Electronic Colloquium on Computational Complexity, Report No. 78 (1998)

The Query Complexity of Program Checking by Constant-Depth Circuits

V. Arvind* K. V. Subrahmanyam † N. V. Vinodchandran ‡

Abstract

In this paper we study program checking (in the sense of Blum and Kannan [7]) using AC^0 circuits as checkers. Our focus is on the number of queries made by the checker to the program being checked and we term this as the *query complexity* of the checker for the given problem. We study the query complexity of both deterministic and randomized AC^0 checkers. We show that, for each $\epsilon > 0$, $\Omega(n^{1-\epsilon})$ is a lower bound to the query complexity of deterministic AC^0 checkers for Parity and certain P-complete and NC^1-complete problems, where n is the input size. On the other hand, we show that Parity and suitably encoded complete problems for P, NL, and NC^1 have randomized AC^0 checkers of *constant* query complexity. The latter results are proved using techniques from the PCP$(n^3, 1)$ protocol for 3-SAT in [4].

1 Introduction

In this paper we study program checking (in the sense of Blum and Kannan [7]) using AC^0 circuits as checkers. Our main focus is on the number of queries made by the checker to the program being checked and we term this parameter as the *query complexity* of the checker for the given problem. Clearly, the query complexity is an important parameter in the design of efficient program checkers; a large query complexity is a serious bottleneck for a checker that may otherwise be very efficient (in terms of sequential/parallel time, space etc.). We would like to mention that constant query checkers are highlighted in [1] as a notion of program checking that is of practical significance. For instance, it is shown in [1] that GCD has a constant query checker.

Before we explain our motivation for considering AC^0 circuits as checkers we recall that the seminal paper of Blum and Kannan [7] already describes an AC^0 checker. More precisely, they give a deterministic CRCW PRAM constant-time program checker for the P-complete problem *LFMIS* (lex. first maximal independent set problem for graphs). However, in the context of the present paper, we note that their checker has large query complexity (the number of queries is proportional to the input size).

Our motivation for considering AC^0 circuits as checkers is two-fold. Firstly, in a complexity-theoretic sense AC^0 represents the easiest model of parallel computation, and one aspect of program checking is to try and make the checker as efficient as possible. In this sense, AC^0 checkers can be seen as constant-time parallel checkers, since they essentially correspond to constant-time CRCW PRAM algorithms. The second motivation rests on the main goal of this paper, namely, to study the query complexity of program checkers. We choose a model of program checking that is amenable to lower bound techniques since one of our aims is to establish nontrivial lower bounds on the query

*Institute of Mathematical Sciences, Chennai 600113, India (email: arvind@imsc.ernet.in)
†SPIC Mathematical Institute, Chennai 600 017, India (email: kv@smi.ernet.in)
‡Institute of Mathematical Sciences, Chennai 600113, India (email: vinod@imsc.ernet.in)

ISSN 1433-8092

complexity of program checking. As it turns out, we are able to prove nontrivial lower bounds on the query complexity of deterministic AC^0 checkers. In particular, we show for the Parity problem that $\Omega(n^{1-\epsilon})$ is a lower bound for the query complexity of deterministic AC^0 checkers. In contrast, we show strong upper bounds for the query complexity of randomized AC^0 checkers. For Parity we are able to design a randomized AC^0 checker of *constant* query complexity.

Building on the results for Parity we next consider a suitably encoded version of the *Circuit Value Problem* (henceforth CVP). Different versions of the CVP are well-known to be complete for different important complexity classes: the unrestricted version is P-complete and the CVP problem for circuits that are formulas (i.e. fanout of each gate is one) is NC^1-complete.

Our first observation is that each of these CVP problems has deterministic AC^0 checkers, but in the naive construction the number of queries made by the checker for an instance of CVP is roughly the number of gates in the circuit. It turns out that our lower bounds for the query complexity of deterministic AC^0 checkers for Parity easily carry over to each of these CVP problems. We show that, for each $\epsilon > 0$, $\Omega(n^{1-\epsilon})$ is a lower bound to the query complexity of deterministic AC^0 checkers for each of these CVP problems.

As regards randomized AC^0 checkers for the CVP problems, we use ideas from the PCP theorem to again design *constant query* randomized AC^0 checkers for each of these CVP problems. The design of this checker uses the randomized AC^0 checker for Parity as subroutine and incorporates the main ingredients from the construction of the PCP$(n^3, 1)$ system for satisfiability based on linearity testing developed in [4].

A central aspect of our results and their proofs is the role of the parity function: the algebraic properties of GF(2) play a role in the upper bound result for randomized AC^0 checkers, and for the lower bound proofs for deterministic AC^0 checkers we invoke the constant depth circuit lower bound results for parity due to [2] (also refer to Hastad's thesis [10]).

