UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Iterative Tree Arrays with Logarithmic Depth

Karel Culik II
Oscar H. Ibarra
Sheng Yu

CS-85-03

March, 1985

ITERATIVE TREE ARRAYS

WITH

LOGARITHMIC DEPTH*

Karel Culik II†, *Oscar H. Ibarra*†† *and Sheng Yu*†

ABSTRACT

An iterative tree array (ITA) is a binary tree-connected systolic network in which each cell is a finite-state machine and the input is applied serially at the root. We present an algorithm for simulating a pushdown stack of size $S(n)$ on an ITA of depth $\log S(n)$ in real-time. Some interesting applications are the following:

(1) Every linear iterative array operating in (simultaneous) time $T(n)$ and space $S(n)$ can be simulated by an ITA in time $T(n)$ and depth $\log S(n)$.

(2) $S(n)$-space bounded on-line TM's are equivalent to $\log S(n)$-depth bounded ITA's.

(3) $\log n$ depth is a necessary and sufficient condition for an ITA to recognize every context-free language.

(4) $\log\log n$ depth is a necessary condition for an ITA to recognize a nonregular set.

(5) Every on-line nondeterministic TM with $\log n$−bounded nondeterminism operating in linear time and space can be simulated by an ITA with $O(\log n)$ depth in linear time.

KEYWORDS

iterative arrays, iterative tree arrays, bounded nondeterminism, systolic systems, parallel computing.

---

* This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant A-7403, and the National Science Foundation MCS83-04756. Research of O.H.Ibarra was also supported by a John Simon Guggenheim Memorial Foundation Fellowship.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

†† Department of Computer Science, University of Minnesota Minneapolis, MN 55455, USA

1. INTRODUCTION

Linear iterative arrays ($LIA's$) have been studied extensively (see [2, 3, 5]). An LIA is a linear bidirectional array of finite-state machines with serial input/output. Recently, an interesting generalization of the LIA, called iterative tree array (ITA), was introduced in [4]. An ITA is a tree-connected synchronous network of finite-state machines with bidirectional data flow whose serial input/output is at the root of the tree. An example of such a device, but one in which the cells are not finite-state, is the dictionary (data base) machine of [1] (see also [11]). The main result in [4] is that every nondeterministic $T(n)$-time bounded on-line TM can be simulated by a $T(n)$-time bounded deterministic ITA. An ITA (even a real-time) may use exponentially many processors and, therefore, is of little practical interest in the general form. In this paper, we study $D(n)$-depth bounded $ITA's$. We show a number of results on the power of such devices as related to Turing machines ($TM's$) and $LIA's$ with various time, space and nondeterminism bounds. Our emphasis is in the case when $D(n) = \log n$.[1]

As our main auxiliary result we present an algorithm for simulating a push-down stack of size $S(n)$ on an ITA of depth $\log S(n)$. Moreover, the simulation is done in real-time (i.e. without time loss). Clearly, since the size of the stack is not known in advance, the algorithm must generate the nodes that are used in the simulation gradually, level by level, as the stack size grows. What complicates matters is the restriction that each cell of the ITA, being a finite-state machine, can only store a constant amount of information. Recently, in [1], a

---

[1] All logarithms in this paper have base 2.

systolic tree implementation of a dictionary machine was given, where an algorithm for storing n elements in a tree of logarithmic depth was presented. The algorithm, which supports all the dictionary and priority queue operations as well as some other data queries, requires each cell of the tree to store an unbounded amount of information and to execute a program that is not finite-state computable. Here , we show that for pushdown stack simulation, the cells of the tree can be made finite-state.

The stack simulation result has a number of interesting applications some of which are listed as follows.

(1) Every linear iterative array operating in (simultaneous) time $T(n)$ and space $S(n)$ can be simulated by an ITA in time $T(n)$ and depth $\log S(n)$.

(2) $S(n)$-space bounded on-line TM's are equivalent to $\log S(n)$-depth bounded ITA's.

(3) $\log n$ depth is a necessary and sufficient condition for an ITA to recognize every context-free language.

(4) $\log\log n$ depth is a necessary condition for an ITA to recognize a nonregular set.

(5) Every on-line nonderministic TM with $\log n$-bounded nondeterminism operating in linear time and space can be simulated by an ITA with $O(\log n)$ depth in linear time.

(6) If the nodes of the ITA are TM's rather than finite-state machines, its power is not increased.

The paper consists of four sections in addition to this section. Section 2

recalls the definitions of linear iterative arrays and iterative tree arrays. Section 3 presents the stack simulation algorithm. Section 4 modifies the algorithm so that it can be used to simulate a linear iterative array of size $S(n)$ on an iterative tree array of depth $\log S(n)$ in real-time. Finally, Section 5 discusses some applications.

