Improved Bounds on the Space Complexity of Circuit Evaluation

Yakov Shalunov
Department of Computer Science, University of Chicago, IL, USA

Abstract

Williams (STOC 2025) recently proved that time-t multitape Turing machines can be simulated using $O(\sqrt{t \log t})$ space using the Cook-Mertz (STOC 2024) tree evaluation procedure. As Williams notes, applying this result to fast algorithms for the circuit value problem implies an $O(\sqrt{s} \cdot \operatorname{polylog} s)$ space algorithm for evaluating circuits with s gates.

In this work, we provide a direct reduction from circuit value to tree evaluation without passing through Turing machines, simultaneously improving the bound to $O(\sqrt{s \log s})$ space and providing a proof with fewer layers of abstraction.

This result can be thought of as a "sibling" result to Williams' for circuit complexity instead of time; in particular, using the fact that time-t Turing machines have size $O(t \log t)$ circuits, we can recover a slightly weakened version of Williams' result, simulating time-t machines in space $O(\sqrt{t} \log t)$.

2012 ACM Subject Classification Theory of computation → Circuit complexity; Theory of computation → Communication complexity

Keywords and phrases circuit value problem CVP, space complexity, tree evaluation problem

Digital Object Identifier 10.4230/LIPIcs.CCC.2026.3

Supplementary Material *Software (Formalization of key lemma in Lean)*: `https://github.com/Yakov-Shalunov/space-complexity-circuiteval-formalization`
archived at `swh:1:dir:a0330b101611eb4eccc32994641173be36fed69e`

Acknowledgements I would like to thank William Hoza for suggesting this research direction, helping review this article, and general guidance. I am also grateful to Alexander Razborov for feedback on presentation.

1 Introduction

Recently, Williams proved an extremely counterintuitive result:

▶ **Theorem 1** (Williams [15]). *All time-$t(n) \geq n$ multitape Turing machines can be simulated using $O(\sqrt{t \log t})$ bits of space.*

This holds no matter how they use the $\Omega(t)$ cells of space they can touch and improves dramatically on the 50-year-old best-known simulation using space $O(t/\log t)$ due to Hopcroft, Paul, and Valiant [9].

This simulation can be composed with Turing machine algorithms for evaluating circuits to obtain space-efficient algorithms for circuit evaluation. In particular, the best known algorithm for circuit simulation on Turing machines is due to Pippenger [11] (Theorem 7) and yields a space $O(\sqrt{s} \cdot \log^{3/2} s)$ algorithm for evaluating arbitrary size-s circuits.

Motivated by the fact that circuits themselves are an appealing model of fine-grained complexity, we directly prove an improved bound on the space complexity of circuit evaluation:

▶ **Theorem 2** (Main theorem). *Given a size s circuit C on $n \leq s$ inputs and an input $x \in \{0,1\}^n$, the output $C(x)$ can be evaluated in space $O(\sqrt{s \log s})$. Further, a logspace algorithm for the tree evaluation problem would immediately imply a space $O(\sqrt{s})$ algorithm for circuit evaluation.*

The uniform formulation immediately implies a nonuniform version:

▶ **Corollary 3.** *If $f : \{0,1\}^n \to \{0,1\}$ can be computed by a size s circuit then f can be computed by a size-$2^{O(\sqrt{s \log s})}$ branching program.*

Additionally, Theorem 2 together with standard results for simulating Turing machines with circuits (see Theorem 9) gives $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}\left[\sqrt{t}\log t\right]$, re-proving Williams' result up to log factors.

Technique outline

Similarly to Williams', our result is based on a reduction to the tree evaluation problem. Roughly speaking, the tree evaluation problem (Definition 13) is the task of evaluating a height h, d-ary tree where each leaf is a b-bit value and each internal node is a function $\{0,1\}^{db} \to \{0,1\}^b$, given as an explicit table of 2^{db} b-bit entries.

The natural depth-first approach requires space $O(hdb)$ to evaluate the tree, storing db bits at each of h recursive levels. Though this depth-first approach intuitively seems "inherent" to the problem (indeed, the problem was originally posed to attempt to separate L from ¶), Cook and Mertz came up with an $O(h \log db + db)$-space algorithm for the problem, improving the space complexity quadratically when $h \approx db$ [4].[1]

Williams' result reduces Turing machine computation to a tree evaluation instance of arity $d = O(1)$ with height $h = \Theta(t/b)$ and value-size b for some parameter b – evaluating this instance naively earns nothing, since $\Theta\left(\frac{t}{b}\right) \cdot b = \Omega(t)$, but balancing t/b and b and then applying the Cook-Mertz tree evaluation procedure yields quadratic savings.

We provide a direct reduction from circuit evaluation to tree evaluation which bypasses reasoning about the motion of Turing machine heads.

For intuition, we first sketch a warm-up result:

▶ **Proposition 4.** *Size s circuits can be evaluated in space $\tilde{O}(s^{2/3})$.*

Proof sketch. Similar to the approach used for Turing machines, we partition our circuit into blocks of size b. This, in particular, ensures that the *number* of blocks (and thus depth) is $\frac{s}{b}$. Our "blocks" will be intervals of consecutive vertices under a topological ordering, thus ensuring that the quotient graph[2] is a DAG.

We can view this quotient DAG as a "circuit" over values from $\{0,1\}^b$ instead of over individual bits: the value at each vertex in the quotient graph is the list of values of all b gates in the corresponding block of the original circuit and each wire carries all b of those bits.

Since the quotient DAG connects blocks together when any pair of underlying gates have a connection, all gates in the block have sufficient information to be evaluated and our "quotient circuit" is able to simulate the original circuit.