As described in [4], the PCP theorem has evolved from interactive proofs [9] and program checking [7, 8]. In particular, there is a strong influence of ideas from self-correcting programs in the PCP$(n^3, 1)$ protocol for 3-SAT [4]. It is not surprising, therefore, that ingredients of the PCP$(n^3, 1)$ protocol find application in some of our results on program checking. Our emphasis on the query complexity of program checkers seems to naturally lead to the ideas underlying probabilistically checkable proofs. More applications of ideas from the PCP theorem to other specific problems in program checking appears to be an attractive area worth exploring.

2 Preliminaries

We first formally define program checkers introduced in [7].

Definition 1 [7] *Let A be a decision problem, a program checker for A, C_A, is a (probabilistic) oracle algorithm that for any program P (supposedly for A) that halts on all instances, for any instance x of A, and for any positive integer k (the security parameter) presented in unary:*

1. *If P is a correct program, that is, if $P(x) = A(x)$ for all instances x, then with probability $\geq 1 - 2^{-k}$, $C_A(x, P, k)$=Correct.*
2. *If $P(x) \neq A(x)$ then with probability $\geq 1 - 2^{-k}$, $C_A(x, P, k)$=Incorrect.*

The probability is computed over the sequences of coin flips that C_A could have tossed. Importantly, C_A is allowed to make queries to the program P on some instances.

When we speak of AC^0 checkers we mean that the checker C_A is described by a (uniform) family of AC^0 circuits, one for each input size.

We will also consider the (stronger) notion of deterministic checkability. The decision problem A is said to be *deterministically* checkable if C_A in the above definition is a deterministic algorithm.

Next we define the query complexity of AC^0 checkers.

Definition 2 *Let L be a decision problem that is deterministically* AC^0 *checkable. The* AC^0 *checker defined by the circuit family $\{C_n\}_{n\geq 0}$ is said to have* query complexity *$q(n)$ if $q(n)$ bounds the number of queries made the checker circuit C_n for any input $x \in \Sigma^n$.*

We now describe the CVP problems and the encodings of their instances. Let C denote a boolean circuit over the standard base (of NOT, AND, and OR gates). We consider circuits of fanin bounded by two. We will encode the circuit C as 4-tuples (g_1, g_2, g_3, t) where t is a couple of bits to indicate the type of the gate labeled g_1, and g_2 and g_3 are the gates whose values feed into the gate labeled g_1. For uniformity, we can assume that NOT gates are also encoded as such 4-tuples, except that $g_2 = g_3$. Furthermore, we insist that in the encoding, the gate labels g_i be *topologically sorted* consistent with the *DAG* underlying the circuit C. Thus, in each 4-tuple (g_1, g_2, g_3, t) present in the encoding of C, it will hold that $g_1 > g_2$ and $g_1 > g_3$. This stipulation ensures that checking whether an encoding indeed represents a circuit can be done in AC^0.

For a circuit C with n inputs let $C_g(x_1, x_2, \ldots, x_n)$ denote the value of the circuit C at gate g designated as output gate. We now define the *circuit value problem* which is the decision problem that we shall be mainly concerned with in this paper:

$$\{(C, g, x_1, x_2, \ldots, x_n) \mid C_g(x_1, x_2, \ldots, x_n) = 1\}$$

The circuit C is encoded as described above.

It is well known that the above circuit value problem is P-complete (under projection reducibility). We denote this by CVP. If the input circuit is a formula (i.e. each gate of the input circuit has fanout at most 1) the corresponding circuit value problem is known to be NC^1-complete under projection reducibility; we denote this problem by FVP (for formula value problem). Notice that an AC^0 circuit can check if a given circuit is a formula or not.

Finally, we make another important stipulation on the circuits that are valid inputs for all the circuit value problems that we consider: we insist that the *fanout* of each gate is bounded by the constant two.

Notice that this last restriction on the input circuits does not affect the fact that CVP remains P-complete (in fact, such a restriction already holds for the CVP in the standard P-completeness proof by simulating polynomial-time Turing machines). Also, observe that this extra stipulation on the input circuits can be easily tested in AC^0.

3 Deterministic AC^0 checkers for Parity

We first design deterministic AC^0 checkers for Parity and the circuit value problem.

Theorem 3 *For each constant k, there is a deterministic* AC^0 *checker for* $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$ *of query complexity $n/\log^k n$.*

Proof. Let P be an alleged program for the Parity function. First observe that the AC^0 checker can make parallel queries to P for $\mathrm{Parity}(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n$. In order to verify that the program's value of Parity$(x_1, x_2, \ldots, x_n)$ is correct the checker just has to verify that the answers to the queries for Parity$(x_1, x_2, \ldots, x_i)$ are all locally consistent in the following sense:

$$P(x_1, x_2, \ldots, x_{i+1}) = x_{i+1} \oplus P(x_1, x_2, \ldots, x_i)$$

for $2 \leq i \leq n-1$.