2. PRELIMINARIES

A *linear iterative array or automaton* (*LIA*) is a one-dimensional one-way infinite sequence of finite-state machines called cells, see Figure 1. Each of the cells, except for the leftmost cell, communicates with its left and right neighbors. The leftmost cell (the special cell) communicates with the external world and its right neighbor. The device works synchronously. At the beginning (time 0), all the cells are in a quiescent state. The next state of a cell is a function of its current state, the current state of its left neighbor (or possibly an external input for the special cell) and the current state of the right neighbor. The input to the *LIA* is read *on-line* as follows. The states of the special cell are partitioned into two classes: reading states and nonreading states, with the initial state being a reading state. An input string is of the form $a_1a_2 \cdots a_n\$$, $n \geq 0$, each a_i in Σ. The special cell can only read an input symbol a_i (or \$) when it is in a reading state. Thus a_1 is read at time 0. After reading \$, the special cell can no longer enter a reading state. The input $a_1 \cdots a_n$ is accepted if the *LIA* when given $a_1 \cdots a_n\$$ eventually enters an accepting state after reading \$. (Note that an accepting state must be a nonreading state.) The language accepted by the *LIA* is the set of all accepted strings. The *LIA* has time complexity $T(n)$ (or is $T(n)$-

time bounded) if any string of length n that is accepted requires no more than $T(n)$ time to accept. Clearly, $T(n) \geq n+1$. If $T(n) = n+1$, the *LIA* is said to operate in real-time. Similarly, we say that the *LIA* is $S(n)$-space bounded if any string of length n that is accepted uses no more than $S(n)$ cells.

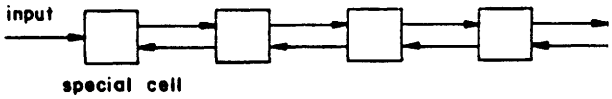


Figure 1. An *LIA*

LIA's in general and real-time *LIA*'s in particular have been studied quite extensively, see, e.g., [2,3,5].

An interesting generalization of the *LIA*, called *iterative tree array or automaton* (*ITA*), was introduced and studied in [4]. The cells of the *ITA* are connected as an infinite full binary tree (see Figure 2). The root of the tree (the special cell), like the leftmost cell in an *LIA*, communicates with the external world. As in an *LIA*, the input $a_1 \cdots a_n\$$ is applied at the root on-line. The *ITA* operates like an *LIA* except now each nonroot cell has three neighbors. The notions of acceptance, time complexity, and space complexity are similar to those for an *LIA*. In addition, we have another complexity measure — depth complexity. An *ITA* is $D(n)$-depth bounded or operates in depth $D(n)$ if any string of length n that is accepted uses no more than $D(n)$ levels of the tree, the root being at level 0.

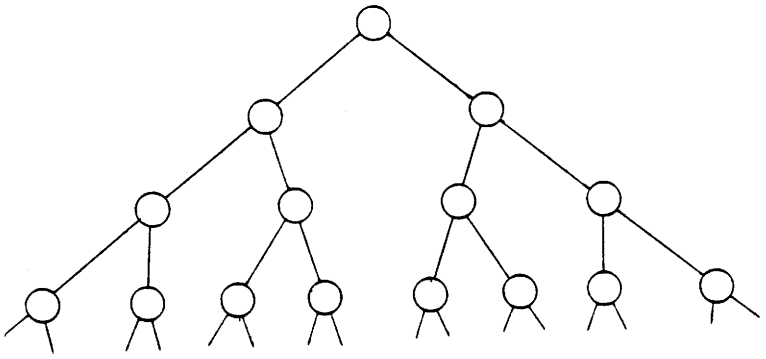


Figure 2. An *ITA*

3. IMPLEMENTATION OF A STACK ON AN ITA

In [4], an algorithm for simulating a (pushdown) stack on an *LIA* was briefly described. The algorithm uses three "registers" in each cell. Hence, up to three stack elements can be stored in a cell. The leftmost cell is the top of stack. The following are the two basic operations of each cell of the iterative array:

(1) If there are three elements in the cell, the rightmost element will be sent to the right neighbor.

(2) If there is only one element in the cell, it will get one element from the right neighbor.

An example of a sequence of stack pushings and poppings is shown in Figure 3.

