Characterizing Unambiguous Augmented Pushdown Automata by Circuits

(Extended Abstract)

Klaus-Jörn Lange Peter Rossmanith

Institut für Infomatik
Technische Universität München
Arcisstr. 21
D-8000 München 2

Abstract

The notions of *weak* and *strong unambiguity* of augmented push-down automata are considered and related to unambiguities of circuits. In particular we exhibit circuit classes exactly characterizing polynomially time bounded unambiguous augmented push-down automata.

Introduction

An important object in parallel complexity theory, the class NC, can be characterized in terms of Parallel Random Access Machines, see e.g. [3, 4, 9], Boolean Circuits [3, 10], Augmented Pushdown Automata [6, 7], and Alternating Turing Machines [1, 6]. There are close connections between these concepts, see e.g. [3, 5].

In [5] unambiguous circuits were considered in order to characterize CREW-PRAMs and to further relate them with the NC-structure. Working with unambiguity we must distinguish between the notions of unambiguity and uniqueness. While uniqueness *uses* the unique existence of a computation path as a tool for acceptance, unambiguity *requires* this unique existence for all accepting computation pathes. We call this kind of unambiguity *weak unambiguity* and additionally consider *strong unambiguity*. An automaton M is said to be strongly unambiguous if there is at most one computation path between two arbitrary configurations of M.

Application of unambiguity to circuits led to the idea of vulnerable gates: Just as in a CREW-PRAM any multiple access to a memory cell is forbidden, we assume an unambiguous circuit to avoid any multiple 1-input to OR-gates of unbounded fan-in (resp. multiple 0-input to AND-gates of unbounded fan-in). This idea in mind, we will

use unambiguous versions of AC^k and SAC^k and additionally $UnambRAC^k$ as a subclass of $UnambRAC^k$, in which even the OR-gates of bounded fan-in are vulnerable.

In [5] $CREW\text{-}TIME(\log n) = UnambRAC^1$ was shown in analogy $CRCW\text{-}TIME(\log n) = AC^1$ of [9]. This result works with strongly unambiguous circuits, since CREW-algorithms do not allow any multiple write-accesses regardless of the influence of the written memory cell to the final result. In addition we will consider in the following the misuse of vulnerable gates as long as their results have no influence on the final results, that is as long as the output evaluates to either 0 or 1 for all inputs. Here our aim is to extend te equation $SAC^1 = NAPDA_{\mathrm{PT}}(\log n)$ of [10]. In fact we will show this for weak and for strong unambiguity!

Preliminaries

We are talking about circuit families of polynomial size and polylogarithmic depth which are composed of bounded AND- and OR-gates and possibly unbounded $\exists$- and $\forall$-gates. We will use without explanation circuit classes like AC^k, SAC^k, and NC^k, see [3, 7, 9]. A circuit C of n inputs is said to be (strongly) unambiguous if for all 2^n assignments of its inputs no $\exists$-gate receives a 1 by two or more of its predecessors and no $\forall$-gate receives a 0 by two or more predecessors. That is, C might be build up by vulnerable gates and still for every input the outcome of every gate is well-defined. This corresponds to a 'strong' notion of unambiguity.

Applying this notion of unambiguity to the unbounded fan-in classes AC^k, $k \geq 1$ (see [3, 7, 9]) and to the semi-unbounded fan-in classes SAC^k, $k \geq 1$, (see [10]), we obtain the families $UnambAC^k$ and $UnambSAC^k$ for every $k \geq 1$. In addition we consider $UnambSAC^k$-circuits which do not use robust OR-gates, which means that even the OR-gates of bounded fan-in are vulnerable. The languages accepted by uniform families of circuits of this type will be denoted by $UnambRAC^k$. (Clearly $(ambiguous)RAC^k = (ambiguous)SAC^k$). In [5] $DSPACE(\log n) \subseteq UnambRAC^1$ was shown, which is why we can use the usual log space uniformity condition of [7].

