# PSPACE Survives Three-Bit Bottlenecks

Jin-yi Cai and Merrick L. Furst
YALEU/DCS/TR-528
March 1987

# PSPACE Survives Three-Bit Bottlenecks

Jin-Yi Cai* Merrick L. Furst†

**Abstract**

**Barrington recently discovered that polynomial-size logarithmic-depth circuits are equivalent to constant-width branching programs [B]. We discuss the relationship between constant-width branching programs and a hierarchy of languages that lies between P and PSPACE. We also introduce the notion of logspace-serializability. For relativized computations, this notion corresponds to logspace-uniform, constant-width branching programs. Applying Barrington's method we show that PSPACE is logspace-serializable.**

## 1 Introduction

Constant-depth circuits have come to play an important role in clarifying the structure and nature of complexity classes. The fact that circuits of constant-depth and polynomial size cannot compute functions such as parity or majority [FSS] means that $AC^0$, the set of langauges acceptable by constant-depth, polynomial-size circuits, is smaller than some of the other complexity classes below Logspace. Just as the hypothesis that PTIME is less powerful than NPTIME leads to a natural notion of Polynomial-time reducibility between NP problems, the fact that $AC^0$ is definitely weaker than Logspace leads to a natural notion of $AC^0$ reducibility between problems much lower in the complexity picture. Exactly what the structure and relationships are among these low-level classes under $AC^0$ reductions remains to be determined.

Constant-depth circuits play a role in the study of the structure of the much richer Meyer-Stockmeyer polynomial-time hierarchy which may lie properly between P and PSPACE. The definition for constant-depth circuits was originally motivated by the relationship bewteen depth-$k$ circuits and levels of the polynomial-hierarchy.

A language $L$ is in the $i^{\text{th}}$ level of the Meyer-Stockmeyer hierarchy if and only if the membership predicate "$x \in L$" can be described by a sentence of the form

$$\exists^p y_1 \forall^p y_2 \cdots Q_i^p y_i R(x, y_1, \ldots, y_i)$$

*Department of Computer Science, Yale University, New Haven CT 06520

†Department of Computer Science, Carnegie-Mellon, Pittsburgh, Pa. 15213

in which there are $i$ alternations of quantifiers ($Q_i$ is $\forall$ or $\exists$ as $i$ is even or odd), each of which quantifies over values whose lengths are polynomially bounded, and in which $R$ is a polynomial-time predicate. A further refinement of the hierarchy appears if we distinguish between sentences that begin existentially, ($\Sigma_i$), or universally, ($\Pi_i$). The class P is $\Sigma_0 = \Pi_0$, the class NP is $\Sigma_1$, and co-NP is $\Pi_1$. Whether any of the levels of the polynomial-hierarchy are distinct is unknown and tantamount to whether P is different from PSPACE.

The polynomial-hierarchy may be constructed relative to an oracle set $A$ in a natural way. One noramlly thinks about computation relative to an oracle $A$ as computation that is, perhaps, *enhanced* by the ability to answer questions about membership in $A$. It also makes sense to turn this computational image around. Relativized computation on an input $x$ may be thought of as a question (posed as $x$) that is intended to determine a property of the oracle. That is, the purpose of the computation on a given input can be thought of not as determining a property of the input string, but rather as determining a property of the oracle set. For example, on input $1^n$ a machine could be asked to accept or reject depending upon the parity of the number of length $n$ strings in the oracle set. By turning the picture around in this way, the problem of separating language classes by oracles becomes the problem of determining which predicates of oracle sets can be computed.

In addition to showing that parity is not computable by polnomial-size, constant-depth circuits, Furst, Saxe and Sipser [FSS] proved that any predicate of an oracle $A$ that is in the $i^{\text{th}}$ level of the relativized polynomial-hierarchy, $\Sigma_i^A$, can be computed by circuits of depth $i+1$ and size $n^{poly\log n}$. This was the original motivation for constant-depth circuit lower bounds. A Turing machine running in PSPACE and having access to an oracle set $A$ can certainly determine if an even or odd number of the strings of length $n$ are in the set $A$. By proving that parity cannot be computed by circuits of constant-depth and less than almost-exponential size, Yao [Y] established the existence of an oracle $A$ such that PSPACE$^A$ is different than the polynomial-hierarchy relative to $A$. Cai showed that not only do these circuits fail to compute the parity function, but they err almost 50% of the time, thus proving that almost every oracle $A$ separates PSPACE$^A$ from the Polynomial-hierarchy relative to $A$ [C].

Constant-width branching programs (the definition is omitted here) are a slightly more powerful model of computation than are constant-depth circuits. Branching programs are provably more powerful since a width-2, length-$n$ branching program can already compute parity. Barrington recently proved a surprising result regarding constant width branching programs. He showed that the class of width-5, polynomial-size branching programs is equivalent to nonuniform $NC^1$ circuits.

His method is a key ingredient of our proof and will be outlined later.

It turns out to be natural to ask the following question: "Is there a hierarchy that corresponds to constant-width branching programs as the polynomial-hiearchy corresponds to constant-depth circuits?" The $i^{\text{th}}$ level of the polynomial hierarchy corresponds to depth-$(i+1)$ circuits. Is there anything similar between P and PSPACE that relates to constant-width branching programs?

There is a hierarchy of languages between P and PSPACE, the levels of which correspond to functions computable by constant-width branching programs. This hierarchy is almost certainly distinct at its first few levels. It then collapses.

## 2 Safe-Storage, or Bottleneck, Machines

Consider a Turing machine with a read-only input tape and a read/write worktape. Provide the machine with two additional devices: a read-only, polynomially-long tape used as a binary counter (a clock), and a device called the "safe store" that can only be in a constant, $i$, number of states.