In [4], the simulation of a stack by an *ITA* is implemented in such a way that a path of the tree is used to function as an *LIA*. Hence, for a stack of size $S(n)$, the tree will have depth $S(n)$. Thus exponential storage is used. We now show a technique, called "snaking technique", in which only $\log S(n)$ depth of the

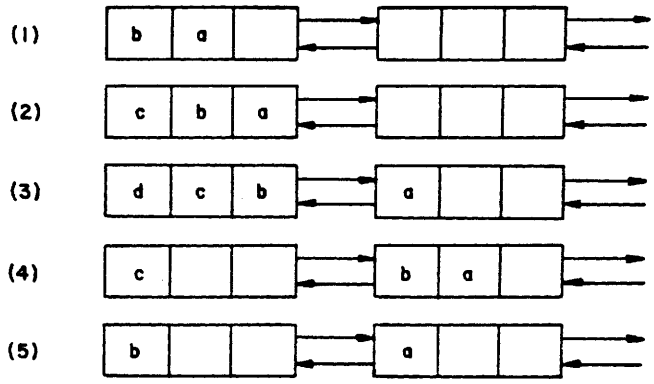


Figure 3.

tree is used to simulate a stack of size $S(n)$. Moreover, the simulation is done in real-time (i.e. without time loss).

The basic idea is to snake the stack on the tree along the in-order traversal route. Figure 4 shows the outline of the sequence of the stack cells. The detailed picture is given in Figure 5.

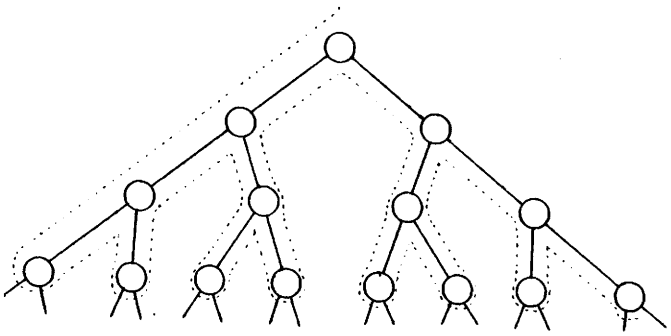


Figure 4.

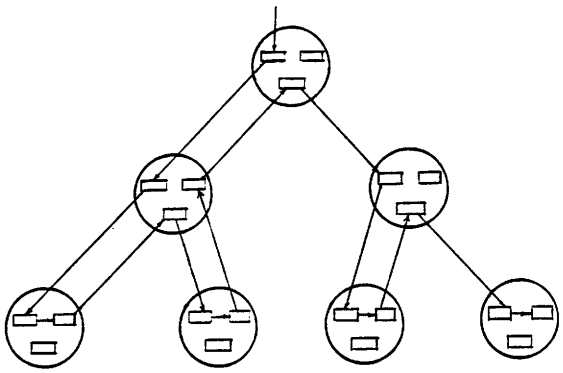


Figure 5.

In this scheme, each node of the tree simulates three cells of the *LIA* implementation of the stack. We call them Cell 1, Cell 2 and Cell 3, respectively, as labeled in Figure 6.

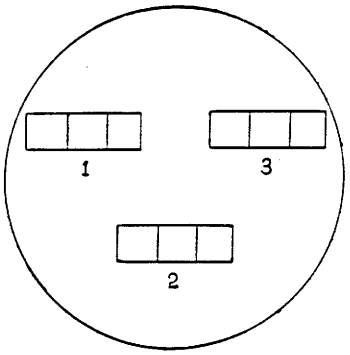


Figure 6.

Since the size of the stack is not known until the end of the input, the depth of the tree to be used cannot be decided in advance. Therefore, the depth cannot be of a static size during the computation. It must be dynamic and expanded gradually level by level as the size of the stack grows. The following points should be noted when considering the correctness of the algorithm.

(A) The stack tree (the part of the tree which is being used for the

implementation of the stack) should be balanced so as to guarantee logarithmic depth.

(B) The stack should never be broken at any time.

(C) There should always be enough space in the stack.

We now describe the algorithm.

(1) Initially, there is only one node, the root, in the stack tree and this node is marked 'bottom'. Cells 1 and 3 of this node are marked 'EP' (Expansion Point) and 'EE' (Expansion End), respectively. See Figure 7.

