Arithmetizing classes around NC^1 and L

Nutan Limaye, Meena Mahajan, and B. V. Raghavendra Rao

The Institute of Mathematical Sciences, Chennai 600 113, India. {nutan,meena,bvrr}@imsc.res.in

Abstract. The parallel complexity class NC^1 has many equivalent models such as polynomial size formulae and bounded width branching programs. Caussinus *et al.* [CMTV98] considered arithmetizations of two of these classes, $\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$. We further this study to include arithmetization of other classes. In particular, we show that counting paths in branching programs over visibly pushdown automata is in $\mathsf{FLogDCFL}$, while counting proof-trees in logarithmic width formulae has the same power as $\#\mathsf{NC}^1$. We also consider polynomial-degree restrictions of SC^i, denoted sSC^i, and show that the Boolean class sSC^1 is sandwiched between NC^1 and L, whereas sSC^0 equals NC^1. On the other hand, the arithmetic class $\#\mathsf{sSC}^0$ contains $\#\mathsf{BWBP}$ and is contained in FL, and $\#\mathsf{sSC}^1$ contains $\#\mathsf{NC}^1$ and is in SC^2. We also investigate some closure properties of the newly defined arithmetic classes.

1 Introduction

The parallel complexity class NC^1, comprising of languages accepted by logarithmic depth, polynomial size, bounded fan in Boolean circuits, is of fundamental interest in circuit complexity. NC^1 is known to be contained within logarithmic space L. The classes NC^1 and L have many equivalent characterizations, notably in terms of branching programs and small-width circuits. Bounded width branching programs BWBP, as well as bounded width circuits SC^0, (both of polynomial size), were shown by Barrington [Bar89] to be equivalent to NC^1, while it is folklore that polynomial size $O(\log n)$ width circuits SC^1 equals L.

However, arithmetizations of these classes are not necessarily equivalent. In [CMTV98], Caussinus *et al.* proposed three arithmetizations of NC^1: (1) counting proof-trees in an NC^1 circuit, (2) computation by a polynomial size log depth circuit over $+$ and $\times$, and (3) counting paths in a nondeterministic bounded width branching program. It is straightforward to see that the first two definitions of function classes, over $\mathbb{N}$, coincide (see for instance [Vin91,Vol99]); and this class is denoted $\#\mathsf{NC}^1$. It is shown in [CMTV98] that the third class, $\#\mathsf{BWBP}$, is contained in $\#\mathsf{NC}^1$, though the converse inclusion is still open. (However, the arithmetizations over $\mathbb{Z}$ are shown to coincide.) Also, using the programs over monoids framework, [CMTV98] observe that $\#\mathsf{BWBP}$ equals $\#\mathsf{BP\text{-}NFA}$, the class of functions that count the number of accepting paths in a nondeterministic finite-state automaton NFA when run on the output of a deterministic branching program. It is known (see *e.g.* [All04,Vol99]) that $\#\mathsf{NC}^1$ has Boolean polynomial size circuits of depth $O(\log n \log^* n)$ and is thus very close to NC^1. It follows from more recent results [CDL01] that $\#\mathsf{NC}^1$ is contained in FL; see *e.g.* [All04].

We continue this study here (and also extend it to L) by arithmetizing other Boolean classes also known to be equivalent to NC^1. The first generalization we consider is from NFA to VPA. If we generalize NFA all the way to arbitrary pushdown automata PDA, we get the well-studied Boolean and arithmetic classes LogCFL and $\#\mathsf{LogCFL}$ respectively (languages

logspace many-one reducible to some context free language), containing the Boolean and arithmetic analogues of nondeterministic logspace NL and #L. A non-trivial restriction of PDA is the class of languages accepted by visibly pushdown automata VPA. These are PDA with no ϵ-moves, whose stack behaviour (push/pop/no change) is dictated solely by the input letter under consideration. They are also referred to as input-driven PDA, and have been studied in [Meh80,BV83,Dym88,AM04,AKMV05]. In [Dym88], languages accepted by such PDA are shown to be in NC^1, while in [AM04] it is shown that such PDA can be determinized. Thus they lie properly between regular languages and deterministic context-free languages, and membership is complete for NC^1. The arithmetic version we consider is #BP-VPA, counting the number of accepting paths in a VPA, when run on the output of a deterministic branching program. It is clear that this contains #BP-NFA. We had claimed in a preliminary version of this paper [LMR07] that in fact the two are equal, and thus adding a stack to an NFA but restricting usage of the stack to a visible or input-driven nature adds no power to the closure of the class under projections. Unfortunately, the proof in [LMR07] is incorrect, and we do not know an alternative proof. What we do show here is that functions in #BP-VPA can be evaluated in the deterministic analogue of LogCFL, FLogDCFL.

The next class we consider is arithmetic formulae. It is known that formulae F (circuits with fanout 1 for each gate) and even logarithmic width formulae LWF have the same power as NC^1 [IZ94]. Applying either of definition (1) or (2) above to formulae gives the function classes #F and #LWF. It is known [BCGR92] that $\#\mathsf{LWF} \subseteq \#\mathsf{F} = \#\mathsf{NC}^1$. We show that this is in fact an equality. Thus even in the arithmetic setting, LWF have the full power of NC^1.

Next we consider bounded width circuits. SC is the class of polynomial size poly-logarithmic width (width $O(\log^i n)$ for SC^i) circuits, and corresponds in the uniform setting to a simultaneous time-space bound. (SC stands for Steve's Classes, named after Stephen Cook who proved the first non-trivial result about polynomial time log-squared space PLoSS, *i.e.* SC^2, in [Coo79]. See for instance [Joh90].) It is known that SC^0 equals NC^1 [Bar89].

However, this equality provably does not carry over to the arithmetic setting, since it is easy to see that even SC^0 over $\mathbb{N}$ can compute values that are infeasible (needing super-polynomially long representation). So we consider the restriction to polynomial degree, denoted by sSC^0, before arithmetizing to get $\#\mathsf{sSC}^0$. We observe that in the Boolean setting, this is not a restriction at all; sSC^0 equals NC^1 as well. However, the arithmetization does not appear to collapse to either of the existing classes. We show that $\#\mathsf{sSC}^0$ lies between #BWBP and FL.

The polynomial-degree restriction of SC^0 immediately suggests a similar restriction on all the SC^i classes. We thus explore the power of sSC^i and sSC, the polynomial-degree restrictions of SC^i and SC respectively, and their corresponding arithmetic versions $\#\mathsf{sSC}^i$ and #sSC. This restriction automatically places the corresponding classes in LogCFL and #LogCFL, since LogCFL is known to equal languages accepted by polynomial size polynomial degree circuits [Sud78,Ruz80], and since the arithmetic analogue also holds [Vin91,NR95]. Thus we have a hierarchy of circuit classes between NC^1 and LogCFL. Other hierarchies sitting in this region are (1) polynomial size branching programs of polylog width, limited by NL in LogCFL, and (2) polynomial size log depth circuits with AND fan in 2 and OR fan in

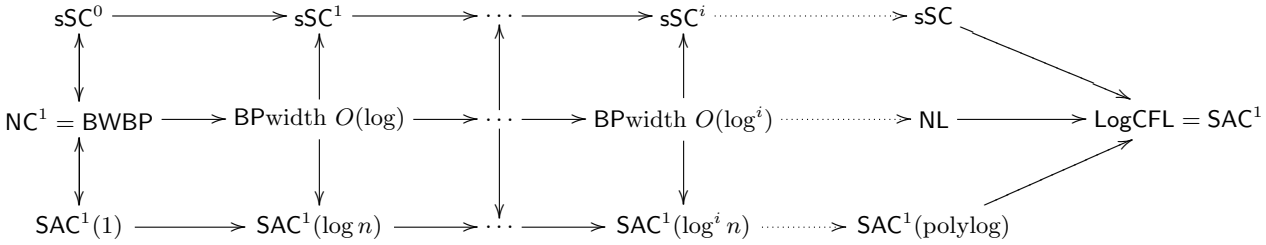

Fig. 1. Hierarchies of classes between NC^1 and LogCFL

polylog, limited by SAC^1 which equals LogCFL [Ven91]; see [Vin96]. We denote the i-th level of the latter hierarchy, *i.e.* poly size, log depth circuits with AND fan in 2 and OR fan in $O(\log^i n)$, by $\mathsf{SAC}^1(\log^i n)$. In both of these hierarchies, [Vin96] establishes closure under complementation. For sSC^i, we have a weaker result: co-sSC^i is contained in sSC^{2i}. Figure 1 shows these hierarchies and their relationships.

It is not clear what power the Boolean class sSC^1 possesses: is it strong enough to equal SC^1, or is the polynomial degree restriction crippling enough to bring it down to SC^0=NC^1? We show that all of $\#\mathsf{NC}^1$ is captured by $\#\mathsf{sSC}^1$, which is contained in Boolean SC^2. Note that the maximal fragments of NC hitherto known to be in SC were LogDCFL [Coo71,DC89,FLR96] and randomized logspace RL [Nis94]; we do not know how this fragment compares with them. In fact, turning the question around, studying sSC is an attempt to understand fragments of SC that lie within NC.