By $CREW^k$ we denote the class of languages recognized by some CREW-PRAM in time $O(\log^k n)$ using polynomially many processors [3, 4].

By $LOGUCFL$ we denote the class of all languages log space reducible to any unambiguous context-free language.

In the following we consider augmented push-down automata (see [2]). We make the technical assumption that accepting computations always end with an empty stack and an empty working tape and there is exactly one final state, i.e. there is exactly one accepting configuration. In addition we require the machine either to push or pop a symbol in every step. This does not restrict the power of nondeterministic or deterministic automata, but is useful when working with unambiguous automata. $NAPDA^k$ (resp. $DAPDA^k$) is the class of languages recognized in time $2^{O(\log^k n)}$ by a nondeterministic (resp. deterministic) pushdown automaton augmented with a log space bounded working tape [2]. In the usual way, we call an augmented push-down automaton *M (weakly)*

$\wedge$	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

$\vee$	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

Table 1: Boolean functions extended to domain $\{0, 1, \perp\}$

$\wedge!$	0	1	$\perp$
0	$\perp$	0	$\perp$
1	0	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

$\vee!$	0	1	$\perp$
0	0	1	$\perp$
1	1	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Table 2: Vulnerable boolean functions

unambiguous if for every input there is at most one accepting computation of M. M is strongly unambiguous if for every pair of configurations there is at most one computation path between them (here we do not mean surface-configurations). $UnambAPDA^k$ (resp. $StUnambAPDA^k$) is the class of languages recognized in time $2^{O(\log^k n)}$ by an unambiguous (resp. strongly unambiguous) pushdown automaton augmented with a log space bounded working tape.

Weakly unambiguous circuits

To define weak unambiguity in circuits formally, we extend in the usual way the boolean functions $\wedge$ and $\vee$ to domain $\{0, 1, \perp\}$ (see Table 1). In this context $\perp$ means "undefined" which is used as the outcome of a blown gate. Like a (strongly) unambiguous circuit a weakly unambiguous one consists of 'robust' gates of bounded fan-in and vulnerable gates of unbounded fan-in. But in contrast to the strong case in a weak circuit vulnerable gates inside the circuit are allowed to blow and to yield the value $\perp$, as long as the output gate always evaluates to either 0 or 1. Here we assume that vulnerable gates forward any $\perp$-input (see Table 2). The resulting weakly unambiguous classes are denoted by $WeakUnambAC^k$ (resp. $WeakUnambSAC^k$, $WeakUnambRAC^k$).

Results

We will now show that $LOGUCFL$ can be recognized by $UnambRAC^1$-circuits. For unambiguous linear languages this was done in [5] by an *exact* decomposition of the Savitch-algorithm. Here we are able to do this for unambiguous context-free languages by exhibit-

ing an exact decomposition of the algorithm of Lewis, Stearns, and Hartmanis. Rytter showed $LOGUCFL \subseteq CREW^1$ by quite different methods to ours, but we need some of his ideas. Rytter's algorithm for a CREW-PRAM can be transfered to unambiguous networks with nearly no changes. This was done in [5] yielding $LOGUCFL \subseteq UnambAC^1$. The stronger result $LOGUCFL \subseteq UnambRAC^1$ cannot be shown by this approach because *negations* are essentially needed in Rytter's construction when computing the maximal element in a path. We will preserve unambiguity by a simpler but somewhat tricky approach. A unique tree decomposition will be chosen not only by computations done by the circuit itself, but also by computations done by the *circuit constructor* leading to a special shape of the circuit that prevents any ambiguity.