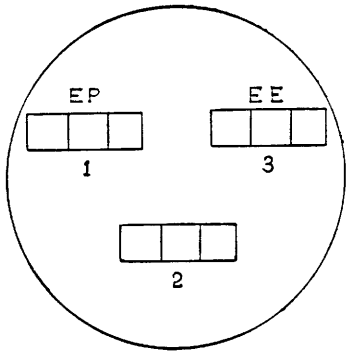
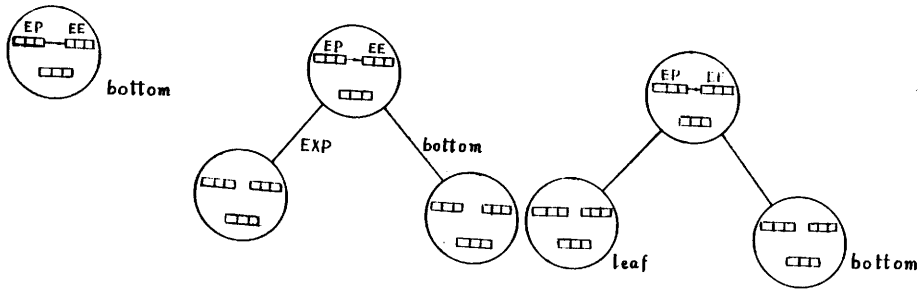


Figure 7.

(2) Whenever a symbol is pushed into Cell 1 of the node with the 'bottom' mark, it gives the 'bottom' mark to its right son and sends an expansion signal 'EXP' along the path up to the root. Every node on this path, in turn, sends the expansion signal 'EXP' down the left subtree after receiving th signal. To be precise, there are three cases:

Case 1. When the root or an internal node receives the signal 'EXP', it just sends 'EXP' signals to its left and right sons.

Case 2. When a node on the new level (of the tree) receives the signal 'EXP', the node is marked 'leaf', and 'EXP' is no longer sent down. Figure 8 shows three successive configurations of the tree after an element is pushed into Cell 1 of Figure 8(a).



(a) (b) (c)

Figure 8.

Case 3. When a node marked 'leaf' receives the signal 'EXP', Cells 1 and 3 of this node are marked 'EP' and 'EE', respectively, the 'leaf' mark is removed, and 'EXP' signals are sent to its left and right sons. Figure 9 shows the configurations that the tree assumes when an element is pushed into Cell 1 of the bottom node (Figure 9(a)).

(3) The rules for linking the cells (local expansion) are as follows. With one exception, when an 'EP' cell holds three symbols, a new cell will be inserted into the stack, and the 'EP' mark will be given to it. The choice for the new cell depends on the case.

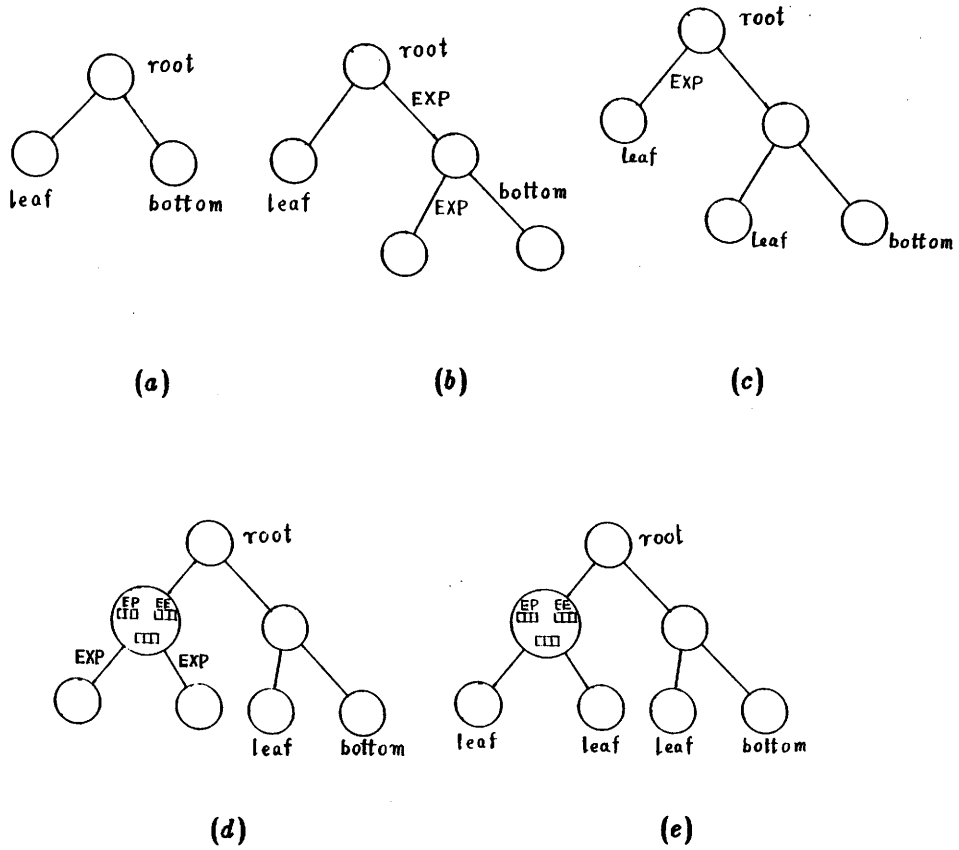


Figure 9.