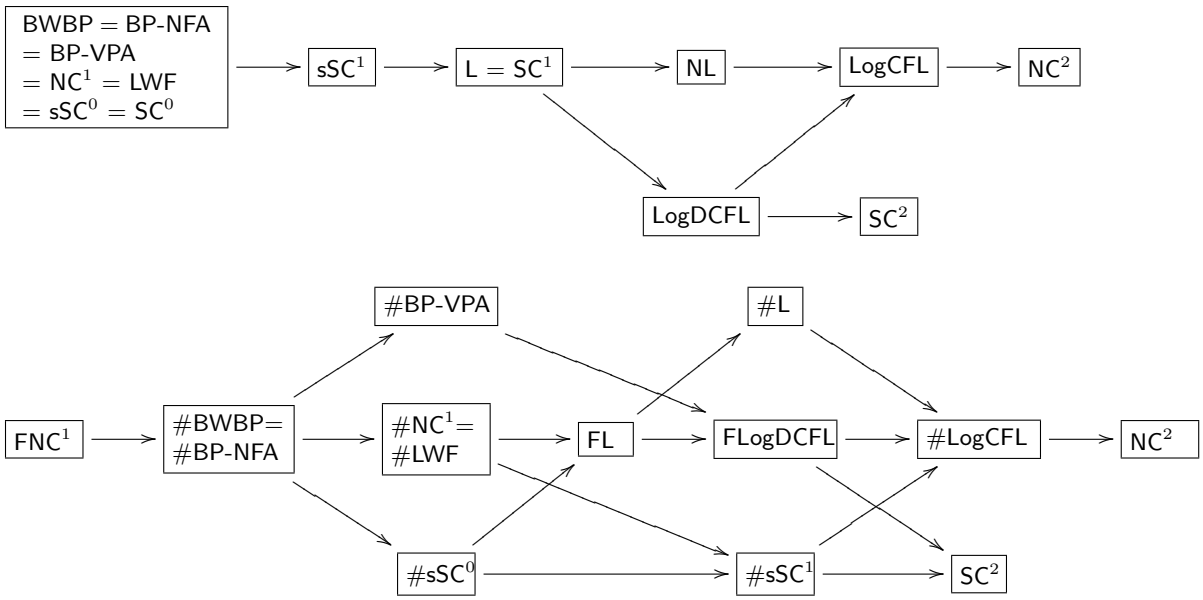

Fig. 2. Boolean classes and their arithmetizations

Our main results can be summarized in Figure 2. It shows that corresponding to Boolean NC^1, there are four naturally defined arithmetizations, while the correct arithmetization of L is still not clear. We also show that three of the arithmetizations of NC^1 coincide under modulo tests, for any fixed modulus; see Figure 3. For the fourth arithmetization, #BP-VPA, we show that a modulus test for any fixed modulus is in L.

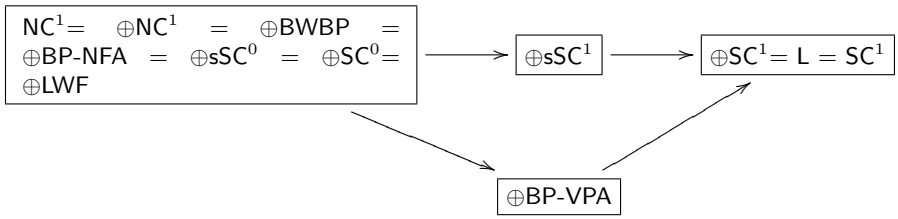

Fig. 3. Parity Classes around NC^1

A key to understanding function classes better is to investigate their closure properties. We present some such results concerning $\#\mathsf{sSC}^i$.

This paper is organized as follows. Definitions and notation are presented in Section 2. Sections 3 and 4 present the bounds on #BP-VPA and #LWF, respectively. Section 5 introduces and presents bounds involving sSC^i and $\#\mathsf{sSC}^i$. Some closure properties of these classes are presented in Section 6, where the collapse of the modulus test classes: $\mathsf{NC}^1 = \oplus\mathsf{NC}^1 = \oplus\mathsf{BWBP} = \oplus\mathsf{sSC}^0$ is also presented (as shown in Figure 3).

2 Preliminaries

Machine classes: L and NL denote the classes of languages accepted by deterministic and nondeterministic logspace-bounded machines. LogCFL denotes the class of languages logspace many-one reducible to some context-free language CFL, and is equivalently characterised as the class of languages accepted by AuxPDA(poly), nondeterministic logspace machines when augmented with a pushdown stack but restricted to halt within polynomial time. Deterministic counterparts of LogCFL and AuxPDA(poly), namely $\mathsf{LogDCFL}$ and $\mathsf{DAuxPDA}$(poly), are similarly defined and are also known to be computationally equivalent.

Boolean circuit classes: By NC^1 we denote the class of languages which can be accepted by a family $\{C_n\}_{n\geq 0}$ of polynomial size $O(\log n)$ depth bounded circuits, with each gate having a constant fanin. A branching program is a layered acyclic graph G with edges labeled by constants or literals, and with two special vertices s and t. It accepts an input x if it has an $s \rightsquigarrow t$ path where each edge is labeled by a true literal or the constant 1; we call such a path a valid path on input x. BWBP denotes the class of languages that can be accepted by families of polynomial size bounded width branching programs $\{G_n\}_{n\geq 0}$, where the graph G_n considers n variables. BWC is the class of languages which can be accepted by a family

$\{C_n\}_{n\geq 0}$ of constant width, polynomial size circuits, where *width* of a circuit is the maximum number of gates at any level of the circuit. Here the circuit is assumed to be layered: a gate at layer i can receive as input either a constant, or a circuit input, or the output of a gate at layer $i-1$. A branching program can be equivalently viewed as a skew circuit, i.e., a circuit in which each AND gate has at most one input wire that is the output of another gate of the circuit rather than a circuit input (either an input variable or its negation or a constant); hence BWBP is in BWC. SC^i is the class of languages which can be accepted by a family $\{C_n\}_{n\geq 0}$ of polynomial size circuits of width $O(\log^i n)$. Thus we have by definition, $\mathsf{BWC} = \mathsf{SC}^0$. For the class SC^i we assume, without loss of generality, that every gate has fan-in $O(1)$ (fan-in $f = O(\log^i n)$ is replaced by a width $O(1)$, depth $O(f)$ circuit). In the uniform setting, the class SC^i is equivalent to the class of languages accepted by deterministic Turing machines which use $O(\log^i n)$ space and run in polynomial time (see [Coo79] and [Joh90]). LWF is the class of languages which can be accepted by a family $\{F_n\}_{n\geq 0}$ of polynomial size formulae with width bounded by $O(\log n)$. Without the width bound, denote the family of polynomial size formula by F. By TC^0 we denote the class of languages which can be accepted by a family $\{C_n\}_{n\geq 0}$ of polynomial size $O(1)$ depth bounded circuits, with unbounded fanin AND, OR and Majority gates.

Visibly pushdown automata: A visibly pushdown automaton (VPA) is a PDA $M = (Q, Q_{in}, \Delta, \Gamma, \delta, Q_F)$ working over an input alphabet Δ that is partitioned as $(\Delta_c, \Delta_r, \Delta_{int})$. Q is a finite set of states, $Q_{in}, Q_F \subseteq Q$ are the sets of initial and final states respectively, Γ is the stack alphabet containing a special bottom-of-stack marker $\perp$, and acceptance is by final state. The transition function δ is constrained so that if $a \in \Delta_c$, then $\delta(p,a) = (q,\gamma)$ (push move, independent of top-of-stack). If $a \in \Delta_r$, then $\delta(p,a,\gamma) = q$ (pop), $\delta(p,a,\perp) = q$ (pop on empty stack). If $a \in \Delta_{int}$, then $\delta(p,a) = q$ (internal move independent of the top-of-stack).

The input letter completely dictates the stack movement. Also the pushdown automata is assumed to be ϵ-move-free, while δ is allowed to be non-deterministic.

Programs over automata: For defining branching programs over automata, we follow the notation from [CMTV98]. A nondeterministic automaton is a tuple of the form $(Q, \Delta, q_0, \delta, F)$, where Q is the finite set of states, Δ is the input alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\delta : Q \times \Delta \to \mathcal{P}(Q)$.

A projection $P = (\Sigma, \Delta, S, B, E)$ over Δ is a family $P = (P_n)_{n\in N}$ of n-projections over Δ, where an n-projection over Δ is a finite sequence of pairs (i, f) with $1 \leq i \leq n$ and $f : \Sigma \to \Delta$. The length of the sequence is denoted by S_n, its j-th instruction is denoted by $(B_n(j), E_n(j))$ where $S : \mathbb{N} \to \mathbb{N}$, $B : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $E : \mathbb{N} \times \mathbb{N} \to \Delta^{\Sigma}$. B pulls out a letter $x_{B_{|x|}(j)} \in \Sigma$ from the input x and E projects it to a letter in the alphabet Δ. Thus the string $x \in \Sigma^*$ is projected to a string $P(x) \in \Delta^*$. P is said to be $\mathsf{FDLOGTIME}$ uniform if on input $\langle x, j\rangle$, the jth letter of $P(x)$ can be computed in determinsitic logarithmic time; see [CMTV98] for a more detailed definition.

A branching program over a nondeterministic automaton $N = (Q, \Delta, q_0, \delta, F)$ is a projection $P = (\Sigma, \Delta, S, B, E)$. It accepts $x \in \Sigma^*$ if N accepts the projection of x. $\mathsf{BP\text{-}NFA}$

is the class of all languages recognized by uniform polynomial length programs over a non-deterministic automaton[1]. Analogously, BP-VPA is the class of all languages recognized by uniform polynomial length programs over a VPA.

Arithmetic classes: We now define the corresponding arithmetic classes.

$$\#\mathsf{NC}^1 = \left\{ f : \{0,1\}^* \to \mathbb{N} \;\middle|\; \begin{array}{l} f \text{ can be computed by a polynomial size } O(\log n) \\ \text{depth bounded fanin circuit over } \{+, \times, 1, 0, x_i, \overline{x_i}\}. \end{array} \right\}$$

$$\#\mathsf{BP\text{-}NFA} = \left\{ f : \{0,1\}^* \to \mathbb{N} \;\middle|\; \begin{array}{l} f(x) = \#\text{accept}(P_{|x|}, x) \text{ for some uniform} \\ \text{polynomial length BP } P \text{ over an NFA } N \end{array} \right\}$$

Here, $\#\text{accept}(P, x)$ denotes the number of distinct accepting paths of N on the projection of x, $P(x)$.

$$\#\mathsf{BWBP} = \left\{ f : \{0,1\}^* \to \mathbb{N} \mid \begin{array}{l} \exists P \in \mathsf{BWBP}, \forall x \in \{0,1\}^* \\ f(x) = \text{number of valid paths on } x \text{ in } P \end{array} \right\}$$

For each of these counting classes, the corresponding Diff classes are defined by taking the difference of two # functions, while the Gap classes are defined by taking closure of the class under subtraction (equivalently, by allowing the constant -1 in the circuit). For reasonable classes (in particular, for the classes we consider), Diff and Gap coincide, see [Vol99].