Now we observe that in much the same way that one can expand a circuit into a formula, we can expand this quotient circuit into a tree evaluation instance (i.e., duplicating each vertex once for every path to reach it from the root). This increases the size exponentially (which is fine, because we'll never write down the whole tree, instead computing it on-demand) but, crucially, does not affect the depth.

---

[1] Several years prior to this $O(\log n \cdot \log\log n)$ result, Cook and Mertz first showed that the problem can be solved in space $O(\log^2 n/\log\log n)$ [2, 3]. (Here, $n = \tilde{\Theta}(d^h \cdot 2^{db})$ so $\log n = \Theta(h \log d + db)$.)

[2] Here, when we say "quotient" we mean a graph with a vertex for each part of the partition and a single edge between any two parts whose components have any edges in the original graph. A more formal definition can be found in the preliminaries in Section 2.

Since the depth of the original quotient circuit was bounded by the number of blocks in it, which in turn was $\frac{s}{b}$, the height of our tree evaluation instance is $h = \frac{s}{b}$, and since each block has b gates, the value-size parameter is exactly b.

In order to get $\tilde{O}(\sqrt{s})$, we ultimately need an arity of $d = O(1)$; unfortunately the naive construction given here only allows a trivial bound on the in-degree: given a block B, each wire coming into B (i.e., edges (u, v) with $v \in B$) in the original circuit connects to one vertex and thus adds at most one block (the block $B(u)$ containing u) to the list of blocks with wires to B. Since there are b gates and each has two inputs (we work with binary circuits), there are at most $2b$ total incoming wires, and thus at most $2b$ connected blocks.

This gives us $d \approx b$ and so the Cook-Mertz procedure gives us space $O(\frac{s}{b} \log b^2 + b^2)$. Setting $b \approx \sqrt[3]{s}$ gives $\tilde{O}(s^{2/3})$. ◀

In order to get $\tilde{O}(\sqrt{s})$, we will need the key technical contribution of this work: a trick for reducing the in-degree of the quotient graph, expressed (in slightly simplified form[3]) in the following graph-theoretic lemma:

▶ **Lemma 5** (Low-degree DAG partition (simplified)). *For any directed acyclic graph G of size s and maximum in-degree $d > 1$ and any integer choice of $b \in [d, s]$, there is a subdivision[4] G' of G and a partition P of G' such that:*

- *Every part in P has at most b vertices.*
- *Every vertex in the quotient graph G'/P has in-degree at most 2.*
- *All directed paths in G'/P have length less than $d \cdot \frac{s}{b}$.*

Since we consider binary circuits, our application will have $d = 2$, and $b = \tilde{\Theta}(\sqrt{s})$, yielding a depth of $2 \cdot \frac{s}{b} - 1 = \tilde{O}(\sqrt{s})$ and giving us the desired result.

Intuitively, we are "buying" a lower depth by "paying" with increased block size – were we doing a naive depth-first evaluation, this would get us nothing, but using Cook-Mertz tree evaluation procedure allows us to have increased block size "for free" (up to the point where it exceeds depth).

▶ Remark 6. When we reduce the depth of the graph in the proof of the lemma, we do so by "brute-force," of a sort: we first create a quotient graph on the original graph G which has a total *size* of $d \cdot \frac{s}{b}$ (thereby bounding the depth) and then use careful subdivision to reduce the in-degree without increasing the depth. This allows us to reduce the depth of an *arbitrary* graph without care for its structure since the depth of a graph cannot exceed its size.

Outline

In Section 2, we give some preliminaries on the circuit and tree evaluation problems. In Section 3, we prove the Partition Lemma (Lemma 5). Finally, in Section 4, we apply the Partition Lemma to reduce circuit evaluation to an appropriate TreeEval instance.

[3] The full version is stated as Lemma 15; it provides additional complexity guarantees and allows for trading off between the final in-degree and depth. (In particular, a final in-degree $d + 1$ allows a depth of s/b.)

[4] A subdivision of a graph subdivides some edges by inserting vertices into them (i.e., replacing $e = (u, v)$ with a vertex v_e and edges $(u, v_e), (v_e, v)$). In the context of circuits specifically, these will be additional identity gates spliced into some wires, which trivially does not affect behavior.

2 Preliminaries

Model relationships. First, the following pair of results together illustrate the close connection between circuits and Turing machine time:

▶ **Theorem 7** (Pippenger [11]). *Given a size s circuit C and an input x, the output $C(x)$ can be evaluated in multitape Turing machine time $O(s \log^2 s)$.*

▶ Remark 8. Pippenger's proof of this result does not appear to be available online; correspondingly, we have included an exposition of the algorithm in appendix A. See also Williams' discussion of the result [15].

▶ **Theorem 9** (Pippenger, Fischer [12]; Koucký, cf [1]). *For every $t(n) \geq n$ and $L \in \mathsf{TIME}[t]$, the* log*-space-uniform circuit complexity of L is $O(t \log t)$.*

▶ Remark 10. While this result is originally due to Pippenger and Fischer (proven by passing through an oblivious Turing machine simulation), Koucký developed a simpler direct proof [1]; while Koucký's original proof does not appear to be digitally available, an exposition of this result by Viola can be found in Theorem 1.6 of [14].

Additionally, we follow Williams and remark that unlike for time complexity (where best-known simulations incur polylogarithmic or even quadratic overhead) the case of space complexity is much cleaner: all standard models (e.g., single tape, multitape, and random access Turing machines as well as register-based random access machines) can be mutually simulated with only constant-factor space overhead [6, 13, 15]. Correspondingly, we do not worry about the exact computational model underlying our algorithms.