Case 1. If the 'EP' cell is Cell 1 of a non-leaf node, then the new cell is Cell 1 of its left son.

Case 2. If the 'EP' cell is Cell 1 of a leaf-node, then the new cell is Cell 3 of the same node.

Case 3. If the 'EP' cell is Cell 3 of a leaf-node which is a left son, then the new cell is Cell 2 of its parent node.

Case 4. If the 'EP' cell is Cell 2, then the new cell is Cell 1 of its right son.

The case when the 'EP' cell is Cell 3 of a leaf-node which is a right son is an

exception. In this case, no new cell is inserted since it must be the case that the 'EP' mark has met 'EE'. When this happens, we say that the local expansion is finished.

In order to show that the algorithm is correct, it is essential to prove that the next level of expansion will never occur until the current expansion has been finished. That is, a stack symbol is pushed into the new bottom node only if every cell marked 'EE' has been passed through by some stack symbol.

Now let us consider the 'EE' markers which are in the rightmost node of the maximum sub-stack-tree whose roots are directly connected to the rightmost path of the tree. These nodes are shown in Figure 10 for a stack tree of four levels. The proof for other 'EE' markers is easily derived from the proof of these cases. We name the selected nodes as $C_1, C_2, \ldots, C_n$, where n is the depth of the stack tree. For the 'EE' cell in cell C_i, there are $5*2^{i-1}-4$ cells from the 'EE' cell to the bottom node (exclude the cells in the bottom node). Each cell has at most three symbols. So, there are at most $3*(5*2^{i-1}-4) = 15*2^{i-1}-12$ symbols stored starting from this cell. It takes $2i$ steps for the 'expansion' signal to reach this node. Thus, there are at most $2i$ symbols pushed through during the signal propagation. Therefore at most $15*2^{i-1}+2i-12$ symbols are possibly stored. But $2^{i+1}-2$ (except the rightmost one) new cells can be expanded beyond this 'EE' cell. There are alltogether $9*2^{i-1}-6$ old and new cells and hence more than $18*2^{i-1}-12$ symbols must be stored at some time step in order to reach the next cell. Since $15*2^{i-1}+2i-12 \leq 18*2^{i-1}-12$ for any $i \geq 1$, the bottom cell can be reached only if some symbol(s) pushed through the 'EE' fence, i.e., the expansion at that local point has been finished. The special case when

the root is initially the only node in the stack tree is clearly also correct.

The balance and non-overflow properties of the stack tree can be easily verified by the above result. It is also easy to see that the stack is never broken directly from the expansion technique.

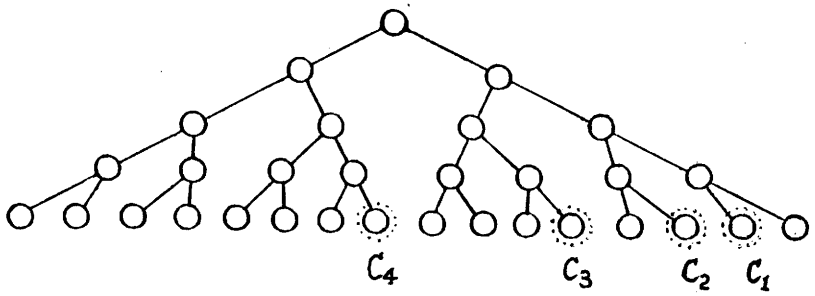


Figure 10.

4. SIMULATION OF AN LIA ON AN ITA

We know that an *LIA* can be simulated by an *ITA*. The *ITA* can simply use one path, say the leftmost path. However, this takes $O(n)$ depth (of the *ITA*) and, hence, exponential space (i.e. number of cells). We can reduce the number of cells used by the *ITA* by using a modified snaking technique.

An *LIA* is different from a stack in that it grows regularly at the right end instead of pushing and popping at the top. Moreover, every element of the *LIA* enters a new state at each time step rather than being simply stored as a memory symbol. In order to use the snaking algorithm, the following modifications are

necessary:

(a) If a cell has one or two simulated *LIA* elements in it, it should get one element from the next cell, provided there is one.