Let $G = (N, T, P, S)$ be an unambiguous context-free grammar in Chomsky normal form without useless nonterminals. We denote productions by $A \to v$ and derivations by $u \stackrel{*}{\Rightarrow} v$. Let $w = a_1\, a_2 \ldots a_3$ be an input string of length n. According to Rytter in [8] we denote the substring $a_{i+1} \ldots a_j$ of w by w_{ij}. We call (A, i, j), $A \in N$, $0 \leq i < j \leq n$ a *node*. A node (A, i, j) is said to be *realizable* iff $A \stackrel{*}{\Rightarrow} w_{ij}$ and a pair of nodes $((A, i, j), (B, k, l))$ is said to be *realizable* iff $A \stackrel{*}{\Rightarrow} w_{ik} B w_{jl}$ and $(A, i, j) \neq (B, k, l)$. We define $y, z \vdash x$ and $z, y \vdash x$ iff $x = (A, i, j)$, $y = (B, i, k)$, $z = (C, k, j)$, and $A \to BC$.

To determine whether a node or a pair of nodes is realizable we will characterize them by recursive equations:

Proposition 1 *A node $x = (A, i, j)$ is realizable iff*

(i) $A \to w_{ij}$ *and* $i + 1 = j$

or (ii) *There exists a realizable node y and (x, y) is realizable, too.*

Proposition 2 *A pair of nodes (x, y), $x = (A, i, j)$, is realizable iff*

(i) $y = (B, i, j-1)$, $A \to BC$, *and* $C \to w_{j-1,j}$

or (ii) $y = (B, i+1, j)$, $A \to CB$, *and* $C \to w_{i,i+1}$

or (iii) *There exists a node y_1 such that (x, y_1), (y_1, y) are both realizable.*

A node x can be interpreted as a syntactic tree and a pair (x, y) as a tree with a gap, i.e., all leaves but one are terminals and the nonterminal leaf is y. We will construct a circuit with gates labeled $\langle x \rangle$ that computes whether x is realizable and with gates labeled $\langle x, y \rangle$ computing whether the pair (x, y) is realizable on a given input string w. This can be done using the recursive equations of Proposition 1 and 2.

If $i + 1 = j$ then a gate $\langle (A, i, j) \rangle$ directly checks the input string. In other cases

$$\langle x \rangle \equiv \bigvee_{y, y_1, y_2} \langle x, y \rangle \wedge \langle y_1 \rangle \wedge \langle y_2 \rangle \tag{1}$$

can be used, where y, y_1, y_2 are nodes such that $y_1, y_2 \vdash y$. Cases (i) and (ii) of Proposition 2 can directly be used to construct the gate $\langle x, y \rangle$. When case (iii) holds,

$$\langle x, y \rangle \equiv \bigvee_{y_1, y_2, y_3} \langle x, y_1 \rangle \wedge \langle y_2, y \rangle \wedge \langle y_3 \rangle \tag{2}$$

will be used. y_1, y_2, y_3 are nodes such that $y_2, y_3 \vdash y_1$.

The correctness, polynomial size, and uniformity of the circuit constructed by now is straightforward. However, it is neither unambiguous nor does it have logarithmic depth. To gain these additional properties, we will use a *both unique and balanced* decomposition of syntactic trees.

We denote the number of leaves of a given node x in a binary tree by $size(x)$. The size of nodes and pairs of nodes is easily computed as $size(A,i,j) = j - i$ and $size((A,i,j),(B,k,l)) = k - i + j - l$.

Lemma 3 *Let T be a binary tree with at least two leaves and root x. Then there exists a unique node $y \in T$ with sons y_1 and y_2 such that $size(y) > h$ and $size(y_1), size(y_2) \leq h$, where $h := \left\lceil \frac{size(x)}{2} \right\rceil$.*

Proof. (sketch) The root x fulfils $size(x) > h$. If the other two conditions do not hold for x, the first condition again holds for some son of x. So one can "climb down" the tree until he finds the desired node. If there is a y and a y' which fulfils Lemma 3, each must be predecessor of the other due to the conditions that $size(y)$ and $size(y')$ must suffice, so $y = y'$.

Lemma 4 *Let T be a binary tree with at least two leaves, x its root, and y one of its leaves. Then there exists a unique node $y_1 \in T$ with sons y_2 and y_3 such that y_1 and y_2 lie on the path from x to y and $size(y_2) \leq \left\lceil \frac{size(x)}{2} \right\rceil \leq size(y_1)$ holds.*

Proof. (sketch) There is a strictly monotone decrease of the size of a node on the path from x to y yielding both existence and uniqueness of y_1.