This can be easily done in parallel in AC^0 since query answers $P(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n$ are available.

This gives us an AC^0 checker which makes $n-1$ queries. In order to design a checker with the number of queries scaled down to $n/\log^k n$ notice that we can compute the parity of $\log n$ boolean variables in AC^0 by brute force. Thus, we can group the n input variables into $n/\log n$ groups of $\log n$ variables each, compute the parity of each group again by brute force, and the problem boils down to checking the program's correctness for the parity of $n/\log n$ variables which we can do as before with $n/\log n$ queries to the program. Clearly, we can repeat the above strategy of grouping variables for a constant number of rounds, and therefore achieve the query complexity of $n/\log^k n$ with an appropriate constant increase of depth of the checker circuit. This completes the proof. ■

Remark: Notice that the above result applies to checking iterated products over arbitrary finite monoids. The proof and construction of the checker is similar.

Indeed, the above idea of doing piece-wise 'local testing' in order to check global correctness was first used in the deterministic CRCW PRAM checker in [7] for an encoding of the P-complete problem *LFMIS*. The same idea works in general for the CVP problem. For the sake of completeness we explain this result (implicit in [7]).

Theorem 4 [7] CVP *has a deterministic* AC^0 *checker.*

Proof. The proof is exactly as for Theorem 3 so we skip the simple details. Let P be an alleged program for CVP and let $(C, g, x_1, \ldots, x_n)$ be an input instance for program P. The AC^0 checker first queries in parallel the program for

$$P(C, g, x_1, \ldots, x_n) \qquad \forall \text{ gates } g \in C$$

Next, for each tuple (g_1, g_2, g_3, t) in the circuit description C the checker verifies that the program's answers are consistent with the gate type. This is again done in parallel for each tuple.

Notice that the checker must also validate the input by verifying that the tuples that describe the circuit indeed describe an acyclic digraph. This is made sure as described in our encoding of the instances of the CVP. It suffices to check that $g_1 > g_2$ and $g_1 > g_3$ for each tuple (g_1, g_2, g_3, t), which can be done in AC^0. This completes the proof. ■

It can be proved exactly as above that FVP has a deterministic AC^0 checker.

We now turn to lower bounds on the query complexity of AC^0 checkers for CVP and FVP. We first observe the following property of languages L having deterministic AC^0 checkers.

Lemma 5 *Let L be a decision problem that is deterministically* AC^0 *checkable. Furthermore, suppose L has an* AC^0 *checker of query complexity $q(n)$. Then, for each $n > 0$, there is a nondeterministic* AC^0 *circuit that takes n input bits and $q(n)$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff $x \in L^{=n}$.*

Observe that, by symmetry, such nondeterministic AC^0 circuits also exist for $\overline{L}$.

The proof of the above lemma is a direct consequence of the definition of deterministic checkers and is simply a variation of a result on self-helping due to Schöning [11]: as observed, for instance, in [3], deterministic checking in polynomial-time coincides with the notion of self-helping defined by Schöning [11] who showed that languages that have self-helpers are already in NP $\cap$ co-NP. The above lemma is just an extension of the same fact to the setting where the checker is in AC^0. The only extra observation made in Lemma 5 is that the number of queries made by the checker naturally translates into the number of nondeterministic bits used by the nondeterministic circuit.

Before we prove our lower bound results we recall an important result due to Ajtai [2] on lower bounds for AC^0 circuits approximating Parity.

Theorem 6 [2] *For all constants k, c, and $\epsilon > 0$, there is no depth k circuit of size n^c that can compute* Parity$(x_1, x_2, \ldots, x_n)$ *correctly for more than a $1/2 + 2^{-n^{1-\epsilon}}$ fraction of the inputs.*

Lemma 7 Parity *does not have* AC^0 *checkers that make $O(n^{1-\epsilon})$ queries for any $\epsilon > 0$.*

Proof. Suppose Parity has AC^0 checkers that makes $O(n^{1-\epsilon})$ queries for some $\epsilon > 0$. From Lemma 5 it follows that for each $n > 0$, there is a nondeterministic AC^0 circuit that takes n input bits and $O(n^{1-\epsilon})$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff x has odd parity. By pigeon-hole principle we can fix the $O(n^{1-\epsilon})$ nondeterministic bits so that the resulting *deterministic* circuit rejects *all* inputs $x \in \Sigma^n$ such that x has even parity, and accepts at least $2^{-n^{1-\epsilon}}$ fraction of inputs $x \in \Sigma^n$ of odd parity. This is impossible since it contradicts Theorem 6 of Ajtai. This completes the proof. ∎

Theorem 8 CVP *(likewise* FVP*) does not have* AC^0 *checkers that make $O(n^{1-\epsilon})$ queries for any $\epsilon > 0$.*

Proof. We prove it for CVP. It follows similarly for the other problem. Notice that we can easily design an AC^0 circuit (call it C') such that given an instance $x \in \Sigma^n$ of Parity the AC^0 circuit produces an instance $(C, x_1, x_2, \ldots, x_n, g)$ of CVP such that

$$\mathrm{Parity}(x_1, x_2, \ldots, x_n) = 1 \text{ iff } (C, x_1, x_2, \ldots, x_n, g) \in \mathrm{CVP}$$

Moreover, the size of $(C, x_1, x_2, \ldots, x_n, g)$ is $O(n \log n)$, since C simply encodes the linear-sized circuit for Parity in the 4-tuple encoding we are using for CVP instances.