Definition 1. *For any function class $\#\mathcal{C}$, $\mathsf{Mod}_p\mathcal{C}$ denotes the class of languages L such that there is an $f \in \#\mathcal{C}$ satisfying*

$$\forall x \in \{0,1\}^* : \quad x \in L \iff f(x) \equiv 0 \mod p$$

$\mathsf{C}_=\mathcal{C}$ denotes the class of languages L such that there is a $g \in \mathsf{Gap}\mathcal{C}$ satisfying

$$\forall x \in \{0,1\}^*, \; x \in L \iff g(x) = 0$$

Known results: In [Bar89], Barrington showed that $\mathsf{NC}^1 = \mathsf{BWBP} = \mathsf{BWC}$. As observed in [CMTV98], BWBP coincides with BP-NFA; thus $\mathsf{NC}^1 = \mathsf{BP\text{-}NFA}$. Later on, Istrail and Zivkovic showed in [IZ94] that $\mathsf{NC}^1 = \mathsf{LWF}$. In [Dym88], Dymond showed that acceptance by VPAs can be checked in NC^1, and hence $\mathsf{BP\text{-}VPA} = \mathsf{NC}^1$. Thus

Lemma 1 ([Bar89,CMTV98,IZ94,Dym88]).

$$\mathsf{NC}^1 = \mathsf{BWBP} = \mathsf{SC}^0 = \mathsf{LWF} = \mathsf{BP\text{-}NFA} = \mathsf{BP\text{-}VPA}$$

Though the above classes are all equal in the Boolean setting, in the arithmetic setting the equivalences are not established, and strict containments are also not known. The best known relationships among these classes are as below.

Lemma 2 ([CMTV98]).

$$\mathsf{FNC}^1 \subseteq \#\mathsf{BWBP} = \#\mathsf{BP\text{-}NFA} \subseteq \#\mathsf{NC}^1 \subseteq \mathsf{GapBWBP} = \mathsf{GapNC}^1 \subseteq \mathsf{L}$$

[1] In [CMTV98], this class is called BP. We introduce this new notation to better motivate the next definition, of BP-VPA.

The Chinese Remaindering Technique: We will frequently use the folklore Chinese Remainder Theorem CRT, and its algorithmic version ([CDL01], see also [All01]).

Lemma 3. *1. The jth smallest prime p_j is bounded by $O(j \log j)$; hence p_j can be computed and represented in space $O(\log j)$. Let $P_k = \prod_{i=1}^{k} p_k$; then P_k is at least 2^k.*
2. Every integer $a \in [0, P_k)$ is uniquely defined by the residues $a_i = a \bmod p_i$ where $i \in [k]$.
3. Given $a_1, \ldots, a_k$, where $a_i \in [0, p_i)$, the unique a such that $a \equiv a_i \bmod p_i$ for each $i \in [k]$ can be found in space $O(\log k)$.
4. Let $f : \{0,1\}^* \to \mathbb{N}$, with $f(x) \in [0, 2^{q(|x|)})$ where q is a polynomial. If the bits of $f(x) \bmod p_i$ can be computed in deterministic space s, for $i = 1, \ldots, q(|x|)$, then f can be computed in deterministic space $O(\log q(|x|) + s)$.

3 Counting accepting runs in visibly pushdown automata

Visibly pushdown automata (VPA) were defined by Alur and Madhusudan ([AM04]) as a restriction of pushdown automata where the stack movement is dictated by the input letter. Languages accepted by VPA are called visibly pushdown languages (VPL). In [AM04,AKMV05], many closure properties and decidability of many properties on VPL were proved. Previously, Mehlhorn ([Meh80]) had defined input driven languages (IDL). It is easy to see that IDL and VPL coincide. In [Meh80], it was shown that recognition problem for IDL is in DSPACE($\log n \log\log n$). This was improved to L in [BV83] and further to NC^1 in [Dym88]. The last result non-trivially used Buss' result for the Boolean formula value problem [Bus87]. Regular languages by definition are contained in VPL and the class REG is complete for NC^1. Thus, the NC^1 upper bound is tight for VPL under reasonable reductions.

We introduce a natural arithmetization of BP-VPA, by counting the number of accepting paths in a VPA rather than in an NFA. The definition mimics that of BP-NFA.

Given a uniform polynomial length branching program P over a VPA M, the number of distinct accepting paths of M on the projection of x is denoted by $\#\text{accept}(P, x)$.

Definition 2.

$$\#\text{BP-VPA} = \left\{ f : \{0,1\}^* \to \mathbb{N} \mid \begin{array}{l} f(x) = \#accept(P_{|x|}, x) \text{ for some uniform} \\ \text{polynomial length BP } P \text{ over a VPA } M \end{array} \right\}$$

The main result of this section is that functions in #BP-VPA can be evaluated in FLogDCFL. In an earlier version of this paper ([LMR07]), we had claimed that in fact #BP-VPA equals #BP-NFA. While this may well be true, the construction we gave there is erroneous, and we do not have an alternative proof.

A secondary result is that functions in #BP-VPA can be evaluated, modulo any fixed number, in FL.

Theorem 1. #BP-NFA$\subseteq$ #BP-VPA$\subseteq$ FLogDCFL.
For each fixed k, Mod_kBP-VPA $\subseteq$ L.

The first containment follows from definitions, since a VPA can simulate a NFA for *any* partition of the input. The rest of this section is devoted to proving the second and third containments. To see these, we consider the approach used by Braunmühl and Verbeek in [BV83]. They show that deciding membership in a fixed VPL is in L by first describing a $O(\log^2 n)$ space, $O(n \log n)$ time procedure and then modifying it to lower the space bound to $O(\log n)$ (at the cost of time going up to $O(n^2 \log n)$).

The essence of our proofs is captured in the following outline:

1. Algorithm 1 of [BV83] can be implemented in LogDCFL.
2. Arithmetizing Algorithm 1 to carry number of accepting paths rather than a Boolean bit indicating (non)-existence of accepting paths requires more than logarithmic auxiliary space. However, if the arithmetic values need $O(\log n)$ bits each, then the algorithm can still be implemented in LogDCFL. Now, using Chinese remaindering (Lemma 3), we can compute the actual function value in FLogDCFL.
3. Arithmetizing Algorithm 2 of [BV83] requires $O(\log^2 n)$ space, even if used in conjunction with Lemma 3. However, if the arithmetic values need $O(1)$ bits each, then the algorithm can be implemented in L. So for any fixed modulus k, counting the number of accepting paths modulo k can be done in L.

We now describe the individual steps. The algorithms of [BV83] assumed that VPAs accept only well-matched strings (strings such that every prefix of the string has at least as many push letters as pop letters, and the total number of push letters in the string equals the total number of pop letters). We first show that this is not a restriction.

Lemma 4. *For every* VPA M *over alphabet* Δ*, there is a corresponding* VPA M' *over an alphabet* Δ' *and a* TC^0 *many-one reduction* g *such that for every* $x \in \Delta^*$*,*

1. $\#acc_M(x) = \#acc_{M'}(g(x))$*, and*
2. $g(x)$ *is well-matched.*

Proof. Let $M = (Q, \Delta, Q_{in}, \Gamma, \delta, Q_F)$. The VPA $M' = (Q', \Delta', Q'_{in}, \Gamma', \delta', Q'_F)$ is essentially the same as M. It has two new input symbols A, B, and a new stack symbol X. A is a push symbol on which X is pushed, and B is a pop symbol on which X is expected and popped. M' has a new state q' that is the only initial state. M' expects an input from $A^*\Delta^*B^*$. On the prefix of A's it pushes X's. When it sees the first letter from Δ, it starts behaving like M. The only exception is when M performs a pop move on $\perp$, M' can perform the same move on $\perp$ or on X. On the trailing suffix of B's it pops X's. It is straightforward to design δ' from δ.

Let $|x| = n$. The TC^0 circuit does the following. It counts the difference d between the number of push and pop symbols in $A^n x$. It then outputs $y = A^n x B^d$. By the way M' is constructed, it should be clear that $\#acc_M(x) = \#acc_{M'}(y)$ and that M', on y, never pops on an empty stack. In fact y is well-matched. □

We now give an overview of the first algorithm from [BV83], stating it as a LogDCFL procedure. We use the characterization of LogDCFL as languages accepted by polynomial-time DAuxPDA.

Lemma 5 (Algorithm 1 of [BV83]). *Let $M = (Q, \Delta, Q_{in}, \Gamma, \delta, Q_F)$ be a* VPA *accepting well-matched strings. Given an input string x, checking if $x \in L(M)$ can be done in* LogDCFL.

Proof. Let $x_{ij} = x_{i+1}..x_j$ be a well-matched substring of the string x. (Define $x_{ii} = \epsilon$, the empty string.) Define a $(|Q||\Gamma| \times |Q||\Gamma|)$ matrix over $0, 1$, where each row and column is indexed by a state-stacktop pair (surface configuration). The entry indexed by $[(q, X), (q', X')]$ is 1 if and only if $X = X'$ and M goes from surface configuration (q, X) to (q', X') while processing the string x_{ij}. We will call such a matrix the table T_{ij} corresponding to the string x_{ij}. M has an accepting run on the string x if and only if the $[(q_0, \perp), (q, \perp)]$-th entry is 1 for some $q \in Q_F$ in the table corresponding to x_{0n}. Thus, it is sufficient to compute this table. However, in order to do so, we may have to compute many/all such tables.

We say that an interval $r = [i, j]$ is *valid* if $i \leq j$ and x_r, the string represented by the interval, is well-matched; otherwise it is said to be *invalid*. A *fragment* is a pair (r, Λ) where Λ is a pair (r', T'), r and r' are valid intervals, T' is a table. The fragments that arise in the algorithm satisfy the properties: (1) the interval r' is nested inside the interval r, and (2) T' is the table corresponding to the string $x_{r'}$, that is, $T' = T_{r'}$. For $r = (i, j)$, $\Lambda = (r', T')$ is *trivial* if $r' = [l, l]$ where $l = \lceil (i + 2j)/3 \rceil$ (this is the value of l used in [BV83] to obtain balanced cuts), $x_{r'} = \epsilon$, and T' is the identity table Id. The recursive procedure $\mathcal{T}$ takes a fragment (r, Λ) as an input and computes the table T_r, assuming that $T' = T'_{r'}$ where r' is a valid interval nested inside r. The main call made to the procedure is $([0, n], \Lambda)$ with trivial Λ.