As is typical for sublinear space algorithms, we assume a read-only input tape and write-only output tape are provided to the machine in addition to its bounded workspace.

Circuits. Formulations of the circuit evaluation/circuit value problem differ. However, all that we are aware of reduce trivially to the FullCircuitEval formulation below. The following lemma justifies working with the simplified CircuitEval.

▶ **Definition 11** (Circuit Evaluation). *FullCircuitEval instances take the form (x, C) where x is an m-bit string and C is an s-gate circuit in the full binary basis on m inputs, provided as a list of triples of the form (ℓ_i, r_i, φ_i) where $\varphi_i : \{0,1\}^2 \to \{0,1\}$ is a binary function and each of ℓ_i and r_i is either a gate index $j \neq i$ or a variable index k for some $k \leq m$.*

The objective is to output the value v_i of each gate. For simplicity, assume $s \geq m$.

A CircuitEval instance C is an s-gate circuit in the full binary basis provided as a list of triples of the form (ℓ_i, r_i, φ_i) where $\varphi_i : \{0,1\}^2 \to \{0,1\}$ is a binary function and each of ℓ_i and r_i are either a gate index $j < i$ (i.e., the input is in topologically sorted order) or a constant bit.

The objective is to output the value of the last gate.

▶ **Lemma 12** (CircuitEval is good enough). *A space S algorithm for CircuitEval on circuits of size s implies a space $S + \log^2 s$ algorithm for FullCircuitEval.*

Proof. First, observe that given an instance of FullCircuitEval, we can replace every reference to variable k with the value x_k in space $\log s$ by simply scanning over the gates of C and for each one, if ℓ_i is a gate index, outputting it as is, and if it is a variable index k then storing the current gate in $\log s$ bits and the variable index in $\log m \leq \log s$ bits and scanning back to find x_k, outputting that value instead of k. Similarly for r_i.

Next, observe that it is possible to topologically sort a graph in $O(\log^2 s)$ space [5].

Finally, suppose M computes CircuitEval in space S. We use space-efficient composition and the above procedures to evaluate the values v_i bit-by-bit: run M on gates $\{0, \ldots, i\}$ for each i, sorting with respect to the sink i, then write v_i to the output tape. Between computations we only need to store which gate we are on using $\log s$ bits and everything else can be reused. ◀

Graphs. When we refer to a quotient graph with respect to a partition, we refer to a graph which collapses together all vertices inside a part of the partition and then removes all duplicate edges that result. That is, if $G = (V, E)$ is a graph and $P : V \to [k]$ is a partition into k parts, we say that for $i, j \in [k]$, parts i, j are connected in G/P if there exist $u, v \in V$ such that $P(u) = i$, $P(v) = j$, and $(u, v) \in E$. There is at most one edge between any pair of parts and we will discard self-loops. We will also use the term "block" to refer to parts of the partition.

Given a graph $G = (V, E)$, an "edge subdivision at e" is the operation of taking an edge $e = (u, v) \in E$ and removing it, adding in its place a vertex v_e and edges (u, v_e) and (v_e, v). The subdivided graph then has $V' = V \cup \{v_e\}$ and $E' = E \cup \{(u, v_e), (v_e, v)\} \setminus \{e\}$. A subdivision G' of G is any graph created by 0 or more edge subdivisions.

For the purposes of the complexity of operations, we restrict the representations of objects: partitions are specified as an explicit function mapping vertices to the labels of their part and the set of labels of vertices in a subdivided graph is a superset of the labels in the original graph. This allows us to maintain the gate information of the circuit and to identify all subdivision vertices to assign them identity gates.

Tree evaluation. Formally, the tree evaluation problem is defined as follows:

▶ **Definition 13** (Tree Evaluation). *TreeEval instances are full d-ary trees of height h where each leaf $\ell \in \{0, \ldots, d-1\}^h$ is labeled with a b-bit value v_ℓ and each internal node $u \in \{0, \ldots, d-1\}^{<h}$ is labeled with an explicit function (provided as a table of values) $f_u : \{0,1\}^{db} \to \{0,1\}^b$.*

We recursively define the value v_u at each internal node u in the natural way to be

$$v_u := f_u(v_{u0}, \ldots, v_{u(d-1)}).$$

The objective is to evaluate $v_{\text{root}} \in \{0,1\}^b$, the value of the root.

(For the history and broader significance of the tree evaluation problem, see Cook and Mertz work [4]. We use the formulation of the problem exposited by Goldreich [7].)

▶ **Theorem 14** (Cook-Mertz [4, 7]). *TreeEval instances of height h, arity d, and value-size b can be evaluated in space $O(h \log db + db)$.*

3 Graph partitioning

First, let us state the lemma in its general form:

▶ **Lemma 15** (Low-degree DAG partition). *For any directed acyclic graph G of size s and maximum in-degree $d > 1$, for any integer choice of parameters $d' \in [2, d+1]$ and $b \in [d/(d'-1), s]$, there is a subdivision G' of G and a partition P of G' such that:*

- *Every part in P has size at most b.*
- *Every vertex in the quotient graph G'/P has in-degree at most d'.*

- *G'/P is a layered DAG with layer count at most $\frac{d}{d'-1} \cdot \frac{s}{b}$.*
- *Given G in topologically sorted order, d', and b, the subdivision G' and partition P can be computed in $O(\log s)$ space.*

▶ Remark 16. By allowing an additional "vertex subdivision"[5] operation, one can obtain a slightly stronger version of the lemma, obtaining optimal depth s/b when $d = d' = 2$. However, the factor-of-2 savings on depth do not change the bottom line result (Theorem 2), the construction is somewhat more convoluted, and it is not immediately clear how it generalizes to arbitrary d and d'.