Now, assume that CVP has an AC^0 checker that makes $O(n^{1-\epsilon})$ queries for some $\epsilon > 0$. Combining the nondeterministic circuit given by Lemma 5 with the AC^0 circuit C', it is easy to see that we get a nondeterministic AC^0 circuit that takes n input bits and $O(n^{1-\delta})$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff x has odd parity, for some suitable $\delta > 0$. This contradicts Lemma 7 and hence completes the proof. ∎

4 A randomized AC^0 checker for Parity

In this section we turn to the question of randomized AC^0 checkers for the parity function. We show that Parity has a randomized AC^0 checker of *constant* query complexity. This brings out the power of randomness in sharp contrast to the lower bound on query complexity for deterministic AC^0 checkers proved in the previous section.

In this and the next section we crucially use the linearity test. Firstly, we recall the relevant definition and result from [8].

Definition 9 [8] *Let F be $GF(2)$ and f, g be functions from F^n to F. The relative distance $\Delta(f, g)$ between f and g is the fraction of points in F^n on which they disagree. If $\Delta(f, g) \leq \delta$ then f is said to be δ-close to g.*

Theorem 10 [8] *Let F be $GF(2)$ and f be a function from F^n to F such that when we pick y, z randomly from F^n,*

$$Prob[f(y) + f(z) = f(y + z)] \geq 1 - \delta$$

where $\delta < 1/6$. Then f is 3δ-close to some linear function.

The theorem gives a linearity test that needs to evaluate f at only a constant number of points in F^n, where the constant depends on δ. If f passes the test then the function is guaranteed to be 3δ-close to some linear function.

Another theorem from [8] that we use is the one about self-correction. We state the result.

Theorem 11 [8] *Suppose $f, \hat{f} : \{0,1\}^n \to \{0,1\}$, $\hat{f}$ is a linear function that is δ-close to f. Then*

$$\Pr_{r \in \{0,1\}^n}[f(x + r) - f(r) \neq \hat{f}(x)] \leq 2\delta$$

Given a function f guaranteed to be δ-close to a linear function $\hat{f}$, and an x in the domain of f, let us denote by SC-$f(x)$ the value $f(x + r) - f(r)$, for a randomly chosen r in the domain of f. Then the above theorem guarantees that with high probability SC-$f(x)$ is equal to $\hat{f}(x)$.

We now state and prove the result of this section.

Theorem 12 Parity *has a randomized* AC^0 *checker of constant query complexity.*

Proof. Let P be a purported program for computing Parity. We first describe the checker.

The randomized checker for parity.

Test 0. The checker first checks that the function computed by the program is δ-close to a linear function (for a suitably small δ, say, 0.01). From Theorem 10 the checker can do this with a constant number of queries to P. For a constant number of random point pairs $\vec{y}, \vec{z}$ it checks if $P(\vec{y}) + P(\vec{z}) = P(\vec{y} + \vec{z})$. If for some pair this fails the checker rejects P as incorrect.

Test 1. The aim of this test is to see if the linear function to which P is δ-close is the zero function and reject if so. The checker builds the fixed vector $\vec{p}$ with $p_1 = 1$ and all other coordinates set to 0. Clearly this can be done in AC^0. It then computes the value SC-$P(\vec{p})$. If it is zero it rejects, since the parity of $\vec{p}$ is 1.

Test 2. The checker generates a random element z_1 of the vector subspace $\Sigma_i x_i = 0$. It then computes the value SC-$P(z_1)$. If this is 1 it rejects, since the parity of z_1 is even.

Notice that generating a random element of the subspace $\Sigma_i x_i = 0$ can be done in random AC^0: The vector subspace $\Sigma_i x_i = 0$ has a basis consisting of the following $n - 1$ vectors, $b_1 = \{1, 1, 0, \ldots, 0\}$,$b_2 = \{0, 1, 1, 0, \ldots, 0\}$,. . .,$b_{n-1} = \{0, 0, \ldots, 1, 1\}$ To generate a random element of this subspace, the checker picks up a random vector k in $\{0,1\}^{n-1}$ and computes $\Sigma_i k_i b_i$. Although this involves a parity computation for each coordinate, notice that for each coordinate there are at most two non-zero terms (in known positions) in the parity sum to be computed. Thus, such a parity computation can be easily done in AC^0.