The procedure $\mathcal{T}$ does the following: If the size of $r - r'$ is at most 2, then it computes the table T_r immediately from δ and T'. If the size of $r - r'$ is more than 2, then it breaks r into three valid intervals $r_1, r_2, r - (r_1 \cup r_2)$, where (1) the size of each of $r_1, r_2, r - (r_1 \cup r_2)$ is small (in two stages, each subinterval generated will be at most three-fourths the size of $r - r'$), (2) one of r_1, r_2 completely contains r', (3) r_1, r_2 are contiguous with r_1 preceding r_2. It then creates fragments (r_1, Λ_1) and (r_2, Λ_2) where $\Lambda_1 = \Lambda$ and Λ_2 is trivial if r_1 contains r', and $\Lambda_2 = \Lambda$ and Λ_1 is trivial if r_2 contains r'. Now it evaluates these fragments recursively to obtain the tables $\mathcal{T}(r_1, \Lambda_1) = T_{r_1}$, $\mathcal{T}(r_2, \Lambda_2) = T_{r_2}$, and obtains the table $T_{r_3} = T_{r_1} \times T_{r_2}$, where $r_3 = r_1 \cup r_2$ and the $\times$ represents Boolean matrix product. Setting $\Lambda_3 = (r_3, T_{r_3})$, it finally makes the recursive call $\mathcal{T}(r, \Lambda_3)$ to compute T_r. In [BV83], it is shown that such fragments can always be defined and can be found deterministically and uniquely. (We will discuss the complexity of finding such fragments shortly.) It is also shown that the tables computed by above recursion procedure have the following property: for the table T corresponding to the interval $r = [i, j]$, the $[(q, X), (q', X')]$-th entry is 1 exactly when the machine has at least 1 path from (q, X) to (q', X') on string x_{ij}. This proof is by induction on the length of the intervals.

Note that the above procedure yields a $O(\log n)$ depth recursion tree, with each internal node having three children corresponding to the three recursive calls made. The leaves of this recursion tree are disjoint effective intervals (for fragment $(r, (r', T'))$, the effective interval is $r - r'$). As the main call is made to the fragment $([0, n], \Lambda)$ with trivial Λ, the size of such a tree will be $O(n)$. Also note that the depth-first traversal of the recursion tree generated by the above procedure can be performed in LogDCFL. This is because the deterministic

AuxPDA will stack one fragment (say (r_1, Λ_1)) and process the other fragment (r_2, Λ_2) on the logspace work-tape. Once it finishes processing both these fragments, it will then have Λ_3 on its work-tape and hence can start processing (r, Λ_3). This amounts to a depth-first traversal of the recursion tree. As the size of the tree is of $O(n)$, the DAuxPDA will run in time $p(n)$ for some polynomial p, provided the selection of the subintervals r_1, r_2 from (r, Λ) can be done on a logspace work-tape.

Now we describe a deterministic log space procedure to compute intervals r_1 and r_2. Let $r = [i, j]$ and $r' = [i', j']$ be such that $i \leq i' \leq j' \leq j$ and $(r - r') > 2$. Consider the larger of the two subintervals $[i, i']$ and $[j', j]$. Break it into two equal size parts. Consider the part closer to r'. In this, find an index t such that the height of the stack of M just after reading x_t (denoted as $h(t)$) is the lowest in that part. Now find two more points b, a such that $h(b) = h(t) = h(a)$, and the interval $[b, a]$ is the maximal valid subinterval containing t and within $[i, j]$. Let $r_1 = [b, t]$, $r_2 = [t, a]$. Observe that r_1 and r_2 are contiguous with r_1 preceding r_2. Also, r' is fully contained inside either r_1 or r_2. (Why? Consider the case when $t \leq i' \leq j'$. t was the lowest in the part preceding r', thus $h(t) \leq h(i')$. If $a < j'$, then by maximimality of $[a, b]$, $h(a + 1) < h(a) = h(t) \leq h(i')$, and $a + 1 \in [t + 1, j']$. By choice of t, $a + 1 > i'$. Thus $a + 1 \in r'$, and $h(a + 1) < h(i') = h(j')$, contradicting the fact that r' is a valid interval. Hence it must be the case that $t \leq i' \leq j' \leq a$, and so r_2 contains r'. The case when $i' \leq j' \leq t$ is similar.)

It is easy to see that $r - r_3$ is of size at most three-fourths the size of $r - r'$, and so is the interval that does not contain r' (either r_1 or r_2). The same may not be true of the third part; the subinterval which contains r', say r_b, can be as large as $r - r'$. But it is easy to observe that at next step, this part will get tri-partitioned into intervals with sizes at most two-thirds the size of r_b each.

Once r_1, r_2 are fixed, the three fragments can be found as described above. Thus, finding the three fragments essentially boils down to finding b, t, a. This can be done in TC^0 for the input string x over pushdown alphabet Δ, and hence in L. □

Adaptation to arithmetic setting: To adapt this algorithm to the arithmetic setting, we now consider the tables that we defined in the proof of Lemma 5, and lift them over to $\mathbb{N}$. Consider the fragment (r, Λ), where $\Lambda = (r', T')$. The interval r' is a sub-interval of the interval r. Let $u, v \in \Delta^*$ be such that $x_r = u x_{r'} v$ (note that one of u, v, $x_{r'}$ can possibly be ϵ). The modifications to procedure $\mathcal{T}$ are as follows.

If $|x_r| \leq 2$ (in this case Λ will be trivial), then the table T_r can be filled as follows: the $[(q, X), (q', X')]$-th entry will be set to k if $X = X'$ and the machine M can start from surface configuration (q, X), read x_r, and reach configuration (q', X') in *exactly* k ways. This can be filled by simply looking up the transition function δ'. Thus at the base case, we can fill the tables.

If $|x_r| > 2$ but $|uv| \leq 2$ (so Λ is not trivial), then assume that inductively, we have the table $T'_{r'}$ computed correctly. Then set the $[(q, X), (q', X)]$-th entry of the table T_r as follows:

If u and v are well-matched, then

$$T_r^{[(q,X),(q',X)]} = \sum_{q_1,q_2 \in Q} \#[(q,X) \rightsquigarrow_u (q_1,X)] \; . \; T_{r'}^{[(q_1,X),(q_2,X)]} \; . \; \#[(q_2,X) \rightsquigarrow_v (q',X)] \qquad (1)$$

Here, the notation $\#[\alpha \rightsquigarrow_w \beta]$ denotes the number of ways of going from surface configuration α to surface configuration β while reading input w.

Otherwise it must be that u is a push letter and v is a pop letter. In this case,

$$T_r^{[(q,X),(q',X)]} = \sum_{q_1,q_2 \in Q, Y \in \Gamma} \#[(q,X) \rightsquigarrow_u (q_1,Y)] \; . \; T_{r'}^{[(q_1,Y),(q_2,Y)]} \; . \; \#[(q_2,Y) \rightsquigarrow_v (q',X)] \qquad (2)$$

Both these cases can be combined into the following single equation:

$$\begin{aligned} T_r^{[(q,X),(q',X)]} &= \sum_{s_1,s_2 \in Q \times \Gamma} \#[(q,X) \rightsquigarrow_u s_1] \; . \; T_{r'}^{[s_1,s_2,]} \; . \; \#[s_2 \rightsquigarrow_v (q',X)] \\ T_r^{[(q,X),(q',X')]} &= 0 \qquad \text{if } X \neq X' \end{aligned} \qquad (3)$$

Since $T'_{r'}$ is available through recursive computation, and since the other terms in these equations can be found from δ, $\mathcal{T}$ can compute T_r.

For handling the case when $|uv| > 2$, we just redefine the $\times$-operator as matrix multiplications over $\mathbb{N}$. Let T_b denote the table corresponding to interval r_b, for $b = 1, 2$, where r_1 and r_2 are contiguous with r_1 preceding r_2. Then the table T_3 for $r_3 = r_1 \cup r_2$ is given by $T_3 = T_1 \times T_2$; that is,

$$T_3^{[(q,X),(q',X')]} = \sum_{p,Y \in Q \times \Gamma} T_1^{[(q,X),(p,Y)]} \; . \; T_2^{[(p,Y),(q',X')]} \qquad (4)$$

(Inductively, this sets an entry to be 0 if $X \neq X'$.)

Under this semantics for tables and $\times$ operator, we can establish the following:

Claim. For every interval $r = [i, j]$ arising in the recursion tree on input $([0, n], ([2n/3, 2n/3], \mathsf{Id}))$, the $[(q, X), (q', X')]$-th entry of the table T_r computed by $\mathcal{T}$ equals the number of distinct paths of M from (q, X) to (q', X) on string x_{ij}.

Proof. (of claim) The procedure $\mathcal{T}$ processes intervals as fragments. The correctness proof proceeds by induction on the effective size of the interval; that is, for a recursive call on input $(r, (r', T'))$, we show by induction on the size of $r - r'$ that if T' is correct for r', then $\mathcal{T}$ returns the correct table for r.

The base case is when $r - r'$ is an interval of size 2 or less. If $|r'| = 0$, then $\mathcal{T}$ computes T_r directly from δ and so is correct. If $r' \neq 0$, then correctness follows from Equation 3.

For the inductive case, consider a fragment where $|r - r'| > 2$. $\mathcal{T}$ computes fragments (r_1, Λ_1) and (r_2, λ_2) and makes recursive calls. Assume that $\{b, c\} = \{1, 2\}$ and that r_c contains r'. As argued in [BV83], the effective interval in fragment (r_c, Λ_c) is strictly smaller than $|r - r'|$, and so by induction, $\mathcal{T}$ correctly computes T_{r_c}. r_b has a trivial pair attached and may not be smaller. Assume for now that it is smaller, so by induction, $\mathcal{T}$ correctly computes

T_{r_b} as well. Now T_{r_3} is computed by Equation 4 which correctly combines paths over x_{r_1} and x_{r_2}. Finally, $\mathcal{T}$ is invoked with $(r, (r_3, T_3))$, and by induction, this call terminates with the correct value of T_r.