We also believe it should be possible to improve the $\frac{d}{d'-1}$ factor to either $\frac{d-1}{d'-1}$ or $\frac{d}{d'}$ without the introduction of a new operation, but this seems to significantly increase the complexity of the proof. This constant-factor improvement would similarly not change the bottom line result.

▶ Remark 17. Theorem 37 in Mertz's thesis [10] provides a way to control arity of TreeEval instances directly. Discussion of the relevant differences appears after the following proof.

We will first describe the construction, then prove that it has the necessary properties. As in the statement of the lemma, let G be a size-s DAG of max in-degree d and let $d' \in [2, d+1]$, $b \in [d/(d'-1), s]$ be parameters.

Symbols and indices. Throughout the proof: s refers to the size of the input graph G; d is the max in-degree of G; d' is the target in-degree of the quotient graph; b is the maximum size of blocks and the value-size in the tree evaluation instance; b_0 refers to the size of the initial blocks; i and j refer to indices of blocks or directly comparable values; ℓ is used to index incoming edges in the quotient graph; finally t refers to the index of the final initial block and is the depth of the graph.

3.1 Construction

Initial partition. Though ultimately we produce a subdivision and a partition of that subdivision, we start with a partition of the original graph. We will use this initial partition to describe the subdivision and ultimate partition. In particular, the mappings of the vertices in G in the final partition will be the same as the initial partition: we will simultaneously add subdivision vertices and blocks to contain them.

While blocks in the ultimate partition are allowed size up to b, the initial blocks have to be somewhat smaller: specifically, they will be size $b_0 = \left\lfloor \frac{d'-1}{d} \cdot b \right\rfloor$. Let $V = (v_0, \ldots, v_{s-1})$ be a topological ordering of G and let $t = \lceil s/b_0 \rceil - 1$. Then for $i < t$, we define

$$B_i := (v_{ib_0}, \ldots, v_{(i+1)b_0-1})$$

where for the final block B_t we truncate to $v_{tb_0}, \ldots, v_{s-1}$. In the notation of a mapping of vertices to parts, this is the mapping $V = \{0, 1, \ldots, s-1\} \to \{0, 1, \ldots, t\}$ given by $k \mapsto \lfloor k/b_0 \rfloor$. Note that the resulting quotient graph is a DAG and the ordering B_i is a topological sort since if $i < j$, $v_k \in B_i$ and $v_{k'} \in B_j$ then $k < k'$. Thus, there is no edge (k', k) and so there can be no edge (j, i).[6]

[5] That is, replacing a v with two vertices v_{in} and v_{out} which are connected by an edge $(v_{\text{in}}, v_{\text{out}})$ and replacing all edges (u, v) with edges (u, v_{in}) and edges (v, w) with (v_{out}, w). In the context of circuits, v_{in} is the original gate and v_{out} is an identity gate.

[6] Note that *any* partition P of G such that G/P is a DAG must have the property that each block in P represents an interval in *some* topological sort.

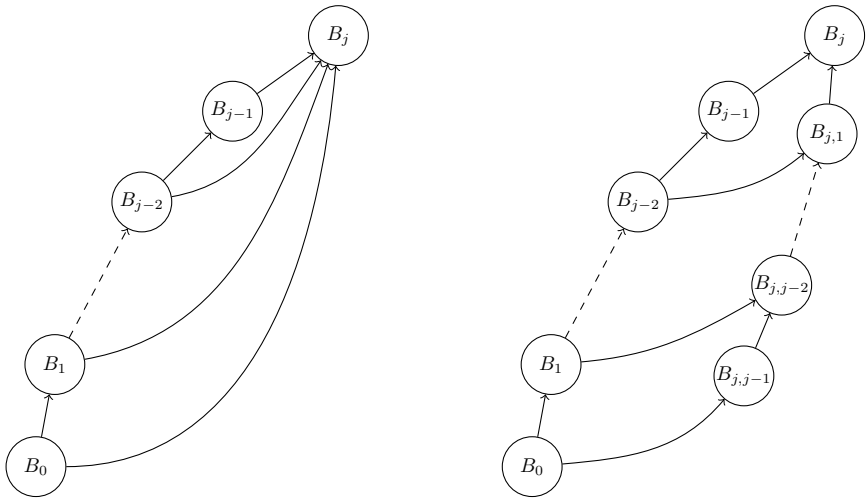

Figure 1 Addition of "cable" blocks to reduce in-degree of block B_j in the $d' = 2$ case. In the full construction, the other B_i blocks have their own cables, which are omitted here for clarity.

As in the sketch, we have successfully created a quotient DAG where the maximum depth is $t+1 = \frac{s}{b_0} = \frac{d}{d'-1} \cdot \frac{s}{b}$ and all parts have size at most b. Now we must address the key issue: the in-degree. The source vertices of those $b \cdot (d'-1)$ edges have no reason to be confined to d' blocks.[7]

Observe that we have chosen b_0 such that each block has at most $b_0 \cdot d = b \cdot (d'-1)$ incoming edges in the original graph. Intuitively, in order to manage the in-degree of the quotient graph, we will subdivide incoming edges and group the newly created vertices into $d'-1$ blocks which funnel them into the block (with the final d'th block being the immediately preceding block).

Cables. In order to resolve this, we will introduce "cables" to "gradually gather together" the incoming edges into a small number of blocks. For every edge $e = (u, v)$ with $u \in B_i$ and $v \in B_j$ for $i < j$, we will subdivide e $j - i - 1$ times, creating vertices $v_{e,k}$ for $1 \le k < j - i$. This gives us our graph G'.