Test 3. The checker computes the value SC-$P(x)$. If this is the same as the value of $P(x)$ it accepts, otherwise it rejects.

We now prove the correctness of the checker.

Claim. *For an input x, the above checker makes a constant number of queries to P and accepts with probability 1 if P correctly computes Parity, and rejects with probability $1/4$ if $P(\vec{x}) \neq \text{Parity}(\vec{x})$.*

Proof of Claim. Clearly the checker only makes a constant number of queries which can be generated in AC^0It is also clear that the checker accepts with probability 1 if P computes parity exactly. In this case, since all the queries made to P are parity queries, they are answered correctly by P.

Now assume that $P(x) \neq \text{Parity}(x)$. We denote by z the concatenation of all random bits that the checker accesses. Let F denote the event that the checker fails, and let L denote the event that the checker passes the linearity Test 0 step. We need to determine $\text{Prob}_z[F]$. However, since $F \subseteq L$, notice that $\text{Prob}_z[F] \leq \text{Prob}_z[F \mid L]$ and so it is enough to bound the right hand side of this expression.

Conditioned on event L there is a unique linear function $\hat{f}$ that is δ-close to the program P.

$$\text{Prob}_z[F \mid L] = \text{Prob}_z[F \;\&\; \hat{f} = 0 \mid L] + \text{Prob}_z[F \;\&\; \hat{f} \neq 0 \mid L]$$

For the first term to contribute, Test 1 must fail. From Theorem 11 this happens with probability at most 2δ.

We rewrite the second term as

$$\text{Prob}_z[F \;\&\; \hat{f} \neq 0 \mid L] = \text{Prob}_z[F \;\&\; \hat{f} \neq \text{Parity} \;\&\; \hat{f} \neq 0 \mid L] + \text{Prob}_z[F \;\&\; \hat{f} = \text{Parity} \;\&\; \hat{f} \neq 0 \mid L]$$

and bound the two terms separately.

Note that if $\hat{f} \neq \text{Parity}$, then for a random z_1, in Test 2, $\text{Prob}_{z_1}[\hat{f}(z_1) = 0 \;\&\; \hat{f} \neq 0 \mid L] = 1/2$. It follows from Theorem 11 and this observation that

$$\text{Prob}_z[F \;\&\; \hat{f} \neq \text{Parity} \;\&\; \hat{f} \neq 0 \mid L] \leq 1/2 + 2\delta$$

Finally from Theorem 11 it is clear that

$$\text{Prob}_z[F \;\&\; \hat{f} = \text{Parity} \;\&\; \hat{f} \neq 0 \mid L] = 2\delta$$

So the overall probability of failure is at most $1/2 + 8\delta$. By choice of δ this probability is at most $3/4$ as claimed. □

This completes the proof of the theorem.

■

5 A constant query randomized AC^0 checker for CVP

In this section we design a constant query AC^0 checker for the CVP problem (also for FVP). We make use of the main ideas from the $\text{PCP}(n^3, 1)$ protocol for 3-SAT from [4]. A crucial point of departure from [4] is when the checker needs to compute the parity of a multiset of input variables and products of input variables. To do this we use as subroutine the checker described in Theorem 12. Another point to remember is that all queries have to be valid instances of CVP (or FVP as the case may be), and they need to be generated in AC^0.

The starting point that leads us to applying techniques from the proof of the PCP theorem is the deterministic AC^0 checker for CVP described in Theorem 4. Recall that given a CVP instance $(C, g, x_1, \ldots, x_n)$ the deterministic AC^0 checker queries the purported program for $(C, g_i, x_1, \ldots, x_n)$, for each gate g_i in the circuit C. Then it checks that the query answers of the program are locally consistent for each gate of the circuit.

Let $y_1, y_2, \ldots, y_m$ denote the list of query answers by P for the queries $(C, g_i, x_1, \ldots, x_n)$, $1 \leq i \leq m$, where C has m gates. We can think of the *unique correct* vector $y_1, y_2, \ldots, y_m$ as a satisfying assignment to the collection of all the gate conditions (each of which is essentially a 3-literal formula). The idea is to avoid querying explicitly for y_i's. Instead, using randomness the checker will make fewer queries for other inputs that encode the y_i's in some form. More precisely, we need to encode the vector $y_1, y_2, \ldots, y_m$ in such a way that making a constant number of queries to the program (which is similar to a constant number of probes into a proof by a PCP protocol) can convince the AC^0 checker with high probability that this underlying vector $y_1, y_2, \ldots, y_m$ is indeed consistent with all the gate conditions.

Before giving the formal details we recall another lemma from [4].