Suppose now that r_b is not smaller than $r - r'$. (This can happen, for instance, if r_c contains just r' and r_b is all the rest of r.) But then $\mathcal{T}$, while processing (r_b, Λ_b), makes calls with inputs (r_{bl}, Λ_{bl}) where $l \in \{1, 2\}$, and each call has a smaller effective interval length. So $T_{r_{b1}}$ and $T_{r_{b2}}$ are computed correctly by induction, and $T'' = T_{r_{b1} \cup r_{b2}}$ is obtained via Equation 4 which correctly combines paths. Then $\mathcal{T}$ is invoked with $(r_b, (r_{b1} \cup r_{b2}, T''))$. By induction, this call terminates with the correct value of T_{r_2}. □

The modified recursive procedure for computing the newly defined tables over $\mathbb{N}$ cannot directly be implemented in LogDCFL, because each entry of a table may need polynomially many bits. (The number of paths of a VPA on any string cannot exceed $2^{O(n)}$, so polynomially many bits suffice.) However, if we were to perform all the operations modulo small primes, each needing logarithmically many bits, then analogous to Lemma 5, the modified procedure can be implemented[2] in FLogDCFL. The fragments that get pushed on to the stack and processed on the work tape will have $O(\log n)$-bit representations owing to not only the indices of the intervals but also the tables. (In the previous case, the tables were of size $O(1)$.) This can be handled by an AuxPDA. If we do the above implementation for sufficiently many primes, then by Chinese Remaindering (Lemma 3) we will be able to recover the exact number in FL. Overall, we have that counting number of accepting paths in machine M over input x can be performed in $\mathsf{FL}^{\mathsf{FLogDCFL}} = \mathsf{FLogDCFL}$. In conjunction with Lemma 4, this shows the second containment of Theorem 1, namely #BP-VPA $\subseteq$ FLogDCFL.

Counting modulo k, for fixed k: Now we come to the third containment of our theorem, namely, Mod_kBP-VPA $\subseteq$ L for each fixed k. We appeal to the second algorithm of [BV83], that yields the following.

Lemma 6 (Algorithm 2 of [BV83]). *Let M be a* VPA *accepting well-matched strings over an alphabet Δ. Given an input string x, checking if $x \in L(M)$ can be done in* L.

Instead of describing this algorithm and the modification required for our result, we directly describe the modified version.

It suffices to compute the table operations, as defined in Equations 1,2, and 4, modulo k. Hence, the tables will be of size $O(|Q|^2|\Gamma|^2 k) = O(1)$. Thus the table size does not cause an increased space requirement.

Further, note that all the fragments need not be carried along explicitly. Just remembering the path in the recursion tree, and the tables for all nodes on the path, suffices. Say the recursion tree is labelled as follows: The three children of a node are called l, r, o to mean 'left', 'right' and 'other', for the recursive calls $\mathcal{T}(r_1, \Lambda_1)$, $\mathcal{T}(r_2, \Lambda_2)$, and $\mathcal{T}(r, \Lambda_3)$ respectively. Label a node by a string $w \in \{l, r, o\}^*$ to denote the position of the node in the tree. (e.g label the leftmost leaf by l^d where d is depth of that leaf, label the root of the tree by ϵ.) It

[2] Note that if a prime p is bounded in value by $O(q(n))$, then for $a, b \in [0, p)$, the brute force algorithm for computing $(a \times b) \mod p$ and $(a + b) \mod p$ can be implemented in deterministic space $O(\log q(n))$.

is easy to see (and this was used in [BV83] to prove correctness of their algorithm) that if one knows the label for a node, then reconstructing the intervals r, r' at this node from this label is possible in L. They also observed that computing the next label from the current node label can be done in L (in fact, it is easy to note that this can be done in TC^0.) Thus at any stage our algorithm needs to remember the label of the current node being processed, and appropriate tables. We already saw that the table size is $O(1)$. Our procedure needs to know at most one table (the table in the Λ part of the fragment) per node along the current path. As the depth of the recursion is bounded by $O(\log n)$, the depth of any node is also bounded $O(\log n)$. Thus, the label size, and the number of tables that need to be stored, are both at most $O(\log n)$ for any node. Hence the procedure does not need to remember more than $O(\log n)$ bits at any stage of the recursion.

4 Counting proof trees in log width formulas

We show that the result of [IZ94], asserting that log width formulas capture NC^1, holds in the arithmetized setting as well. This result is crucial to obtain a connection between $\#\mathsf{NC}^1$and $\#\mathsf{sSC}^1$(refer Theorem 4).

Definition 3.

$$\#\mathsf{F} = \left\{ f : \{0,1\}^* \to \mathbb{N} \;\middle|\; \begin{array}{l} f \text{ can be computed by a polynomial size formula} \\ \text{over } \{+, \times, 1, 0, x_i, \overline{x_i}\}. \end{array} \right\}$$

$$\#\mathsf{LWF} = \left\{ f : \{0,1\}^* \to \mathbb{N} \;\middle|\; \begin{array}{l} f \text{ can be computed by a polynomial size } O(\log n) \\ \text{width formula over } \{+, \times, 1, 0, x_i, \overline{x_i}\}. \end{array} \right\}$$

Theorem 2. $\#\mathsf{LWF} = \#\mathsf{F} = \#\mathsf{NC}^1$

Proof. Clearly, $\#\mathsf{LWF} \subseteq \#\mathsf{F}$. It follows from [BCGR92] (see also [All04]) that $\#\mathsf{F}$ is in $\#\mathsf{NC}^1$. So we only need to show that $\#\mathsf{NC}^1$ is in $\#\mathsf{LWF}$.

Lemma 2 in [IZ94] establishes that $\mathsf{NC}^1 \subseteq \mathsf{LWF}$. Essentially, it starts with a $O(\log n)$ depth formula (any NC^1 circuit can be expanded into a formula of polynomial size and $O(\log n)$ depth), and staggers the computations at each level of the formula. In the process, the size blows up by a factor exponential in the original depth; this is still a polynomial. We observe that since no other structural changes are done, the reduction preserves proof trees. □

5 Polynomial degree small-width circuits and their arithmetization

We now consider arithmetization of SC^i circuits.

A straightforward arithmetization of any Boolean circuit class over $(\wedge, \vee, x_i, \overline{x_i}, 0, 1)$ is to replace each $\vee$ gate by a $+$ gate and each $\wedge$ gate by a $\times$ gate. In the case of SC^0 (SC^i in general), this enables the circuit to compute infeasible values (i.e exponential sized values),

which makes the class uninteresting. Hence we propose bounded degree versions of these classes and then arithmetize them.

The degree of a circuit is the maximum degree of any gate in it, where the degree of a leaf is 1, the degree of an $\vee$ or $+$ gate is the maximum of the degrees of its children, and the degree of a $\wedge$ or $\times$ gate is the sum of the degrees of its children.

Definition 4. sSC^i *is the class of languages accepted by Boolean circuits of polynomial size,* $O(\log^i n)$ *width and polynomial degree.*

$\#\mathsf{sSC}^i$ *is the class of functions computed by arithmetic circuits of polynomial size,* $O(\log^i n)$ *width and polynomial degree. Equivalently, it is the class of functions counting the number of proof trees in an* sSC^i *circuit.*

$$\mathsf{sSC} = \bigcup_{i \geq 0} \mathsf{sSC}^i \qquad\qquad \#\mathsf{sSC} = \bigcup_{i \geq 0} \#\mathsf{sSC}^i$$

Note that SC circuits can have internal NOT gates as well; moving the negations to the leaves only doubles the width. However, when we restrict degree as in sSC, we explicitly disallow internal negations. The circuits have only AND and OR gates, and constants and literals appear at leaves.

It is known that polynomial-size circuits of polynomial degree, irrespective of width or depth, characterize LogCFL, which is equivalent to semi-unbounded log depth circuits SAC^1, and hence is contained in NC^2 [Sud78,Ruz80,Ven91]. This equivalence also holds in the arithmetic settings for # and for Gap, see [Vin91,NR95,AJMV98]. Thus

Proposition 1. *For all* $i \geq 0$*,*

1. $\mathsf{sSC}^i \subseteq \mathsf{LogCFL}$
2. $\#\mathsf{sSC}^i \subseteq \#\mathsf{LogCFL}$
3. $\mathsf{GapsSC}^i \subseteq \mathsf{GapLogCFL}$

Any branching program can be viewed as a skew circuit. A skew circuit's degree is bounded by its size. Thus BWBP is contained in sSC^0. But $\mathsf{SC}^0 = \mathsf{BWBP} = \mathsf{NC}^1$. Thus

Proposition 2. $\mathsf{sSC}^0 = \mathsf{SC}^0 = \mathsf{NC}^1$.

We do not know whether such an equality ($\mathsf{sSC}^i = \mathsf{SC}^i$) holds at any other level. If it holds for any $i \geq 2$, it would bring a larger chunk of SC into the NC hierarchy.

We now show that the individual bits of each $\#\mathsf{sSC}^i$ function can be computed in polynomial time using $O(\log^{i+1} n)$ space. However, the Boolean circuits constructed may not have polynomial degree.

Theorem 3. *For all* $i \geq 0$, $\#\mathsf{sSC}^i \subseteq \mathsf{GapsSC}^i \subseteq \mathsf{SC}^{i+1}$

Proof. We show how to compute $\#\mathsf{sSC}^i$ in SC^{i+1}. The result for Diff and hence Gap follows since subtraction can be performed in SC^0.

Let $f \in \#\mathsf{sSC}^i$. Let d be the degree bound for f. Then the value of f can be represented by at most $d \in n^{O(1)}$ many bits. By the Chinese Remainder Theorem, f can be computed exactly

from its residues modulo the first $O(d^{O(1)})$ primes, each of which has $O(\log d) = O(\log n)$ bits. These primes are small enough that they can be found in logspace. Further, due to [CDL01], the computation of f from its residues can also be performed in $\mathsf{L}= \mathsf{SC}^1$; see also [All01]. If the residues can be computed in SC^k, then the overall computation will also be in SC^k because we can think of composing the computations in a sequential machine with a simultaneous time-space bound.