Now we need to partition the newly-created vertices. For simplicity, let us first consider the case where $d' = 2$. To get the desired result: for every edge e starting outside B_j and terminating in B_j, we will add the vertex $v_{e,k}$ (if it exists) to block called $B_{j,k}$ for $k < j$. Since, in this case, there are at most $db_0 = b \cdot (d'-1) = b$ edges e terminating in B_j, the size of each block $B_{j,k}$ is at most b.

As suggested by the notation $B_{j,k}$, the blocks $B_{j,1}$ through $B_{j,j-1}$ form a sort of "tail" or "cable" connected to B_j; they run along the preceding blocks of the graph, connecting to each one at the appropriate point in the cable (specifically, block B_i is connected to the cable at block $B_{j,j-i-1}$). The cable carries all the edges terminating in block B_j from earlier blocks (see Figure 1).

[7] In fact, it is possible to construct graphs where this procedure will yield max in-degree of $\min(d' \cdot b, t-1)$ (i.e., every incoming edge leads to a different block, as long as enough blocks exist for that). As an explicit example, consider $d = 2, d' = 3$, and $s = b^2$, with edges $(k, k+1)$ for all $0 \le k < s-1$ and edges $(k, tb + k/b)$ for $k < t$ where $b|k$. That is, a graph where a "spine" of edges enforces a specific topological ordering and then the first vertex of each block B_i connects to vertex i in block B_t. Then B_t has in-degree $t - 1 \gg d'$.

Now we consider the case where $d' > 2$. The "bandwidth" of the cable is b, and that becomes insufficient when there are up to $(d'-1)b$ incoming edges. We solve this bandwidth problem in a very natural way: add more cables. Formally: in the case where $d' > 2$, we will instead have blocks $B^{\ell}_{j,k}$ where $k < j$ indexes the "layer" as before and $1 \leq \ell < d'$ indexes the cable. We can enumerate the edges $\{e_1, \ldots, e_{b\cdot(d'-1)}\}$ coming into B_j. Then $v_{e_r,k}$ will be placed into block $B^{\lfloor r/b \rfloor}_{j,k}$.

For ease of computation, we will define this enumeration of the edges to be indexed by first the index of the terminal vertex and then by which of $\leq d$ incoming edges it corresponds to (sorted by whatever order they appear in in the input representation). We will allow indices to be skipped: for example, e_d will be reserved for the dth incoming edge of vertex v_{jb}, even if the in-degree of v_{jb} is strictly less than d and that edge does not exist. This completes the description of the partition P of G'.

3.2 Analysis

Proof of Lemma 15. Recall that we want a subdivision G' and a partition P such that:

- Every part in P has size at most b.
- Every vertex in the quotient graph G'/P has in-degree at most d'.
- G'/P is a strictly layered DAG with depth (i.e., layer count) at most $\frac{d}{d'-1} \cdot \frac{s}{b}$.
- Given G in topologically sorted order, d', and b, the subdivision G' and partition P can be computed in $O(\log s)$ space.

We show that the above construction satisfies these properties.

Size The first condition is trivially satisfied for the initial blocks which have size $b_0 = \frac{d'-1}{d} b \leq b$. It is satisfied for the cable blocks since each cable block gets at most b gates assigned to it because the map $[b \cdot (d'-1)] \to [d'-1]$ given by $r \mapsto \lfloor r/b \rfloor$ is b-to-1.

In-degree The second condition is satisfied for the cable blocks since each $B^{\ell}_{j,k}$ has incoming edges only from $B^{\ell}_{j,k+1}$ and B_{j-k-1} and is satisfied by construction for the initial blocks since B_j has incoming edges only from $B^{\ell}_{j,1}$ for $\ell \in [d'-1]$ and from B_{j-1}.

Layering The third condition can be seen similarly to the second: there are $t+1$ layers $L_0, \ldots, L_t$ where B_j belongs to L_j and $B^{\ell}_{j,k}$ belongs to L_{j-k}. Since each block $B_j \in L_j$ has incoming edges from only $B^{\ell}_{j,1} \in L_{j-1}$ and $B_{j-1} \in L_{j-1}$ and each block $B^{\ell}_{j,k} \in L_{j-k}$ connects only to $B_{j-k-1} \in L_{j-k-1}$ and $B^{\ell}_{j,k+1} \in L_{j-k-1}$. Thus, blocks in L_i connect only to blocks in L_{i+1}, and so the partition $\{L_0, \ldots, L_t\}$ represents a "strict layering." In particular, this trivially implies that the maximum directed path length is at most $t = \frac{s}{b_0} - 1 = \frac{d}{d'-1} \cdot \frac{s}{b} - 1$.

Complexity First, note that the parameter b_0 can be computed from b, d, and d'. Since d is not given, it needs to be computed from the graph G but this can be done by iterating over all vertices and keeping a running max of their in-degrees (which in turn can be computed by iterating over all other vertices and counting how many have an edge to the given vertex).

The computation of the subdivision is arithmetic: for each edge $e = (v_m, v_n)$ we compute $i = \lfloor m/b_0 \rfloor$ and $j = \lfloor n/b_0 \rfloor$ and then subdivide that edge $j - i - 1$ times. Since this is all arithmetic on indices, it is logspace computable.

The computation of the partition is similarly efficient: v_m is assigned to $B_{\lfloor m/b_0 \rfloor}$ and $v_{e,k}$ for $e = (v_m, v_n)$ is assigned to $B^{\ell}_{j,k}$ where $\ell = \lfloor r/b \rfloor$ and $r = d \cdot (n \mod b_0) + p$ where p is the index of e among the incoming edges of v_n (which is logspace computable since we define the ordering on the edges for a given sink vertex to be whatever order they appear in in the input representation).