Lemma 13 [4] *Let $\vec{a}$ and $\vec{b}$ be vectors in $GF(2)^n$. Suppose $\vec{b} \neq \vec{a} \otimes \vec{a}$, then*

$$\mathrm{Prob}[\vec{r}^t(\vec{a} \otimes \vec{a})\vec{s} \neq \vec{r}^t \vec{b} \vec{s}] \geq 1/4$$

where $\vec{r}$ and $\vec{s}$ are randomly chosen from $GF(2)^n$.

Theorem 14 *The* P*-complete problem* CVP *(likewise the* NC^1*-complete problem* FVP*) has a randomized* AC^0 *checker of constant query complexity.*

Proof. We describe the checker only for the P-complete problem CVP (the checker for FVP is similar). Let P be a purported program for CVP and let $(C, g, x_1, \ldots, x_n)$ be an input instance. Assume C has m gates. Let $g_1, \ldots, g_m$ denote the set of gates of C and w.l.o.g. assume $g_m = g$. The naive deterministic checker described in Theorem 4 queries P for $(C, g_i, x_1, \ldots, x_n)$, for each gate g_i in the circuit C. It then performs a local consistency check to test the validity of $(C, g_m, x_1, \ldots, x_n)$. The idea is to use randomness and avoid querying the program explicitly for the value $y_i := (C, g_i, x_1, \ldots, x_n)$ at each gate.

Let p_i denote a GF(2) polynomial corresponding to the ith gate of the circuit C, where p_i is a polynomial in at most three variables (these three variables are from $\{x_1, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_m\}$). We define p_i such that it is zero iff the corresponding variables are consistent with the gate type of g_i. More precisely, let g be an AND gate with output x and inputs y and z then the polynomial corresponding to g is $x + yz$. Similarly, for an OR gate with output x and inputs y and z the polynomial is $x + y + z + yz$, and for a NOT gate with output x and input y it is $x + y + 1$.

The checker

The checker first queries the program on the given input $(C, g, x_1, \ldots, x_n)$ and sets $y_m = P(C, g, x_1, \ldots, x_n)$. To convince itself that C evaluates to y_m on the input $\vec{x} = x_1, \ldots, x_n$, as in [4], it suffices for the AC^0 checker to verify that the following linear function

$$f(\vec{x}, \vec{y}, \vec{z}) = \Sigma_{i=1}^m p_i(\vec{y}) z_i$$

in the new variables $z_i, 1 \leq i \leq m$ is the zero linear function.[1]

Notice that if this function is nonzero then the linear function $f(\vec{x}, \vec{y}, \vec{z})$ evaluated at a randomly chosen $\vec{z} := \langle z_1, \ldots, z_m \rangle$ would be nonzero with probability 1/2. If it were the zero function then it must be zero with probability 1.

Recall that the checker must compute this value by asking a series of CVP queries that are generated in AC^0. Towards this end, we rewrite the above expression as follows:

$$f(\vec{x}, \vec{y}, \vec{z}) = p(\vec{x}, \vec{z}) + q(\vec{y}, \vec{z}) + r(\vec{y}, \vec{z})$$

where,

$$p(\vec{x}, \vec{z}) = \Sigma_{i=1}^{n} \Sigma_{k=1}^{m} \chi_i^k x_i + \Sigma_{i=1}^{n} \Sigma_{j=1}^{n} \Sigma_{k=1}^{m} \chi_{ij}^k x_i x_j$$

$$q(\vec{y}, \vec{z}) = \Sigma_{k=1}^{m} c_i * y_i$$

$$r(\vec{y}, \vec{z}) = \Sigma_{(i,j) \in [m] \times [m]} c_{ij} * y_i y_j$$

In the above expression the coefficient χ_i^k is 1 iff input x_i appears in the polynomial p_k and if z_k is 1. Likewise the coefficient χ_{ij}^k, is 1 iff $x_i x_j$ appears in the polynomial p_k and z_k is 1. From this it is easy to see that an $nm + n^2 m$ length Boolean vector representing each term in p can be obtained in AC^0.

Notice that the coefficients c_i and c_{ij} in q and r depend upon the gates g_i and g_j, the constant number of gates they feed into, the z values corresponding to these gates and a constant number of input bits. So computing each of these coefficients involves computing the parity of a *constant* number of Boolean variables. This can also be done in AC^0.[2]

We describe below how the checker computes the values $\tilde{p}$, $\tilde{q}$, and $\tilde{r}$ with high confidence. To complete the checking the checker evaluates $\tilde{p} + \tilde{q} + \tilde{r}$ and accepts $P(x)$ as correct only if this sum is zero.

Computing p Note that p is the parity of $nm + n^2 m$ Boolean variables. As noted above the value of these variables can be obtained in AC^0 given $\vec{x}$ and $\vec{z}$. The checker constructs a description of a canonical circuit for the parity of $nm + n^2 m$ variables, and queries the program on this input. Next the AC^0 checker checks the answer of P using the checker of Theorem 12 as subroutine. If the answer is wrong then with high probability the subroutine checker will reject the program as incorrect. Thus the AC^0 checker computes a value $\tilde{p}$ which is p with high (constant) probability. In the process only a constant number of queries are made to P.