It thus remains to compute $f \bmod p$ where p is a small prime. Consider a bottom-up evaluation of the $\#\mathsf{sSC}^i$ circuit, where we keep track of the values of all intermediate nodes modulo p. The space needed is $\log p$ times the width of the circuit, that is, $O(\log^{i+1} n)$ space, while the time is clearly polynomial. Thus we have an SC^{i+1} computation. □

In particular, bits of an $\#\mathsf{sSC}^0$ function can be computed in SC^1, which equals L. On the other hand, by an argument similar to the discussion preceding Proposition 2, we know that $\#\mathsf{BWBP}$ is contained in $\#\mathsf{sSC}^0$. Thus

Corollary 1. $\mathsf{FNC}^1 \subseteq \#\mathsf{BWBP} \subseteq \#\mathsf{sSC}^0 \subseteq \mathsf{FL}$.
$\mathsf{GapNC}^1 = \mathsf{GapBWBP} \subseteq \mathsf{GapsSC}^0 \subseteq \mathsf{FL}$.

We cannot establish any direct connection between $\#\mathsf{sSC}^0$ and $\#\mathsf{NC}^1$. Thus this is potentially a fourth arithmetization of the Boolean class NC^1, the other three being $\#\mathsf{BWBP}$, $\#\mathsf{NC}^1$, and $\#\mathsf{BP\text{-}VPA}$.

We also do not know whether sSC^1 properly restricts SC^1=L. Even if it does, it cannot fall below NC^1, since $\mathsf{NC}^1 = \mathsf{sSC}^0$(Proposition 2). We note that this holds in the arithmetic setting as well:

Theorem 4. $\#\mathsf{NC}^1 \subseteq \#\mathsf{sSC}^1$.

Proof. From Theorem 2, we know that $\#\mathsf{NC}^1$ equals $\#\mathsf{LWF}$. But an LWF has log width and has polynomial degree since it is a formula; hence $\#\mathsf{LWF}$ is in $\#\mathsf{sSC}^1$. □

Since the levels of sSC are sandwiched between NC^1 and LogCFL, both of which are closed under complementation, it is natural to ask whether the levels of sSC are also closed under complement. While we are as yet unable to show this, we show that for each i, co-sSC^i is contained in sSC^{2i}; thus sSC as a whole is closed under complement.

Theorem 5. *For each $i \geq 1$, co-sSC^i is contained in sSC^{2i}.*

Proof. Consider the proof of closure under complement for LogCFL, from [BCD$^+$89]. This is shown by considering the characterization of LogCFL as semi-unbounded log depth circuits, and applying an inductive counting technique to such circuits. Our approach for complementing sSC^i is similar: use inductive counting as applied by [BCD$^+$89]. However, one problem is that the construction of [BCD$^+$89] uses monotone NC^1 circuits for threshold internally, and if we use these directly, the degree may blow up. So for the thresholds, we use the construction from [Vin96]. A careful analysis of the parameters then yields the result.

Let C_n be a Boolean circuit of length l, width $w = O(\log^i n)$ and degree p. Without loss of generality, assume that C_n has only $\vee$ gates at odd levels and $\wedge$ gates at even levels.

Also assume that all gates have fan in 2 or less. We construct a Boolean circuit C'_n, which computes $\bar{C}_n$. C'_n contains a copy of C_n. Besides, for each level k of C_n, C'_n contains the gates $cc(g|c)$ where g is a gate at level k of C_n and $0 \leq c \leq w$. These represent the conditional complement of g assuming the count at the previous level is c, and are defined as follows:

$$cc(g|c) = \begin{cases} cc(a_1|c) \vee cc(a_2|c), & \text{if } \ g = a_1 \wedge a_2 \\ Th^c(b_1, \cdots, b_j), & \text{if } \ g = a_1 \vee a_2 \end{cases}$$

where $b_1, \cdots, b_j$ range over all gates at the previous level except a_1 and a_2.

C'_n also contains, for each level k of C_n and $0 \leq c \leq w$, the gates $count(c, k)$. These gates verify that the count at level k is c, and are defined as follows:

$$count(c,k) = \begin{cases} Th1(c,k) \wedge \bigvee_{d=0}^{w} [count(d, k-1) \wedge Th0(c,k,d)] & \text{if } k > 0 \\ 1 \quad \text{if } k = 0, c = \#\text{ of inputs with value 1 at level 0} \\ 0 \quad \text{otherwise} \end{cases}$$

Th^c is the c-threshold value of its inputs , $Th1(c,k) = Th^c$ of all original gates (gates from C_n) at level k, $Th0(c,k,d)$ is Th^{Z-c} of all $cc(g|d)$ at level k where Z is the number of gates in C_n at level k. Finally, the output gate of C'_n is $comp(g) = \bigvee_{c=0}^{w} Count(c, l-1) \wedge cc(g|c)$, where g is the output gate of C_n, at level l. Correctness follows from the analysis in [BCD$^+$89].

A crucial observation, used also in [BCD$^+$89], is that any root-to-leaf path goes through at most two threshold blocks.

To achieve small width and small degree, we have to be careful about how we implement the thresholds. Since the inputs to the threshold blocks are computed in the circuit, we need monotone constructions. We do not know whether monotone NC^1 is in monotone sSC^0, for instance. But for our purpose, the following is sufficient: Lemma 4.3 of [Vin96] says that any threshold on K bits can be computed by a monotone branching program (hence small degree) of width $O(K)$ and size $O(K^2)$. This branching program has degree $O(K)$. Note that the thresholds we use have $K = O(w)$. The threshold blocks can be staggered so that the $O(w)$ extra width appears as an additive rather than multiplicative factor. Hence the width of C'_n is $O(w^2)$. (The conditional complement gates cause the increase in width; there are $O(w^2)$ of them at each level.)

Let q be the degree of a threshold block; $q = O(K) = O(w)$. If the inputs to a threshold block come from computations of degree p, then the overall degree is pq. Since a $cc(g|c)$ gate is a threshold block applied to gates of C_n at the previous level, and since these gates all have degree at most p, the $cc(g|c)$ gate has degree at most pq.

Also, the degree of a $count(c,k)$ gate is bounded by the sum of (1) the degree of a $count(c, k-1)$ gate, (2) the degree of a threshold block applied to gates of C_n, and (3) the degree of a threshold block applied to $cc(g|c)$ gates. Hence it is bounded by $p^{O(1)} w^{O(1)} l$, where l is the depth of C_n. Thus, the entire circuit has polynomial degree. □

6 Extensions and Closure Properties

In this section, we show that some closure properties that hold for $\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$ also hold for $\#\mathsf{sSC}^0$. The simplest closures are under addition and multiplication, and it is straightforward to see that $\#\mathsf{sSC}^0$ is closed under these. The next are weak sum and weak product: add (or multiply) the value of a two-argument function over a polynomially large range of values for the second argument. (See [CMTV98,Vol99] for formal definitions.) A simple staggering of computations yields:

Lemma 7. *For each $i \geq 0$, $\#\mathsf{sSC}^i$ is closed under weak sum and weak product.*

$\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$ are known to be closed under decrement $f \ominus 1 = \max\{f-1, 0\}$ and under division by a constant $\lfloor \frac{f}{m} \rfloor$. ([AAD00] credits Barrington with this observation for $\#\mathsf{NC}^1$. See the appendix for detailed constructions.) We show that these closures hold for $\#\mathsf{sSC}^0$ as well. The following property will be useful.

Proposition 3. *For any f in $\#\mathsf{sSC}^0$ or $\#\mathsf{SC}^0$ or $\#\mathsf{NC}^1$, and for any constant m, the value $f \mod m$ is computable in FNC^1. Further, the boolean predicates $[f > 0]$ and $[f = 0]$ are computable in $\#\mathsf{sSC}^0$.*

Proof. Consider $f \in \#\mathsf{NC}^1$. Note that, for a constant m, if $a, b \in \{0, \ldots, m-1\}$, then the values $[(a+b) \mod m]$ and $[(ab) \mod m]$ can be computed by an NC^0 circuit. Thus by induction on depth of f, $[f \mod m]$ can be computed in FNC^1. Now consider $f \in \#\mathsf{sSC}^0$. We will argue by induction on the depth of a circuit for f, that $[f \mod m] \in \mathsf{sSC}^0$. The base case is obvious. If $f = g + h$, then by the induction hypothesis, $g \mod m$, $h \mod m \in \mathsf{sSC}^0 = \mathsf{NC}^1$. Thus, $(g \mod m + h \mod m) \mod m \in \mathsf{NC}^1 = \mathsf{sSC}^0$. The case when $f = gh$ is similar. Thus $f \mod m \in \mathsf{FNC}^1$.

Clearly $[f > 0] \in \mathsf{NC}^1$. Since NC^1 is closed under complementation, $[f = 0] \in \mathsf{NC}^1$. Since NC^1 circuits have deterministic branching programs of constant width, and branching programs are nothing but skew circuits, we obtain constant width arithmetic circuits for $[f > 0]$ and $[f = 0]$. □

Lemma 8. *$\#\mathsf{sSC}^0$ is closed under decrement and under division by a constant m.*

Proof. Consider $f \in \#\mathsf{sSC}^0$, witnessed by an arithmetic circuit C_n of width w, length l and degree p. Also for a fixed m, $(f \mod m)$ can be computed in FNC^1 (see Proposition 3). If g, h are in $\#\mathsf{sSC}^0$, then the functions t_1, t_2 defined below can be computed in FNC^1 and $\#\mathsf{sSC}^0$.

$$t_1 = \left\lfloor \frac{g \mod m + h \mod m}{m} \right\rfloor \qquad t_2 = \left\lfloor \frac{(g \mod m)(h \mod m)}{m} \right\rfloor$$

f at level l is either $g + h$ or gh. Let $op \in \{\ominus 1, \mathrm{div}\, m\}$. The circuit for $op(f)$ takes values of g and h from level $(l-1)$ of C_n, and values of $op(g)$ and $op(h)$ that are inductively available at level $(l-1)$. Appropriate circuits ($\#\mathsf{sSC}^0$ circuits computing the predicates $[f > 0]$ and $[f = 0]$, or $\#\mathsf{sSC}^0$ circuits computing $(g \mod m)$, $(h \mod m)$, t_1, t_2) for each

gate at level $l-1$ are explicitly substituted, contributing a multiplicative factor for width $O(w)$ and length $O(l)$ to the constructed circuit.