When outputting the subdivision, we can trivially output all vertices of the original graph first followed by the subdivision vertices, satisfying the condition that labels of vertices be preserved. ◀

▶ Remark 18. Mertz's result [10] allows for controlling the arity of a TreeEval instance by separating layers in the instance. One might hope to stop at G/P, expand this to a TreeEval instance, and then do something similar. However, because the nodes of a TreeEval instance are arbitrary functions, this approach to decreasing the arity requires increasing the value-size proportionally to the decrease in arity – that is, the product of value-size and arity which governs the space usage of the Cook-Mertz algorithm remains unchanged. (Working with arbitrary TreeEval instances also requires a logarithmic increase in depth to decrease the arity.)

In our use case of $d = d' = 2$, G/P would become a TreeEval instance of height $\Theta(\frac{s}{b})$, value-size b, and arity $\Theta(b)$. Attempting to reduce the arity to $d = 2$ at this point would require increasing the value-size to $\Omega(b^2)$ and the height to $\Omega(\frac{s}{b} \cdot \log b)$.

By decreasing the in-degree before we "forget" that the TreeEval instance came from the quotient of a DAG with small in-degree, we are able to leverage the additional structure to reduce arity from $\Theta(b)$ to 2 without significant penalty in value-size or height.

4 Circuit evaluation reduction

Our reduction to tree evaluation can be stated as follows:

▶ **Theorem 19.** *Given a topologically sorted circuit of size s and any $b \in [2, s]$, we can produce an equivalent TreeEval instance*[8] *with arity $d = 2$, height $h = O(\frac{s}{b})$ and word-size b as chosen. Further, this reduction is locally computable in space $O(b + \log s)$.*

That is, there is a space $O(b + \log s)$ procedure which, on input (C, b, u, x, y) (where C is a circuit, $b \in [2, s]$, $u \in \{0,1\}^{\leq h}$ specifies a node in the tree, $x, y \in \{0,1\}^b$) computes $f_u(x, y)$ where f_u is from the TreeEval instance equivalent to C.

In particular, choosing $b = \sqrt{s \log s}$ and applying Theorem 14 immediately gives the first half of Theorem 2; further, a space $O(h + b)$ procedure for TreeEval would immediately imply a space $O(\sqrt{s})$ procedure for CircuitEval by choosing $b = \sqrt{s}$, giving the second half of Theorem 2.

▶ Remark 20. While Stoeckl showed that TreeEval can be solved in space $o(\log n \cdot \log \log n)$ (Theorem 21), the specific parameters do not give us a better result than $\sqrt{s \log s}$. To improve upon it, we must have $h + b = o(\sqrt{s \log s})$. Note that $h + b = \Omega(\sqrt{s})$ unconditionally. Then:

$h + b = O(\sqrt{s \log s})$ implies $b = \Omega\left(\sqrt{s \log^{-1} s}\right)$ and so $\log b = \Omega(\log s - \log \log s) = \Omega(\log s)$. Similarly, we must have $b = O(\sqrt{s \log s})$ so $\log b = O(\log s)$. Thus,

$$(h + b) \cdot \frac{\log b}{\log \log b} = \Omega(\sqrt{s}) \cdot \frac{\Omega(\log s)}{O(\log \log s)} = \Omega(\sqrt{s \log s}).$$

▶ **Theorem 21** (Stoeckl, cf. [8])**.** *There exists an algorithm for evaluating binary TreeEval instances of height h and value-size b which runs in space $\Theta\left((h + b) \cdot \frac{\log b}{\log \log b}\right)$.*

[8] Technically, the TreeEval instance outputs b bits while the circuit outputs 1 bit; we actually produce a TreeEval instance where the output of the circuit is bit $s \bmod b_0$ of the output for an efficiently computable b_0.

Proof of Theorem 19. At a high level, we apply the DAG Partition Lemma (vertices introduced by subdivision become identity gates) to C and expand C'/P into a tree, letting the function $f_B(x, y)$ computed at a node $B \in C'/P$ be simply the subcircuit B evaluated with the external input wires taken from x, y. Note that B may be smaller than size b (they can even be size 0) and may have fewer than $2b$ external input wires. We can embed each part into a value by saying that the values of gates fill the bits in the $\{0,1\}^b$ values in topologically sorted order.

To ensure that we get a full binary tree, we will always make nominal connections $B_{j-1} \to B_j$, $B_{j,1} \to B_j$, and $B_{j,i} \to B_{j,i-1}$. If there are no wires between any of those pairs of blocks in the original circuit then f_B will simply ignore those inputs. To ensure the blocks $B_{j,j-1}$ are binary, we will add empty $B_{j,j}$ blocks.

Further, the part of computing $f_B(x, y)$ corresponding to evaluating the subcircuit B can be done by naively evaluating each gate in topological order from previous ones and writing it down, yielding the $O(b)$ component of the space.

It turns out that there are no pitfalls, and the above description simply works. Nonetheless, it is stated somewhat more rigorously below for completeness.

Proof. First, note that we can efficiently identify the children of a given block (e.g., by iterating over all blocks and checking whether they are connected, which can in turn be done by iterating over all pairs of vertices in the two blocks being considered). We can identify the root block by finding which block the vertex v_{s-1} maps to. We will consider the "left" child of a block to be the child block whose label is lexicographically first and the "right" child to be the one whose label is lexicographically second.