Notice that, unlike in [4], we have to deal with both $y_1, \ldots, y_m$ as well as $x_1, x_2, \ldots, x_n$ which occur in the polynomials p_i. The crucial difference between the x_j's and y_j's is that $x_1, x_2, \ldots, x_n$ are *bound* to the input values. Thus, computing the value of p is a *parity computation* which the checker requires to get done. As explained above, this is done using the checker of Theorem 12 as subroutine.

Computing q and r To compute q and r the checker goes through the following steps.

1. It builds a circuit C_1 with new inputs $r_1, r_2, \ldots, r_m$ that computes the function $\Sigma_{i=1}^{m} y_i r_i$. Recall that y_i is the output of the ith gate of the input circuit C on input $x_1, x_2, \ldots, x_n$. Clearly, the encoding of C_1 can be generated by an AC^0 circuit from the encoding of C.

[1]Although p_i is a function of both $\vec{x}$ and $\vec{y}$ we write p_i as a function of only $\vec{y}$ for readability.

[2]It is easier to first conceive of a constant time CRCW PRAM algorithm for this task.

2. Similar to the above step, the checker builds a circuit C_2 with new inputs $r_{ij}, 1 \leq i, j \leq m$ that computes $\Sigma_{i=1}^m \Sigma_{j=1}^m y_i y_j r_{ij}$. It is clear that an encoding for C_2 can also be generated by an AC^0 circuit.

3. The checker verifies that the program's behavior on C_1 is a function that is δ-close to a linear function in the variables r_i, and the program's behavior on C_2 is a function that is δ-close to a linear function in the variables r_{ij}. This can be done as described in the previous section using Theorem 10. If either of the tests fails, it rejects the program as being incorrect.

4. Like in the PCP protocol the checker now performs a consistency check. Let $\Sigma_{i=1}^m \Sigma_{j=1}^m b_{ij} r_{ij}$ be the linear function to which C_2 is δ-close. The checker does a constant query test, and ensures with high probability that the matrix b_{ij} is the tensor product of $\vec{y}$ with itself.

   To do this the checker employs the test given by Lemma 13. For two randomly random vectors r_1, r_2 of length m it verifies that

   $$\text{SC-}C_1(r_1) * \text{SC-}C_1(r_2) = \text{SC-}C_2(r_1 \otimes r_2)$$

   Note that the tensor product can be computed in AC^0 and the checker needs to ask 6 queries of P.

Having performed the linearity and consistency tests the checker evaluates q and r by self-correction. Let $\vec{c}$ denote the m-vector $c_1, c_2, \ldots, c_m$ and let $\vec{d}$ denote the $m \times m$-vector consisting of $c_{ij}, 1 \leq i, j \leq m$. The checker sets $\tilde{q}$ to $\text{SC-}C_1(\vec{c})$ and $\tilde{r}$ to $\text{SC-}C_2(\vec{d})$.

Correctness.

Firstly, if P is a correct program for CVP then we can easily see that the checker will pass P as correct with probability 1.

Now suppose P is incorrect for the input instance $(C, g, x_1, \ldots, x_n)$ and let $P(C, g, x_1, \ldots, x_n) = b$.

Let F be the event that the checker fails to detect the program as incorrect. Let T be the event that the checker passes the linearity and consistency tests done in the course of computing q and r. Let us denote by w the concatenation of all random strings the checker needs. We need to compute $\text{Prob}_w[F]$. However, notice that since $F \subseteq T$ it holds that $\text{Prob}_w[F] \leq \text{Prob}_w[F \mid T]$ and so it suffices to bound the right-hand side.

Since we are conditioning on the event T we may assume that C_1 is δ-close to a unique linear function of the r_i's, $\Sigma_{i=1}^m y_i r_i$, wherein $y_m = b$. Likewise C_2 computes a function δ-close to the linear function $\Sigma_{(i,j)\in[m]\times[m]} y_i y_j * c_{ij}$. Let $\hat{y} = \langle y_1, \ldots, y_m \rangle$ be this unique linear function.