When $op = \ominus$, we have

$$(g+h) \ominus 1 = (g \ominus 1 + h) \times [g > 0] + (h \ominus 1) \times [g = 0]$$

$$gh \ominus 1 = [(g \ominus 1) \times h + h \ominus 1] \times [g > 0] \times [h > 0]$$

When $op = \text{div } m$, we have

$$\left\lfloor \tfrac{g+h}{m} \right\rfloor = \left\lfloor \tfrac{g}{m} \right\rfloor + \left\lfloor \tfrac{h}{m} \right\rfloor + t_1$$

$$\left\lfloor \tfrac{gh}{m} \right\rfloor = \left\lfloor \tfrac{g}{m} \right\rfloor \times h + \left\lfloor \tfrac{h}{m} \right\rfloor \times (g \mod m) + t_2$$

The constructed arithmetic circuit for $op(f)$ has width $O(w^2)$ and length $O(l^2)$. Let $p = deg(C_n)$, $q_1 = \max\{deg([f > 0]), deg([f = 0])\}$, and $q_2 = \max\{deg(g \mod m), deg(h \mod m), deg(t_1), deg(t_2)\}$. Then the circuit for $f \ominus 1$ has degree at most $p + lq_1$, while that for $\lfloor \frac{f}{m} \rfloor$ has degree at most $p + lq_2$.

Thus we have $op(f) \in \#\mathsf{sSC}^0$. □

Another consequence of Proposition 3 can be seen as follows. We have three competing arithmetizations of the Boolean class NC^1: $\#\mathsf{BWBP}$, $\#\mathsf{NC}^1$ and $\#\mathsf{sSC}^0$. (Until we can show that $\#\mathsf{BP\text{-}VPA}$ is at least in FL, $\#\mathsf{BP\text{-}VPA}$ cannot really be considered a natural arithmetization of NC^1.) The most natural one is $\#\mathsf{NC}^1$, defined by arithmetic circuits. It contains $\#\mathsf{BWBP}$, which is contained in $\#\mathsf{sSC}^0$, though we do not know the relationship between $\#\mathsf{NC}^1$ and $\#\mathsf{sSC}^0$. Applying a "> 0?" test to any yields the same class, Boolean NC^1. We show here that applying a "$\equiv 0 \bmod p$?" test to any of these arithmetic classes also yields the same language class, namely NC^1.

Theorem 6. *For any fixed p, $\mathsf{Mod}_p\mathsf{BWBP} = \mathsf{Mod}_p\mathsf{sSC}^0 = \mathsf{Mod}_p\mathsf{NC}^1 = \mathsf{NC}^1$.*

Proof. The NC^1-hardness for each of these three problems is obvious. From Proposition 3, for $f \in \{\#\mathsf{sSC}^0, \#\mathsf{BWBP}, \#\mathsf{NC}^1\}$, and a constant m, the value $[f(x) \bmod m]$ can be computed in FNC^1. Hence the predicate $[f(x) \equiv 0 \bmod m]$ can be computed in NC^1. □

There is another natural way to produce Boolean circuits from arithmetic circuits, by allowing the circuit to perform a "test for nonzero" operation. Such circuits, known as *Arithmetic-Boolean* circuits, were introduced by von zur Gathen, and have been studied extensively in the literature see *e.g.* [vzGS91,vzG93,BCGR92,All04]. We extend this a little further, by looking at bounded width restrictions.

Definition 5. *Let $\mathcal{C}$ be any of the arithmetic circuit class studied above, then Arith-Bool $\mathcal{C}$, is defined to be the set of languages, which are accepted by circuits, with the following additional gates,*

$$test(f) = \begin{cases} 0 & \text{if } f = 0 \\ 1 & \text{otherwise} \end{cases} \qquad select(f_0, f_1, y) = \begin{cases} f_0 & \text{if } y = 0 \\ f_1 & \text{if } y = 1 \end{cases}$$

where y is either a constant or a literal.

Assigning $deg(select(f_0, f_1, y)) = 1 + \max\{deg(f_0), deg(f_1)\}$ and $deg(test(f)) = deg(f)$, we have the following,

Lemma 9. *1. Arith-Bool*$\#\mathsf{NC}^1 = \#\mathsf{NC}^1$*.[All04]*
2. Arith-Bool$\#\mathsf{BWBP} = \#\mathsf{BWBP}$*.*
3. Arith-Bool$\#\mathsf{sSC}^0 = \#\mathsf{sSC}^0$

Proof. 1 and 2 are straight forward. If $f \in \#\mathsf{sSC}^0$then the predicate $[f > 0]$ can be computed by an unambiguous skew-sSC^0circuit. Now, given any Arith-Bool$\#\mathsf{sSC}^0$circuit C of length l, starting from the bottom, replace every $test(f)$ gate by the sSC^0 circuit which computes $[f > 0]$, and each $select(f_0, f_1, y)$ by the circuit $\bar{y}f_0 + y.f_1$. We also stagger the resulting circuit C', so that it has width $5w$, and length ll', where l' is an upper bound on the length of the circuit for $[f > 0]$. It can also be seen that $deg(C') \leq deg(f).q$, where q is a polynomial upper bound on the degree of $[f > 0]$. □

However, for the Gap classes, we do not have such a collapse. Analogous to the definitions of SPP and SPL, define a class SNC^1: it consists of those languages L for which there is a GapNC^1 function f satisfying

$$\forall x: \quad \begin{array}{l} x \in L \iff f(x) = 1 \\ x \notin L \iff f(x) = 0 \end{array}$$

Then we have the following conditional result.

Lemma 10. *Arith-Bool*$\mathsf{GapNC}^1 = \mathsf{GapNC}^1$ *if and only if* $\mathsf{SNC}^1 = \mathsf{C}_=\mathsf{NC}^1$.

Proof. If Arith-Bool$\mathsf{GapNC}^1 = \mathsf{GapNC}^1$, then the characteristic functions of languages in $\mathsf{C}_=\mathsf{NC}^1$ can be computed in GapNC^1. (Put a single test operation above the circuit for the GapNC^1 function.) This implies that $\mathsf{C}_=\mathsf{NC}^1$ is in SNC^1. Conversely, if $\mathsf{SNC}^1 = \mathsf{C}_=\mathsf{NC}^1$, then any test operation can be performed in GapNC^1. Select can be implemented using test and arithmetic operations anyway. □

7 Discussion

We have studied arithmetizations of some classes that are equivalent to NC^1 in the Boolean setting. Some interesting questions arise from our study.

1. Augmenting an NFA with a pushdown stack pushes up the class to LogCFL, which is powerful enough to contain all of NL. If the pushdown stack is restricted to be visible, then the Boolean complexity remains the same (NC^1), but the arithmetic complexity is not yet clear. We have shown that it is upper bounded by $\mathsf{FLogDCFL}$. We believe, however, that it is in fact within FL, and possibly coincides with either $\#\mathsf{NC}^1$ or $\#\mathsf{BWBP}$; a proof would probably need to carefully combine techniques from [Dym88] and [Bus87]. Settling this is an interesting question. In [LMM08], some further extensions beyond VPA are shown to coincide with VPA in both Boolean and arithmetic settings, while a significant generalization still lies within $\mathsf{LogDCFL}$.

2. A related question is: Is there some restriction on stack usage that will allow us to characterize L in the Boolean setting, and $\#\mathsf{NC}^1$ or FL in the arithmetic setting? Deterministic one-turn PDA do characterize L ([HL93], see also [Lan93,Mah07]), but for arithmetizing them we need a nondeterministic equivalent.
3. In the Boolean setting, exactly how much does the polynomial degree constraint restrict SC^1? In $\mathsf{NC}^1 \subseteq \mathsf{sSC}^1 \subseteq \mathsf{SC}^1 = \mathsf{L}$, are any of the containments strict?
4. Are the levels of the sSC hierarchy closed under complement? We have only showed co-$\mathsf{sSC}^i \subseteq \mathsf{sSC}^{2i}$.
5. In the arithmetic setting, exactly where do the classes $\#\mathsf{sSC}^0$ and $\#\mathsf{sSC}^1$ lie? In particular, can we show that $\#\mathsf{sSC}^0$ equals $\#\mathsf{NC}^1$, or that $\#\mathsf{sSC}^1$ is in FL? What closure properties do these arithmetic classes possess?
6. A study of the language classes SNC^1 and $\mathsf{C}_{=}\mathsf{NC}^1$ (in particular, their closure properties) may reveal interesting connections. The exact counting and the threshold language class analogues of the classes $\#\mathsf{sSC}^0$ or $\#\mathsf{BWBP}$ may lead to potentially new complexity classes that might also turn out to be interesting.

Acknowledgments

The authors gratefully acknowledge the anonymous referees, whose comments helped improve the readability of the paper.

References

[AAD00] M. Agrawal, E. Allender, and S. Datta. On TC^0, AC^0, and arithmetic circuits. *Journal of Computer and System Sciences*, 60(2):395–421, 2000.

[AJMV98] E. Allender, J. Jiao, M. Mahajan, and V. Vinay. Non-commutative arithmetic circuits: depth reduction and size lower bounds. *Theoretical Computer Science*, 209:47–86, 1998.

[AKMV05] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming ICALP*, pages 1102–1114, 2005.

[All01] E. Allender. The division breakthroughs. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 74, 2001.

[All04] E. Allender. Arithmetic circuits and counting complexity classes. In Jan Krajicek, editor, *Complexity of Computations and Proofs*, Quaderni di Matematica Vol. 13, pages 33–72. Seconda Universita di Napoli, 2004. An earlier version appeared in the Complexity Theory Column, SIGACT News 28, 4 (Dec. 1997) pp. 2-15.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the ACM Symposium on Theory of Computing STOC*, pages 202–211, 2004.

[Bar89] D. A. M. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[BCD+89] A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal of Computing*, 18(3):559–578, 1989.

[BCGR92] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.

[Bus87] S. Buss. The Boolean formula value problem is in ALOGTIME. In *Proceedings of the ACM Symposium on Theory of Computing STOC*, pages 123–131, 1987.