Given as input a circuit C in the full binary basis, the parameter b, and (u, x, y), using space-efficient composition, we:

1. Apply the DAG Partition Lemma with b as given and $d' = 2$. Initialize $B \leftarrow \mathsf{root}$.
2. For each bit u_i of u, identify the left or right child of B based on whether $u_i = 0, 1$ and update B to that child.
3. Once we have found the block B corresponding to the tree node u, we must evaluate it. If it is a leaf block, it must have no inputs, so we compute it by naive dynamic programming.[9]

   Otherwise, since we defined values to fill the b-bit values in topologically sorted order, for each input $v \in B_c$ (for $c \in \{x, y\}$) to a gate $w \in B$, we iterate over G' and count how many gates we find in B_c before reaching v (call it ℓ); we then take bit c_ℓ as the corresponding input to w. This allows us to locally compute the circuit B with external inputs substituted in.
4. Having computed (or rather, expressed a procedure by which we can space-efficiently compute bits of) the circuit B and which bit of x, y each input to B corresponds to, we compute the values of B via naive dynamic programming.

In order to extract the value of the circuit from the reduction, we then ultimately take bit $s \mod b_0$ of v_{root}, since that is the location of the last gate in the final block.

The correctness of the reduction is straightforward: the behavior of the circuit C' is the same as C since we just inserted some identity gates into wires. Further, the value of a block $B \in C'/P$ can be computed using the values of all the gates some gate in B takes input from, which by construction are just the 2 children of B.

[9] The space complexity of the reduction could be improved to $O(\sqrt{b \log b} + \log s)$ by recursing, but this would not improve the bottom line space complexity of circuit evaluation since the space is already dominated by the evaluation of the tree evaluation instance.

For complexity, observe that on-demand computing bits of G' and P as needed takes space $O(\log s)$; step 2 requires holding a label B and index i into u, which also takes space $O(\log s)$; iterating over G' in step 3 requires a single log-sized counter and counting gates in B_c similarly requires logspace since we just compute for each gate which block it belongs to (which we are given explicitly) and maintain a counter of how many have been in B_c; step 4 is done via naive dynamic programming, which requires space $O(b)$ to evaluate a size-b circuit.

Finally, observe that using space-efficient composition routines, we can then compose this with the Cook-Mertz algorithm using $O(b+h)$ bits of overhead. Naively: we can iterate over all (u, x, y) to print the whole tree, which is $O(2^h \cdot 2^{2b})$ in size, and then use standard space-efficient composition techniques which use log-sized indices of size $O(b+h)$ to keep track of head locations on synthetic read/write tapes.

We can do much better[10] but any TreeEval algorithm will necessarily use at least $\Omega(b+h)$ space to index its input, so this is good enough. ◀

References

1. Markus Bläser, Valentine Kabanets, Ronen Shaltiel, and Jacobo Torán. Algebraic and Analytic Methods in Computational Complexity (Dagstuhl Seminar 22371). *Dagstuhl Reports*, 12(9):41–59, 2023. doi:10.4230/DagRep.12.9.41.
2. James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 752–760, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3357713.3384316.
3. James Cook and Ian Mertz. Encodings and the tree evaluation problem. *Electron. Colloquium Comput. Complex.*, TR21, 2021. URL: https://eccc.weizmann.ac.il/report/2021/054/.
4. James Cook and Ian Mertz. Tree evaluation is in space $O(\log n \cdot \log \log n)$. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, pages 1268–1278, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649664.
5. Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1):2–22, 1985. International Conference on Foundations of Computation Theory. doi:10.1016/S0019-9958(85)80041-3.
6. Peter van EMDE BOAS. Chapter 1 - machine models and simulations. In JAN VAN LEEUWEN, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 1–66. Elsevier, Amsterdam, 1990. doi:10.1016/B978-0-444-88071-0.50006-0.
7. Oded Goldreich. On the Cook-Mertz tree evaluation procedure. *Electron. Colloquium Comput. Complex.*, TR24, 2024. URL: https://api.semanticscholar.org/CorpusID:271770168.
8. Oded Goldreich. *Computational Complexity and Local Algorithms : On the Interplay Between Randomness and Computation*, chapter Solving Tree Evaluation in $o(\log n \cdot \log \log n)$ Space, pages 113–118. Springer Nature Switzerland, Cham, 2025. doi:10.1007/978-3-031-88946-2_7.
9. John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *J. ACM*, 24(2):332–337, 1977. doi:10.1145/322003.322015.
10. Ian Mertz. *The Complexity of Composition: New Approaches to Depth and Space*. PhD thesis, University of Toronto, 2022.
11. Nicholas Pippenger. *Fast simulation of combinational logic networks by machines without random-access storage*. IBM Thomas J. Watson Research Division, 1977.

[10] This algorithm already makes local computations and so we simply need to convert a given tape index i into u, x, y via arithmetic. We can further improve it by observing that the Cook-Mertz algorithm already implicitly divides its seeks into u, x, y.

12 Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, April 1979. doi:10.1145/322123.322138.

13 Cees Slot and Peter van Emde Boas. The problem of space invariance for sequential machines. *Information and Computation*, 77(2):93–122, 1988. doi:10.1016/0890-5401(88)90052-1.

14 Manu Viola. Mathematics of the impossible. https://web.archive.org/web/20250716031147/https://www.ccs.neu.edu/home/viola/papers/moti.pdf, May 2025. [Online; accessed 2025-07-16].

15 R. Ryan Williams. Simulating time with square-root space. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, STOC '25, pages 13–23, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3717823.3718225.