Lemma 3 and 4 will now be used to restrict the domain of the $\exists$-gates of $\langle x \rangle$ and $\langle x,y \rangle$. y, y_1, and y_2 in (1) are restricted by the additional conditions $size(y) > h$ and $size(y_1), size(y_2) \leq h$ for $h = \left\lceil \frac{size(x)}{2} \right\rceil$. From Lemma 3 and the fact that there is at most one syntactic tree deriving x follows that there exists *exactly* one y fulfilling the restricted conditions iff there exists *at least* one y fulfilling the unrestricted conditions, what is equivalent to 'x is realizable'. We can simultaneously make y, y_1, and y_2 unique if we additionally claim $y_1 \ll y_2$ where $\ll$ is some logspace computable total order on nodes. We are now sure that gates $\langle x \rangle$ are unambiguous. Checking the sizes of y, y_1, and y_2 shows that $\langle x \rangle$ is computed by gates having at most only half the size of x.

In (2) we restrict y_1, y_2, y_3 by $size(y_2,y) \leq \left\lceil \frac{size(x,y)}{2} \right\rceil \leq size(y_1,y)$. Lemma 4 yields correctness because there is at most one syntactic tree with gap y deriving x since the gap can be closed ny some terminal string which is derived by y (there are no useless nonterminals). The uniqueness of y_1 also follows from Lemma 4 and y_2, y_3 are unique, too, because exactly one of the sons of y_1 lies on the path from x to y.

Again, we can see that $size(x,y_1)$ and $size(y_2,y)$ are at most half of $size(x,y)$, but $size(y_3)$ can nearly be as large as $size(x,y)$ itself. However, y_3 is not a pair of nodes, so it will be cut in halfes in the next step (compare this with [7]).

Clearly the circuit has logarithmic depth because the size decreases by a factor of $\frac{1}{2}$ at least every four levels and the output node, labeled $(S,0,n)$, has a size of n. Unambiguity

was also proved and uniformity is straightforward. Since $UnambRAC^1$ is closed under logspace reductions [5] we gain:

Theorem 5 *Each language logspace reducible to an unambiguous context-free language can be recognized by an $UnambRAC^1$-circuit, i.e., $LOGUCFL \subseteq UnambRAC^1$.*

Using the methods used to prove Theorem 5 we will now show $UnambAPDA^1 \subseteq WeakUnambRAC^1$ and $StUnambAPDA^1 \subseteq UnambRAC^1$. By capital letters we will denote surface configurations of unambiguous augmented pushdown automata. Surface configurations, we consider, contain the topmost symbol and height of the pushdown store, the final state, and the contents of the auxiliary tape. Let M be an $UnambAPDA^1$ or $StUnambAPDA^1$. We assume without loss of generality that M pushes or pops exactly one symbol during each step and accepts by empty stack. M is time-bounded by some polynom $p(n)$. We call (A, B, i, j) a *node* if A and B are surface configurations and $0 \leq i \leq j \leq p(n)$. A node (A, B, i, j) is *realizable* iff there exists a computation $A \vdash^{j-i} B$ from A to B in exactly $j - i$ steps starting with surface configuration A, the head on the ith symbol of the input tape and the level of the pushdown is the same for A and B and does not go below this level during the computation. Obviously, M accepts iff some node $(S, F, 0, j)$ is realizable where S is the start surface configuration and F is a final surface configuration. We write $x, y \vdash z$ and $y, x \vdash z$ iff

(i) $x = (C, D, i + 1, k)$, $y = (E, B, k + 1, j)$, $z = (A, B, i, j)$

(ii) The level of the pushdown store is equal for A, E, and B.

(iii) There exists a computation $A \vdash^1 C$ from A to C in one step starting with the input head on the ith symbol of the input tape and pushing a onto the pushdown store during this step.