[BV83] B. von Braunmühl and R. Verbeek. Input-driven languages are recognized in $\log n$ space. In *Proceedings of the Fundamentals of Computation Theory Conference FCT, LNCS*, pages 40–51, 1983.

[CDL01] A Chiu, G Davida, and B Litow. Division in logspace-uniform NC^1. *RAIRO Theoretical Informatics and Applications*, 35:259–276, 2001.

[CMTV98] H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic NC^1 computation. *Journal of Computer and System Sciences*, 57:200–212, 1998.

[Coo71] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of Association for Computing Machinery*, 18:4–18, 1971.

[Coo79] S. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the ACM Symposium on Theory of Computing STOC*, pages 338–345, 1979.

[DC89] P.W. Dymond and S. Cook. Complexity theory of parallel time and hardware. *Information and Computation*, 80:205–226, 1989.

[Dym88] P.W. Dymond. Input-driven languages are in $\log n$ depth. In *Information Processing Letters*, pages 26, 247–250, 1988.

[FLR96] H. Fernau, K.-J. Lange, and K. Reinhardt. Advocating ownership. In *Proceedings of the 16th Foundations of Software Technology and Theoretical Computer Science Conference FST&TCS, LNCS 1180*, pages 286–297, 1996.

[HL93] M. Holzer and K.-J. Lange. On the complexities of linear LL(1) and LR(1) grammars. In *Proceedings of the 9th International Symposium on Fundamentals of Computation Theory FCT, LNCS*, pages 299–308, 1993.

[IZ94] S. Istrail and D. Zivkovic. Bounded width polynomial size Boolean formulas compute exactly those functions in AC^0. *Information Processing Letters*, 50:211–216, 1994.

[Joh90] D. S. Johnson. A catalog of complexity classes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 67–161, Elsevier and MIT Press, 1990.

[Lan93] K.-J. Lange. Complexity and structure in formal language theory. In *Proceedings of the IEEE Structure in Complexity Theory Conference*, pages 224–23, 1993.

[LMM08] N. Limaye, M. Mahajan, and A. Meyer. On the complexity of membership and counting in height-deterministic pushdown automata. In *Proceedings of the 3nd International Computer Science Symposium in Russia CSR, LNCS 5010*, 2008.

[LMR07] N. Limaye, M. Mahajan, and B. V. R. Rao. Arithmetizing classes around NC^1 and l. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science STACS, LNCS 4393*, pages 477–488, 2007.

[Mah07] M. Mahajan. Polynomial size log depth circuits: between NC^1 and AC^1. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 91, 2007.

[Meh80] K. Mehlhorn. Pebbling mountain ranges and its application to DCFL-recognition. In *Proceedings of the 7th International Colloquium on Automata, Languages, and Programming ICALP, LNCS*, pages 422–432, 1980.

[Nis94] N. Nisan. RL $\subseteq$ SC. *Computational Complexity*, 4(11):1–11, 1994.

[NR95] R. Niedermeier and P. Rossmanith. Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits. *Information and Computation*, 118(2):227–245, 1995.

[Ruz80] W.L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[Sud78] I. Sudborough. On the tape complexity of deterministic context-free language. *Journal of Association of Computing Machinery*, 25(3):405–414, 1978.

[Ven91] H. Venkateswaran. Properties that characterize LogCFL. *Journal of Computer and System Sciences*, 42:380–404, 1991.

[Vin91] V Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proceedings of 6th IEEE Structure in Complexity Theory Conference*, pages 270–284, 1991.

[Vin96] V Vinay. Hierarchies of circuit classes that are closed under complement. In *Proceedings of the 11th Annual IEEE Conference on Computational Complexity CCC*, pages 108–117, Washington, DC, USA, 1996.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York Inc., 1999.

[vzG93] J. von zur Gathen. Parallel linear algebra. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 573–617. Morgan Kaufmann, 1993.

[vzGS91] J. von zur Gathen and G. Seroussi. Boolean circuits versus arithmetic circuits. *Information and Computation*, 91(1):142–154, 1991.

Appendix

$\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$ are known to be closed under decrement $f \ominus 1 = \max\{f-1, 0\}$ and under division by a constant $\lfloor \frac{f}{m} \rfloor$. In [AAD00], this observation for $\#\mathsf{NC}^1$ is credited to Barrington. However, we have not seen a published proof, so for completeness, we give details here.

We will repeatedly use the following fact:

Proposition 4 (Barrington [Bar89], see also [CMTV98]). *For any f in $\#\mathsf{BWBP}$ or $\#\mathsf{NC}^1$, the predicates $[f(x) = 0]$ and $[f(x) > 0]$ are in $\#\mathsf{BWBP}$ and $\#\mathsf{NC}^1$. That is, they can be computed by 0-1 valued arithmetic branching programs / circuits.*

Proof. Start with $f \in \#\mathsf{NC}^1$. By replacing $+$ by $\vee$ gates and $\times$ by $\wedge$ gates, we can see that the predicates $[f > 0]$ and $[f = 0]$ are in NC^1. By [Bar89], these predicates can be computed by deterministic branching programs. The $\#\mathsf{BWBP}$ functions computed by these programs are thus 0-1 valued, as desired. Since $\#\mathsf{BWBP} \subseteq \#\mathsf{NC}^1$, the predicates also have 0-1 valued $\#\mathsf{NC}^1$ circuits. □

Lemma 11. *The classes $\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$ are closed under decrement and division by a constant m.*

Proof. Let $f \in \#\mathsf{NC}^1$. First consider decrement. We show that $f \ominus 1 = \max\{f(x) - 1, 0\} \in \#\mathsf{NC}^1$ by induction on depth of the circuit. The base case, when depth is zero, is straightforward: $f \ominus 1 = 0$. Now consider a circuit of depth d computing f. f is either $g + h$ or gh for some g, h computed at depth $d - 1$.

$$(g + h) \ominus 1 = (g \ominus 1 + h) \times [g > 0] + (h \ominus 1) \times [g = 0]$$

$$(gh) \ominus 1 = [(g \ominus 1) \times h + h \ominus 1] \times [g > 0] \times [h > 0]$$

By induction and using Proposition 4, it follows that $f \ominus 1 \in \#\mathsf{NC}^1$.

Next consider division: we want to show $\lfloor \frac{f}{m} \rfloor \in \#\mathsf{NC}^1$. Note that

$$\left\lfloor \frac{g+h}{m} \right\rfloor = \left\lfloor \frac{g}{m} \right\rfloor + \left\lfloor \frac{h}{m} \right\rfloor + \left\lfloor \frac{g \mod m + h \mod m}{m} \right\rfloor$$

$$\left\lfloor \frac{gh}{m} \right\rfloor = \left\lfloor \frac{g}{m} \right\rfloor h + (g \mod m) \left\lfloor \frac{h}{m} \right\rfloor + \left\lfloor \frac{(g \mod m)(h \mod m)}{m} \right\rfloor$$

Now the required result follows from Proposition 3, using induction on depth.

In the case of $\#\mathsf{BWBP}$, we use induction on the length of the program. Let $f \in \#\mathsf{BWBP}$. Let w be the width and l be the length of the branching program P for f. We assume without loss of generality that all the edges in any one layer are labeled by the same variable (or a constant). The base case is branching programs of length one, in which case $f \ominus 1$ and $\lfloor \frac{f}{m} \rfloor$ are trivially 0, since $f \in \{0, 1\}$. Assume that for all branching programs P' with length at most $l - 1$, $\#P' \ominus 1$ and $\lfloor \frac{\#P'}{m} \rfloor$ are in $\#\mathsf{BWBP}$. Let P be a length l branching program. Let $S = \{v_1, v_2, \ldots, v_w\}$ be the nodes at level $l - 1$ of P. We also denote by v_i the value of the

#BWBP function computed at node v_i. Let the edges out of this level be labeled by 1, x or $\overline{x}$ for some variable x. Now f can be written as $f = \sum_{i \in S_1} v_i + x \sum_{i \in S_2} v_i + x \sum_{i \in S_3} v_i$, where nodes in S_1 have an edge labeled 1 to the output node, nodes in S_2 have an edge labeled x to the output node, and nodes in S_3 have an edge labeled $\overline{x}$ to the output node. Let U denote $\sum_{i \in S_2} v_i$ and Y denote $\sum_{i \in S_3} v_i$. Now

$$f \ominus 1 = [v_1 > 0](v_1 \ominus 1 + v_2 + \cdots + v_{j_1} + xU + \overline{x}Y) + [v_1 = 0][v_2 > 0](v_2 \ominus 1 + v_3 + \cdots + v_{j_1} + xU + \overline{Y}) + \cdots$$

Using Proposition 4 and induction, we see that $f \ominus 1$ can be computed within #BWBP, with width $(3w + 5)w + 2w$.
Note that, in order to achieve the above width bound, we need to stagger the programs for each term in the above sum. The constant 5 is for computing the predicates like $[v_i > 0]$ and $[v_i = 0]$, which follows from Barrington's construction([Bar89]). The length of the resulting program will be w^2lq, where q is an upper bound for the length of the branching programs which compute predicates $[v_i > 0]$ and $[v_i = 0]$.

$$\left\lfloor \frac{f}{m} \right\rfloor = \sum_{i=1}^{j_1} \left\lfloor \frac{v_i}{m} \right\rfloor + x. \sum_{i=1}^{j_2} \left\lfloor \frac{u_i}{m} \right\rfloor + \overline{x} \sum_{i=1}^{j_3} \left\lfloor \frac{y_i}{m} \right\rfloor + \left\lfloor \frac{\sum_{i=1}^{j_1} v_i \mod m + x \sum_{i=1}^{j_2} u_i \mod m + x \sum_{i=1}^{j_3} y_i \mod m}{m} \right\rfloor$$

For each i, $v_i \mod m, u_i \mod m$ and $y_i \mod m$ can be computed in NC^1. Since w is a constant, we can compute the whole sum in FNC^1 and hence in #BWBP. By our inductive hypothesis, all v_i, u_i and y_i's are in #BWBP, hence $\lfloor \frac{f}{m} \rfloor \in$ #BWBP. The width of the resulting program is bounded by $2mw$, and size by mwl. □