A Fast circuit evaluation

Because it is not digitally available, we provide an exposition of Pippenger's algorithm below. We make no claims of originality. Note that this exposition is not intended to be rigorous so much as explanatory, so we have omitted a proof of correctness or runtime analysis.

▶ **Theorem 22** (Pippenger [11]). *Given a size s circuit C and an input x, the output $C(x)$ can be evaluated in multitape Turing machine time $O(s \log^2 s)$.*

We will describe the high-level recursive algorithm; in order to implement it on a multitape Turing machine, one simply needs to create a tape for each "local variable" and treat this tape as a stack, pushing the new value of each local variable to its stack when recursing. By, for example, using a marker to separate layers of the stack, any "well-behaved" recursive algorithm (i.e., one which does not try to read back up into the caller's stack frames) can be expressed using a constant number of tapes, since the "current level" of each stack can function as an arbitrary (one-sided) Turing machine tape.

We will work with lists of "records," which will be various tuples of indices (i.e., all entries in the list have the same size which is $O(\log s) = O(\log(\text{input length}))$). We will use the following high-level operations:

Classify Given a list of n records and a linear-time predicate, "classify" the records into two lists based on whether they satisfy the predicate. On a multitape Turing machine, we can perform this operation in time $O(n \log s)$ trivially if the source list and two destination lists are all on separate tapes.

Merge Given a linear-time comparison operation and two lists of lengths n and m records sorted according to the comparator, "merge" the two lists into one sorted list of length $n + m$. On a multitape Turing machine, we can perform this operation in time $O((n + m) \log s)$ if the source lists and destination list are all on different tapes.

Sort Given a list of n records and a linear-time comparator, sort the list of n records according to the comparator. On a multitape Turing machine, this can be done recursively with a constant number of auxiliary tapes in time $O(n \log n \log s)$.

Much like in the FullCircuitEval problem, we will assume that the input is a pair (x, C) of input and circuit, with the circuit encoded as a list of s records. However, we will assume that C is provided in topologically sorted order. We will adopt the convention that $0 < 1 < x_1 < \cdots < x_n < v_1 < \cdots < v_s$. Note that unlike in the space-bounded computation case, we cannot simply assume that the caller only wants the last bit, since rerunning the algorithm for each prefix of the circuit would make the running time quadratic. Correspondingly, the algorithm outputs the value of all gates.

The first step is to eliminate the input x and bake it into the circuit as constants: first, we copy x and C to separate tapes. In the process of copying C, we will replace each record (ℓ_i, r_i, φ_i) with the record $(\ell_i, r_i, \varphi_i, i)$ (which we can do by keeping the counter i on an auxiliary tape).

Then we sort C by the field ℓ. Then scan C and x in parallel, advancing the x head whenever ℓ increases, substituting the appropriate constants for the left inputs of each gate which takes a variable input. We can then repeat sorting by field r to eliminate the rest of the inputs and, finally, sort C by the index field to restore the original order.

At this point, we can copy C back to the original input tape and discard the contents of all other tapes.

Now, we will convert the circuit to (a variant of) Pippenger's "alternate representation" – instead of a list of gate records which each store their inputs, we will have a list of gate records, which are now just the index and gate function, together with a second list of "wire"/"value" records of the form (u, v, d) where u is either a gate index (wire) or constant (value), v is a gate index, and $d \in \{L, R, O\}$ indicates whether this wire is the left input to v, the right input, or an "output record." In the case of an output record, u will be either $*$ (wire) or a constant (value), representing either a place holder for the value of the gate or the computed value of the gate.

We can do this by simply scanning over the circuit and for each gate $v = (\ell, r, \varphi, i)$ outputting wire records (ℓ, i, L), (r, i, R), and $(*, i, O)$ and gate record (φ, i).

Given sets of gates I, J, let $W_{I\to J}$ represent the set of wires starting in I and terminating in J. Let $V_{I\to J}$ be the set of value records corresponding to those wires. In both cases let $I \to$ and $\to J$ refer to wires/values starting in I and terminating in J respectively. Let G_I denote the gate records of I.

In the following procedure, each expression like G_K or $V_{\to I}$ should be interpreted as the name of a local variable. Performing the Turing machine construction by replacing each variable with a tape interpreted as a stack of tape yields a multitape Turing machine with around 20 tapes (including a couple work tapes in addition to the stacks).

We now describe a recursive procedure which, given an interval K, $G(K)$, $V_{\to K}$, and $W_{K\to}$ computes $V_{K\to}$. If $|K| = 1$, we simply evaluate the single gate in K using the two value records in $V_{\to K}$ and then substitutes that value into each record in $W_{K\to}$. Otherwise:

- Split K into I and J, each half the size of K.
  - Classify G_K into G_I and G_J.
  - Classify $W_{K\to}$ into $W_{I\to}$ and $W_{J\to}$.
  - Classify $V_{\to K}$ into $V_{\to I}$ and $V_{\to J} \setminus V_{I\to J}$.
- Letting $K \leftarrow I$, recurse to compute $V_{I\to}$ (which we move from the output stack $V_{K\to}$ to the stack $V_{I\to}$).
- Classify $V_{I\to}$ into $V_{I\to J}$ and $V_{I\to} \setminus V_{I\to J}$.
- Merge $V_{I\to J}$ with $V_{\to J} \setminus V_{I\to J}$ to produce $V_{\to J}$.
- Letting $K \leftarrow J$, recurse to compute $V_{J\to}$ (which we again move to stack $V_{J\to}$).
- Merge $V_{I\to}$ and $V_{J\to}$ into $V_{K\to}$

In order to compute our final output, we ensure $V_{K\to}$ is sorted by output gate, and then scan over it and extract the bits b from the records of the form (b, v, O).