(b) In the local expansion, the new expanded cells get elements from the 'EE' marked cell rather than the 'EP' marked cell.

We leave the details to the reader.

From the above discussion, we have

Theorem 1. *Every LIA operating in (simultaneous) time $T(n)$ and space $S(n)$ can be simulated by an ITA in time $T(n)$ and depth $\log S(n)$.*

The next result shows that *ITA*'s are considerably more powerful than *LIA*'s.

Corollary 1. *The class of languages accepted by LIA's in real-time is properly contained in the class of languages accepted by ITA's in real-time and depth $\log n$.*

Proof: Let

$$L = \{x_1\#x_2\# \cdots \#x_{2^t}\$y\$^{2t+1} \mid |x_1| = |x_2| = \cdots = |x_{2^t}| = |y| = t,$$
$$x_1, x_2, \ldots, x_{2^t}, y \in \{0,1\}^+, y = x_i \text{ for } some\ 1 \leq i \leq 2^t\}$$

Using the same approach as in [4], it is easy to prove that L is accepted by an *ITA* in real-time and depth $\log n$. The fact that L is not accepted by any real-time *LIA* can be easily verified by using Cole's Theorem [3]. □

5. APPLICATIONS

In this section, we investigate the relationship between time- and depth-bounded *ITA*'s and time- and space-bounded *on-line* multitape Turing machines (or simply on-line *TM*'s). See [8] for the definitions of on-line and off-line *TM*'s. We also look at recognition of context-free languages by *ITA*'s. We begin with the following theorem.

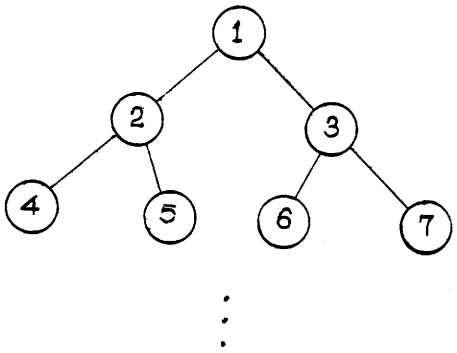
Theorem 2. *An on-line TM operating in time* $T(n)$ *and space* $S(n)$ *can be simulated by an ITA in time* $T(n)$ *and depth* $\log S(n)$.

Proof: Clearly, each $S(n)$-space bounded worktape of the on-line *TM* can be replaced by two $S(n)$-space bounded pushdown stacks. The theorem now follows from the stack-implementation algorithm (described in Section 2) by using several channels at each node of the *ITA*, each channel corresponding to a pushdown stack. □

Conversely, we have

Theorem 3. *An ITA operating in time* $T(n)$ *and depth* $D(n)$ *can be simulated by an on-line TM in time* $T(n)2^{D(n)}$ *and space* $2^{D(n)}$.

Proof: The cells of the *ITA* are stored sequentially on a worktape of the on-line *TM* according to the following labeling:



It follows that the on-line TM needs at most $2^{D(n)}$ steps to update the states of the cells of the ITA for each move of the ITA. □

From Theorems 2 and 3, we get the following corollary.

Corollary 2. *$S(n)$-space bounded on-line TM's are equivalent to $\log S(n)$-depth bounded ITA's.*

Corollary 3. *$\log n$-depth bounded ITA's are equivalent to deterministic linear-bounded automata.*

Corollary 4. *$O(\log n)$-depth bounded ITA's are equivalent to polynomial space-bounded TM's.*

The next corollary shows that an additive constant in the depth complexity does not change the computing power of the ITA. However, a multiplicative constant increases the power.

Corollary 5.

(1) *For any rational constant $c \geq 1$, $(\log n + c)$-depth bounded ITA's can be converted into $\log n$-depth bounded ITA's.*

(2) *For any rational constants* $1 \leq c < d$, $d \log n$*-depth bounded ITA's are strictly more powerful (i.e. accept more languages) than* $c \log n$*-depth bounded ITA's.*

Proof: This follows from Corollary 2 and the tape-reduction and tape-hierarchy theorems for *TM*'s [8]. □

Next, we show that an *ITA* needs $\log\log n$ depth to accept a nonregular language.

Theorem 4.

(1) *There are nonregular languages accepted by ITA's in* $\log\log n$ *depth.*

(2) *If* L *is accepted by an ITA in* $D(n)$ *depth and* L *is nonregular, then* $\inf_{n \to \infty} \frac{D(n)}{\log\log n} > 0$.

Proof:

(1) The nonregular language $L = \{0^n 1^n \mid n \geq 1\}$ can be accepted by an on-line *TM* in $\log n$ space and, hence, by Corollary 2 can be accepted by an *ITA* in $\log\log n$ depth.