Since P is incorrect on input x, for this vector $\hat{y}$, the function[3]

$$f(\hat{y}, \vec{z}) = \Sigma_{i=1}^m p_i(\hat{y}) z_i$$

is a nonzero linear function of the z_i's. Hence,

$$\text{Prob}_w[f(\hat{y}, \vec{z}) = 1 \mid T] = \frac{1}{2}$$

Now,

[3]Notice that in the sequel we drop $\vec{x}$ from $f(\vec{x}, \hat{y}, \vec{z})$ and write $f(\hat{y}, \vec{z})$ for readability.

$$\begin{aligned}\mathrm{Prob}_w[F \mid T] &= \mathrm{Prob}_w[F \;\&\; f(\hat{y},\vec{z}) = 1 \mid T] + \mathrm{Prob}_w[F \;\&\; f(\hat{y},\vec{z}) = 0 \mid T] \\ &\le \mathrm{Prob}_w[F \;\&\; f(\hat{y},\vec{z}) = 1 \mid T] + 1/2 \\ &= \mathrm{Prob}_w[f(\hat{y},\vec{z}) = 1 \mid T] * \mathrm{Prob}_w[F \mid f(\hat{y},\vec{z}) = 1 \;\&\; T] + 1/2 \\ &= 1/2 * \mathrm{Prob}_w[F \mid f(\hat{y},\vec{z}) = 1 \;\&\; T] + 1/2\end{aligned}$$

Since $f(\hat{y},\vec{z}) = p(\vec{x},\vec{z}) + q(\hat{y},\vec{z}) + r(\hat{y},\vec{z})$ and F is the event that $\tilde{p}+\tilde{q}+\tilde{r} = 0$, we observe

$$\begin{aligned}& \mathrm{Prob}_w[F \mid f(\hat{y},\vec{z}) = 1 \;\&\; T] \\ = \;& \mathrm{Prob}_w[\tilde{p} \ne p(\vec{x},z) \mid T] + \mathrm{Prob}_w[\tilde{q} \ne q(\hat{y},z) \mid T] + \mathrm{Prob}_w[\tilde{r} \ne r(\hat{y},z) \mid T]\end{aligned}$$

From Theorem 12 the first term is bounded by 3/4. Given the event T, each of the other two terms is bounded by 2δ. So we get,

$$\mathrm{Prob}_w[F \mid T] \le 1/2 * (3/4 + 4\delta) + 1/2$$

This is smaller than 15/16 if we choose δ smaller than 1/32. By usual amplification techniques we can make the error probability an arbitrarily small constant by repeating the checker a constant number of times. This completes the proof. ■

Finally, we note that using similar techniques we can also show that there are encodings of NL-complete problems that have constant query randomized AC^0 checkers. Recall that the circuit value problem for *skew* circuits (i.e. a circuit in which at least one of the inputs of each AND gate is a circuit input) is NL-complete. However, if we follow the proof of the above theorem for skew circuits, notice that the construction of circuit C_2 will not preserve the skew property. However, we can get around this problem by considering circuits which have a formula on top whose inputs are the outputs of skew circuits. Notice that, by the closure properties of NL, the circuit value problem for such circuits is also NL-computable (and hence NL-complete). We can encode these new circuits by putting an additional tag bit on each gate label that indicates if the gate is part of the formula on top or not. The restriction on the circuit is now as follows: Tagged gates all have fanout 1, the untagged AND gates respect the skewness property, and no tagged gate feeds into an untagged gate. These constraints are easily checked in AC^0. The proof of Theorem 14 will now go through for this NL-complete problem. We thus have our final result.

Theorem 15 *There is an NL-complete circuit value problem for that has a randomized* AC^0 *checker of constant query complexity.*

Acknowledgments. For interesting discussions on AC^0 lower bounds and the results in Section 3 we thank Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh.

References

[1] L. A. Adleman, H. Huang, K. Kompella, Efficient checkers for number-theoretic computations, *Information and Computation,* **121**, 93–102, 1995.

[2] M. Ajtai, Σ_1^1 formulas on finite structures, *Annals of Pure and Applied Logic,* **24**, (1983) 1-48.

[3] V. Arvind, Constructivizing membership proofs in complexity classes, *International Journal of Foundations of Computer Science,* **8(4)** 433–442, 1997, World Scientific.

[4] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, Proof Verification and the intractability of approximation problems. In *Proceedings 33rd Symposium on the Foundations of Computer Science,* 14–23, IEEE Computer Society Press, 1992.

[5] J. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity II,* Springer–Verlag, 1990.

[6] J. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity I,* Springer–Verlag, second edition, 1995.

[7] M. Blum and S. Kannan, Designing programs that check their work, *Journal of the ACM,* **43**:269–291, 1995.

[8] M. Blum, M. M. Luby, and R. Rubinfeld, Self-testing/correcting with applications to numerical problems, *J. Comput. Syst. Sciences,* **47**:73–83, 1993.

[9] S. Goldwasser, S. Micali and C. Rackoff, The knowledge complexity of interactive proof systems. *SIAM Journal of Computing,* **18(1)**:186-208, 1989.

[10] J. Hastad, Computational limitations for small depth circuits. M.I.T. press, Cambridge, MA, 1986.

[11] U. Schöning, Robust algorithms: a different approach to oracles, Theoretical Computer Science, **40**: 57–66, 1985.