(iv) There exists a computation $D \vdash^1 E$ from D to E in one step starting with the input head on the $(i + 1)$-st symbol of the input tape and popping a from the pushdown store.

The next Lemma is straightforward:

Lemma 6 *A node $x = (A, B, i, j)$ is realizable iff $A = B$ and $i = j$ or there exist nodes y and z such that $y, z \vdash x$. If there is only one computation $A \vdash^{j-i} B$ from A to B starting at the i-th symbol, then there exists at most one y and z with $y, z \vdash x$.*

We define a set T of binary trees consisting of nodes. A binary tree is contained in T iff the leaves are of shape (A, A, i, i) and for each $x, y, z \in T$ holds $y, z \vdash x$ if y and z are the sons of x. A simple consequence of Lemma 6 is

Lemma 7 *A node x is realizable iff there is a binary tree in T with root x.*

We are now ready to consider *pairs of nodes*. We call a pair of nodes (x, y), $x \neq y$ *realizable* iff there is some binary tree with root x, one leaf is y and all other leaves are of shape (A, A, i, i) and for each node $\langle x \rangle$ in this tree again holds $y, z \vdash x$ if y and z are

the sons of x. These trees are like those in T but with a "gap" y. If $size(x)$ denotes the number of leaves of a tree with root x, then $size(x) = \frac{j-i}{2} + 1$ for $x = (A, B, i, j)$. If $size(x, y)$ denotes the number of leaves of a tree with gap y and root x then $size(x, y) = \frac{j-i+k-l}{2} + 1$ for $x = (A, B, i, j)$ and $y = (C, D, k, l)$. Using these equalities, new recursive equations can be given for a node x:

Lemma 8 *A node x is realizable iff*

(i) $x = (A, A, i, i)$

or (ii) *There exists some realizable node y and (x, y) is also realizable.*

Lemma 9 *A pair of nodes (x, y) is realizable iff*

(i) *There exists some realizable node z such that $y, z \vdash x$ holds.*

or (ii) *There exists some node y_1 such that (x, y_1) and (y_1, y) are both realizable.*

The construction of the simulating circuit can now be given. Gates labeled $\langle x \rangle$ will compute whether x is realizable. If $x = (A, A, i, i)$ then $\langle x \rangle \equiv 1$. Otherwise $\langle x \rangle$ is defined as

$$\langle x \rangle \equiv \underset{y,y_1,y_2}{\exists} \langle x, y \rangle \wedge \langle y_1 \rangle \wedge \langle y_2 \rangle, \tag{3}$$

where y, y_1, and y_2 are nodes such that $y_1, y_2 \vdash y$ and $y_1 \ll y_2$ holds. $\ll$ is again a logspace computable total order. y_1, y_2, and y_3 are further restricted by $size(y_1), size(y_2) \leq \left\lceil \frac{size(x)}{2} \right\rceil < size(y)$. Gates labeled $\langle x, y \rangle$ compute whether the pair (x, y) is realizable. They are definded as follows:

$$\langle x, y \rangle \equiv (\underset{z}{\exists} \langle z \rangle) \vee (\underset{y_1,y_2,y_3}{\exists} \langle x, y_1 \rangle \wedge \langle y_2, y \rangle \wedge \langle y_3 \rangle) \tag{4}$$

y_1, y_2, y_3, and z are nodes restricted by $y, z \vdash x$, $y_1, y_2 \vdash y_3$ and $size(y_2, y) \leq \left\lceil \frac{size(x,y)}{2} \right\rceil \leq size(y_1, y)$. The correctness of this circuit follows from the recursive equations in Lemma 8 and 9 and the fact that consideration only of balanced nodes is no restriction (Lemma 3 and 4). The circuit is uniform since sizes of nodes and pairs of nodes are logspace computable. It has logarithmic depth because of the same reasons as in Theorem 5.