(2) This follows from Corollary 2 and the fact that if L is a nonregular language accepted by an on-line *TM* in space $S(n)$, then $\inf_{n \to \infty} \frac{S(n)}{\log n} > 0$ [8]. □

Corollary 6. *Every (one-way) deterministic counter language can be accepted by an ITA in linear time and* $\log\log n$ *depth.*

Proof: It is obvious that every deterministic counter language can be accepted by an on-line *TM* in linear time and $\log n$ space. The result follows from Theorem 2. □

The next result concerns context-free languages (*CFL*'s).

Theorem 5. $\log n$ *depth is a necessary and sufficient condition for an ITA to recognize every CFL.*

Proof: Every *CFL* can be accepted by an off-line *TM* in space $\log^2 n$ and, therefore, by an on-line *TM* in space n. Hence, by Corollary 2, every *CFL* can be accepted by an *ITA* in depth $\log n$. The necessity follows from Corollary 2 and the observation that the *CFL* $L = \{x\#x^R \mid x \text{ in } \{0,1\}^+\}$ requires $\inf_{n \to \infty} \frac{S(n)}{n} > 0$ for recognition by an on-line $S(n)$-space bounded *TM*. □

It can be shown that every *CFL* can be accepted by an *LIA* in quadratic time and space (see, e.g., [9,10]). Hence, from Theorem 1 we have

Corollary 7. *Every CFL can be accepted by an ITA in quadratic time and* $2\log n$ *depth.*

For the case of deterministic *CFL*'s, the simulation is more efficient.

Corollary 8. *Every deterministic CFL can be accepted by an ITA in linear time and* $\log n$ *depth. Moreover, the* $\log n$ *depth is necessary.*

Proof: This follows from Theorem 2 and the observation that a deterministic pushdown automaton can be simulated by an on-line *TM* in linear time and n

space. That $\log n$ is necessary follows from the proof of Theorem 5 since L is a deterministic *CFL*. □

We can define nondeterministic *LIA*'s and *ITA*'s in the obvious way (i.e., each cell is a nondeterministic finite-state machine), and our results carry over to the nondeterministic case. For example, Corollary 3 becomes

Corollary 9. *$\log n$-depth bounded nondeterministic ITA's accept exactly the context-sensitive languages.*

Since every *CFL* can be accepted by a real-time pushdown automaton [7], we have

Corollary 10. *Every CFL can be accepted by a real-time nondeterministic ITA in depth* $\log n$.

Finally, we show that an *ITA* can efficiently simulate an on-line nondeterministic *TM* (on-line *NTM*). In [4] it was shown that every $T(n)$-time bounded on-line *NTM* can be simulated by a deterministic *ITA* in $O(T(n))$ time. Here we determine the depth that the tree needs for the simulation in relation to the bounds on space and nondeterminism of the simulated *NTM*. We use the notion of $N(n)$-bounded nondeterminism from [6].

Theorem 6. *Let M be an on-line NTM with $N(n)$-bounded nondeterminism operating in time $T(n)$ and space $S(n)$. If $N(n)$ is computable by an ITA with $D(n)$ depth in time $O(T(n))$, then M can be simulated by an ITA with $O(\max(N(n)+\log S(n), \log n, D(n)))$ depth operating in $O(T(n))$ time.*

Proof: We may assume that the on-line *NTM* M has pushdown stacks for its worktapes and that it has at most two choices for each move. The idea of the proof follows that of Theorem 5.2 in [4]. M is simulated by the *ITA* as follows.

(1) First, the entire input string is read and stored (in the snaking way, if necessary) and the function $N(n)$ is computed. The computation of $N(n)$ is such that at the end of the process, all nodes in level $N(n)$ of the *ITA* are tagged.

(2) At the beginning of the simulation, the root of the *ITA* functions as the finite control of M, and every path starting from the root and ending at a tagged node simulates the stacks of M.

(3) To simulate a move with two choices, the node which is currently simulating the finite control pushes all its stacks one level down, and passes the current state to its two sons. Its two sons then simulate two different choices, respectively.

(4) When a path is not long enough for simulating the stacks, it extends from the tagged node, and the extension follows the rules of the *snaking technique.*

(5) The computed results from all branches ar *OR*ed and sent back to the root.

