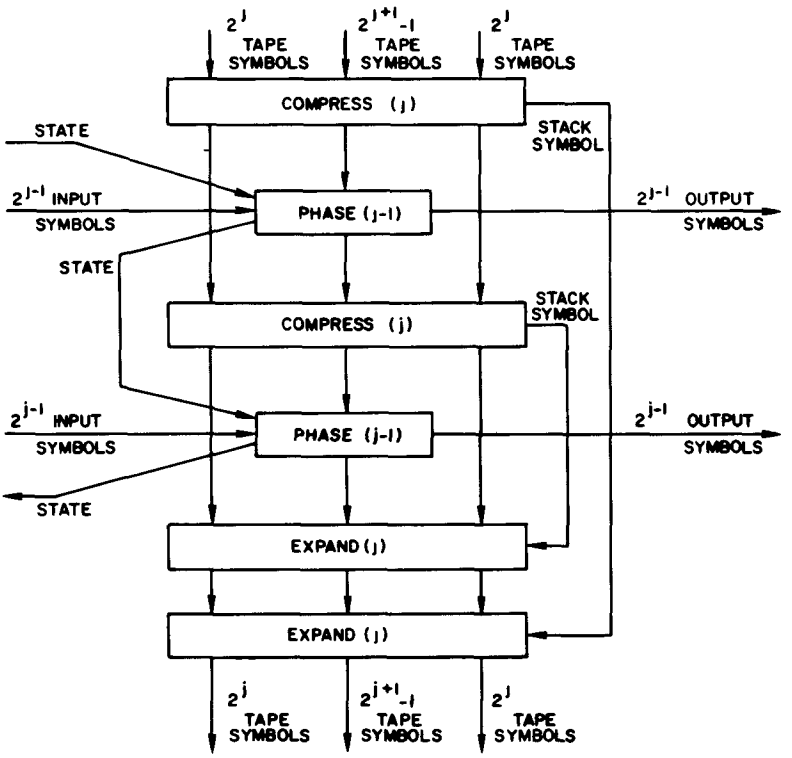

FIG. 1 Recursive construction for PHASE(j) network

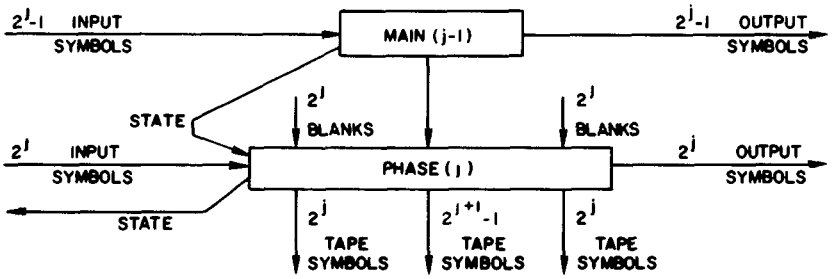

FIG 2 Recursive construction for MAIN(j) network

PHASE(j) that implements the computation of the corresponding subroutine, and then (as shown in Figure 2) to form a network MAIN(j) that implements the computation of the first $j + 1$ phases of the MAIN program, and therefore of the first $2^{j+1} - 1$ steps by $\mathcal{M}$.

The STEP module has cost $O(1)$ and depth $O(1)$. When suitably designed, the COMPRESS(j) module has cost $O(2^j)$ and depth $O(j)$; this depth is needed to determine whether the head position marker lies to the left or the right of the origin marker. Similarly, the EXPAND(j) module has cost $O(2^j)$ and depth $O(1)$. Thus the cost $L(j)$ and depth $D(j)$ of the PHASE(j) network satisfy

$$L(0) = O(1)$$
$$D(0) = O(1)$$
$$L(j) = 2L(j-1) + O(2^j)$$
$$D(j) = 2D(j-1) + O(j).$$

These equations imply

$$L(j) = O(j2^j)$$
$$D(j) = O(2^j).$$

The cost $L'(j)$ and depth $D'(j)$ of the MAIN(j) network satisfy

$$L'(0) = L(0) + \cdots + L(j) = O(j2^j)$$
$$D'(0) = D(0) + \cdots + D(j) = O(2^j).$$

Since this network implements $2^{j+1} - 1$ steps by $\mathcal{M}$, it follows that there is a network that implements n steps by $\mathcal{M}$ with cost $O(n \log n)$ and depth $O(n)$.

This completes the proof of Theorem 4.

ACKNOWLEDGMENTS. The authors are indebted to Albert R. Meyer and Joel I. Seiferas for many helpful comments on this work.

REFERENCES

1. COOK, S A, AND AANDERAA, S O On the minimum computation time of functions *Trans AMS 142* (Aug 1969), 291–314
2. FISCHER, P.C, MEYER, A R, AND ROSENBERG, A L Real-time simulation of multihead tape units *J ACM 19*, 4 (Oct 1972), 590–607
3. HENNIE, F C On-line Turing machine computations *IEEE Trans EC 15*, 1 (Feb 1966), 34–44
4. HENNIE, F C, AND STEARNS, R E Two-tape simulation of multitape Turing machines *J ACM 13*, 4 (Oct 1966), 533–546
5. LEONG, B, AND SEIFERAS, J New real-time simulations of multihead tape units Proc 9th Annual ACM Symp Theory of Comptng, Boulder, Colo, May 1977, pp 239–248
6. MULLER, D E Complexity in electronic switching circuits *IRE Trans EC 5*, 1 (March 1956), 15–19
7. PATERSON, M S, FISCHER, M J, AND MEYER, A R An improved overlap argument for on-line multiplication *SIAM-AMS Proc 7* (1974), 97–111
8. SAVAGE, J E Computational work and time on finite machines *J ACM 19*, 4 (Oct 1972), 660–674

9 SCHNORR, C P The network complexity and the Turing machine complexity of finite functions *Acta Informatica 7* (1976), 95–107
10. STOSS, H -J Zwei-band Simulation von Turingmaschinen *Computing 7* (1971), 222–235
11 TURING, A M On computable numbers, with an application to the *Entscheidungsproblem Proc London Math Soc (2) 42* (1936), 230–265, corrections *43* (1937), 544–546

RECEIVED JUNE 1977, REVISED SEPTEMBER 1978