Suppose there is only one tree in T with root x. Then each of its subtrees is unique, too. The cases (i) and (ii) are disjoint in Lemma 8 and 9. The $\exists$-gates in (3) and (4) are vulnerable because at most one combination of y, y_1, y_2 in (3), respectively of y_1, y_2, y_3, and z in (4) holds. If the simulated automaton is strictly unambiguous all trees are unique and no gate will be blown. Therefore the circuit is itself strictly unambiguous. However, if the automaton is only weakly unambiguous, then some gates may be blown. The output of the circuit is $\underset{B,i}{\exists} \langle S, B, 0, i \rangle$, where S is the start configuration and $0 \leq i \leq p(n)$ the number of steps. Since each computation represented by $(S, B, 0, i)$ is an accepting one, a tree with root $(S, B, 0, i)$ must be unique. So neither $\langle S, B, 0, i \rangle$ nor its predecessors in the circuit can be blown because each predecessor of $\langle S, B, 0, i \rangle$ is also contained in the tree with root $(S, B, 0, i)$ and therefore unique. These results can be summarized as

Lemma 10 $UnambAPDA^1 \subseteq WeakUnambRAC^1$, $StUnambAPDA^1 \subseteq UnambRAC^1$

The next two results show that the reverse of Lemma 10 is also true.

Lemma 11 $WeakUnambRAC^1 \subseteq UnambAPDA^1$, $UnambRAC^1 \subseteq StUnambAPDA^1$

Proof. Let M be an *UnambAPDA*. M checks whether a given circuit accepts an input string w by evaluating its root $\langle x \rangle$. Our construction is similar to the corresponding one of [6, 10]. But here M has to push *idle* stack entries in order to work strongly unambiguously in case of rejection. A gate $\langle y \rangle$ is evaluated as follows:

- If $\langle y \rangle$ is an $\exists$-gate or an bounded OR-gate, it is marked "idle" and pushed onto the pushdown store. Then M guesses a son $\langle y' \rangle$ and recursively checks if $\langle y' \rangle$ evaluates to 1.
- If $\langle y \rangle$ is an AND-gate, it is marked "idle" and pushed onto the stack, then M computes its sons $\langle y_1 \rangle$ and $\langle y_2 \rangle$. Let $\langle y_1 \rangle$ be smaller than $\langle y_2 \rangle$ with regard to some total order. M pushes $\langle y_2 \rangle$ onto the stack and evaluates recursively whether $\langle y_1 \rangle$ evaluates to 1.
- If $\langle y \rangle$ is an input to the circuit, M rejects if this input has value 0. If it has value 1, M accepts if the stack is empty and otherwise pops a gate $\langle y' \rangle$ from the pushdown and checks whether $\langle y' \rangle$ evaluates to 1. However, if $\langle y' \rangle$ is marked "idle", M does not evaluate $\langle y' \rangle$ but continues popping gates until it finds one which is not "idle" or the stack is empty. In the latter case M accepts.

Besides the additional use of "idle" gates our simulation is essentially the same as in [10], where $SAC^1 \subseteq NAPDA^1$ was shown. The correctness of the algorithm and the polynomial time constraint holds as proved in [10]. All left to do is to show that M is unambiguous:

1. $WeakUnambRAC^1 \subseteq UnambAPDA^1$:

During an *accepting* computation, M cannot check a vulnerable gate with more than one true input. Since the circuit is weakly unambiguous there must be some AND-gate $\langle y \rangle$ that evaluates to 0 on each path from the blown gate to the root. M must evaluate $\langle y \rangle$ and then rejects.

If M accepts it must not guess a false input of any $\exists$-gate it considers during its computation, so all guesses have to be correct. However, there is at most one right guess per gate, so there is at most one accepting computation.

2. $UnambRAC^1 \subseteq StUnambAPDA^1$:

This part of the proof is a little awkward. We call a computation sequence of M an *accepting cycle* iff the sequence starts with pushing some $\exists$-gate that is marked "idle" onto the stack and it finishes with popping it. Obviously, the first and last configuration in an accepting cycle are mates and the outcome of some $\exists$-gate $\langle y \rangle$ is 1 iff there is some accepting cycle starting with pushing $\langle y \rangle$. A computation sequence is called a *rejecting cycle* iff it again starts with pushing some $\exists$-gate $\langle y \rangle$ that is marked "idle" and it finishes by rejection *before* $\langle y \rangle$ is popped from the stack.