Since $N(n)$ levels are reserved for branching in order to simulate $N(n)$ nondeterministic moves and $\log S(n)$ levels are necessary for the simulation of the stacks, we need $O(N(n)+\log(S(n))$ depth of the *ITA* to implement the simulation, provided $\log(n)$ and $D(n)$ are "small", where $\log(n)$ is the depth

needed for storing the input and $D(n)$ for calculating $N(n)$. □

Corollary 11. *Any on-line NTM with* $\log n$*-bounded nondeterminism operating in linear time and space can be simulated by a deterministic ITA with* $O(\log n)$ *depth in linear time.*

Every node (cell) of an *ITA*, as defined above, is a finite-state machine. We now consider a generalization when every node is a *TM*, performing one computational step at every time step of the whole *ITA*. We call this generalized device an iterative tree machine (*ITM*). The major difference between these two systems is that the storage in each node is finite in the former but infinite in the latter. We will show that, surprisingly, these two kinds of systems have the same power if time is the only concern. Then we will show how much more depth is needed for an *ITA* to simulate an *ITM*.

Theorem 7. *Any ITM can be simulated by an ITA in real time (i.e., without time loss).*

Proof: Let us label the nodes of a tree by the following rules:

(i) The root is labeled "1".

(ii) If a node is labeled "x", then its left and right sons are labeled "$x0$" and "$x1$", respectively.

Each node of the *ITA* simulates the corresponding node of the *ITM* with the same label. The node uses a path starting from it to simulate the stacks of the *TM* as described in the proof of Theorem 6. The node labeled "z" uses the

path $z1^*$ *i.e.* $z, z1, z11, \ldots$ if $z = z'0$, or the path $z0^*$ *i.e.* $z, z0, z00, \ldots$ if $z = z'1$. It is clear that every node of the *ITA* has only one such path passing through it and, therefore, each node can be implemented as a finite-state machine. It is also clear that the simulation can be done in real time. □

Obviously an *ITA* can be simulated by an *ITM* in real time. So, we have the following corollary.

Corollary 12. *A language L is accepted by an ITA in $T(n)$ time if and only if it is accepted by an ITM in $T(n)$ time.*

Although no time is lost in the simulation of an *ITM* by an *ITA*, more processors might be needed. If only $D(n)$ depth of the *ITM* is used in the computation and $D(n)$ is computable by an *ITA*, then we can optimize the space used in the simulation by first marking the $D(n)$ depth and then forcing all the stacks to follow the rules of the snaking technique when they reach the marked level.

Corollary 13. *A $D(n)$-depth bounded ITM with each node being an $S(n)$-space bounded TM can be simulated by an ITA with $O(D(n)+\log S(n))$ depth.*

If the marking of $D(n)$ depth takes the same order of magnitude of time as the computation of the simulated *ITM*, then the total simulation time will still be in the same order of magnitude.

Corollary 14. *If L is accepted by an ITM with $\log(n)$ depth in linear time, then L is also accepted by an ITA with $O(\log(n))$ depth in linear time.*

Proof: This follows from the fact that the marking of the $\log n$ depth takes

linear time, and each node of the *ITM* cannot use more than linear space. □

REFERENCES

[1] Atallah, M., and Kosaraju, S., A Generalized Dictionary Machine for VLSI, IEEE Trans. Comp. 34(2), (1985), 151-155.

[2] Choffrut, C., and Culik II, K., On Real-Time Cellular Automata and Trellis Automata, *Acta Informatica*, 21(1984), 393-407.

[3] Cole, S.N., Real-Time Computation by n-Dimensional Iterative Arrays of Finite-State Machines, *IEEE Trans. Comp. 18*, (1969), 349-365.

[4] Culik II, K., and Yu, S., Iterative Tree Automata, *Theoretical Computer Science 32*, (1984), 227-247.

[5] Fischer, P.C., Generation of Primes by a One-Dimensional Real-Time Iterative Array, *JACM 12*, (1965), 388-394.

[6] Fischer P.C. and Kintala C.M.R., Computations With a Restricted Number of Nondeterministic Steps, *Ninth ACM Symposium on Theory of Computing*, May 1977.

[7] Harrison, M., *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

[8] Hopcroft, J., and Ullman, J., *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969.

[9] Ibarra, O.H., Palis,M.A. and Kim,S.M., Designing Systolic Algorithms Using Sequential Machines, *University of Minnesota, Department of*

Computer Science Technical Report, 1984, submitted for publication.

[10] Kosaraju,S., Speed of Recognition of Context Free Languages by Array Automata, *SIAM J. of Comp.* 4(1975) 331-340.

[11] Ottmann, T.A., Rosenberg, A.L., and Stockmeyer, L.J., A Dictionary Machine (for VLSI), *IEEE Trans. Comp. 31(9)*, (1982), 892-897.