M behaves unambiguously during an accepting cycle because $\langle y \rangle$ can be interpreted as the output gate of a circuit that evaluates to 1. We will now prove that M behaves unambiguously during a rejecting cycle, too: When M rejects, its pushdown store contains a number of "idle" gates. From these gates the path from $\langle y \rangle$ to the input that lead to rejection can be concluded. M followed this path during its computation. Suppose there is another rejecting cycle ending in the same configuration. Then both computations followed the same path P, what is only possible if the guesses at $\exists$-gates lying in P were also the same. Let us now take a closer look at some AND-gate $\langle x \rangle \in P$ (with sons $\langle x_1 \rangle$ and $\langle x_2 \rangle$). One son (e.g., $\langle x_1 \rangle$) lies in P, too. If $\langle x_1 \rangle$ is smaller than $\langle x_2 \rangle$ then $\langle x_2 \rangle$ was checked in neither rejecting cycle. Otherwise $\langle x_2 \rangle$ was checked in both cycles before the computation continued to check $\langle x_1 \rangle$. $\langle x_2 \rangle$ was evaluated to 1 because otherwise rejection would take place before M starts to check $\langle x_1 \rangle$. The evaluation of $\langle x_2 \rangle$ is done in an accepting cycle, so both computations are also identical while checking $\langle x_2 \rangle$. By now we can say that M is unambiguous during *cycles*.

Assume that M is in some (not necessarily reachable) configuration. We cannot foretell how M reacts, but M will behave deterministically until it starts to check some $\exists$-gate. M enters then a (rejecting or accepting) cycle during which M is unambiguous. After an accepting cycle M pops some weired content from the stack trying to treat it like a gate. Again we cannot tell what M will do, but it will do it deterministically until M again checks some $\exists$-gate.

A careful inspection of the proof of Lemma 11 shows that circuits of depth greater than $O(\log n)$ can also be simulated. A depth of $O(\log^k n)$ leads to recognition in time $2^{O(\log^k n)}$:

Corollary 12 $\mathit{WeakUnambRAC}^k \subseteq \mathit{UnambAPDA}^k$,
$\mathit{UnambRAC}^k \subseteq \mathit{StUnambAPDA}^k$

Lemmata 10 and 11 imply a main result of this paper: The languages recognized by *UnambAPDA* (*StUnambAPDA* resp.) in polynomial time correspond to those recognized by *WeakUnambRAC* (*UnambRAC* resp.) with logarithmic depth.

Theorem 13 $\mathit{WeakUnambRAC}^1 = \mathit{UnambAPDA}^1$,
$\mathit{UnambRAC}^1 = \mathit{StUnambAPDA}^1$

In this context the question arises whether Lemma 10 and thereby Theorem 13 are also true for arbitrary powers of the log function.

References

[1] A. K. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28:114–133, 1981.

[2] S. A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18:4–18, 1971.

[3] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Inform. and Comp.*, 64:2–22, 1985.

[4] P. Dymond and W. Ruzzo. Parallel RAMs withs owned global memory and deterministic language recognition. In *Proc. of 13th ICALP, Number 226 in LNCS*, pages 95–164. Springer, 1987.

[5] K.-J. Lange. Unambiguity in circuits. Submitted, 1989.

[6] W. L. Ruzzo. Tree-size bounded alternation. *J. Comput. Syst. Sci.*, 21:218–235, 1980.

[7] W. L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22:365–383, 1981.

[8] W. Rytter. Parallel time $O(\log n)$ recognition of unambiguous context-free languages. *Inform. and Comp.*, 73:75–86, 1987.

[9] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, May 1984.

[10] H. Venkateswaran. Properties that characterize LOGCFL. In *Proc. of 19th STOC*, pages 141–150, 1987.