Define a computation of such a "safe-storage" machine as follows. With the input on the input tape, the "safe store" in a designated initial state, and the counter tape set to all zeroes, begin phase 0. In phase $t$ the machine may perform any reasonable computation; however, after a polynomial number of steps it is obligated to terminate phase $t$ by going to a distinguished state at which point the worktape is erased, all tape heads are reset to their original positions, the counter is incremented by 1 and the computation proceeds from the start state. All that is left of the computations in the previous phases is the state of "safe-store," and the counter indicating which phase the machine is in. An input string is accepted if and only if the machine finishes the last phase (counter tape having all 1's) with the safe-storage tape in a particular state. (In the special case of a safe-store that has only 1 state, acceptance is defined by having the Turing machine enter an accepting state.)

A safe-storage computation can be simulated in PSPACE. Intuitively, safe-storage computations should be different from PSPACE since every so often the computation is squeezed into a "bottleneck" through which very little information about the past is passed.

There is a natural hierarchy of languages lying between P and PSPACE which is derived from safe-storage machines. The $i^{\text{th}}$ level of this safe-store hierarchy contains all languages $L$ that can be recognized by a safe-storage machine having only $i$ safe-store states. That is, $\text{SF}_i$ is the union, over all polynomial-time bounded Turing machines and over all polynomial lengths for the counter,

of languages accepted by safe-store machines having $i$ states of safe-storage.

**Theorem 2.1**

$$SF_1 = P, \text{ and}$$

$$SF_2 \supseteq \Sigma_1 \cup \Pi_1$$

**proof:** The acceptance behavior of a machine with 1 safe-storage state is completely determined by what happens in the final polynomial-time phase. Thus, $SF_1$ is the same as P. To see that $SF_2$ contains $\Sigma_1$, observe that one can use the counter to quantify over all polynomially-long strings and that 2 states of the safe-store are enough to keep track if at least one of the strings satisfies a polynomial-time predicate. Similarly, $SF_2$ contains $\Pi_1$. **QED**

It seems likely, however, that $SF_2$ is more powerful than either of the classes at the first level of the polynomial-hierarchy. For example, the set of Boolean formulas that have an even number of satisfying assignments is in $SF_2$. Moreover, it can be shown that 2 states of safe-storage are enough to accept all languages in the Boolean hierarchy [CH].

# 3 Relativized Safe-Store Machines and CWBP

We save the proof for a final version of the paper, but point out the important theorem that shows the parallel between the safe-storage hierarchy and branching programs on the one hand, and the polynomial hierarchy and constant-depth circuits on the other.

**Theorem 3.1** *If a predicate $T$ of an oracle $A$ can be computed by a safe-storage machine with $i$ states of safe-storage, then the predicate $T$ can be computed by a width-$i$, length $n^{\text{poly}\log n}$ branching program.*

An almost-exponential lower bound on the length of width-3 branching programs computing majority would imply relativized separation results.

# 4 Serializability and PSPACE

One of the earliest and best known theorems in complexity theory is the space hierarchy theorem due to Hartmanis, Lewis and Stearns [HLS]:

follows:

Let the nodes be ID's. List all the possible ID's of $M$ for a fixed time step $t$ ( $0 \leq t < 2^{p(n)}$ ) in a column; assume conveniently that the content of each ID written in binary is used as the index within the column. Syntactically incorrect codes denote dead states. Place all the columns horizontally in increasing order of $t$ (say from left to right.) Denote by $\mathrm{ID}_{it}$ the $i^{\mathrm{th}}$ ID in the $t^{\mathrm{th}}$ column. Place an edge from $\mathrm{ID}_{it}$ to $\mathrm{ID}_{j(t+1)}$ iff $M$ would go in one step from the first configuration to the second.

Fix an input $x$ of length $n$. Let $ID^0$ be the initial configuration of the machine on $x$. Let $ID^Y$ be the unique accepting (Yes) configuration. Clearly $M$ accepts $x$ *iff* there is a path from $ID^0$ to $ID^Y$.

From this formulation it is possible to construct a polynomial $p(|x|)$-deep, $O(2^{p(|x|)})$-size circuit that outputs 1 if and only if $M$ accepts $x$. Define PATH(ID,ID$'$, $k$) to be true if and only if there is a path in the above graph from ID to ID$'$ of length exactly $2^k$. Using the standard recursive observation about the PATH predicate, we see how to express PATH($\mathrm{ID}^0$,$\mathrm{ID}^Y$,$p(|x|)$) with a $p(|x|)$-deep, $O(2^{p(|x|)})$-size Boolean circuit whose leaves are simple predicates about single steps of Turing machine $M$.

Now we will apply Barrington's idea [B] to derive a representation of the statement "$M$ accepts $x$" as a product of permutation variables

$$\sigma_{R(1)}\sigma_{R(2)}\cdots\sigma_{R(2^{q(|x|)})}$$

A permutation variable $\sigma_R$ has the property that depending upon the value of the 0-1 predicate $R$, it evaluates to either the identity permutation or some 5-cycle of $S_5$, the symmetric group on 5 letters.

The product should evaluate to the identity permutation if $M$ accepts $x$, and the cycle (12345) if $M$ rejects $x$.

Barrington's clever idea was to use the commutator in $S_5$ to represent logical OR:

Let

$$c = (12345), \alpha = (15324), \beta = (12534).$$

Then, multiplied from left to right, $\alpha\beta\alpha^{-1}\beta^{-1} = c$.

Let the permutation variables $\sigma_s$ and $\tau_t$ represent the truth of two statements $s$ and $t$, such that $\sigma_s$ is the identity permutation $e$ if $s$ is true, and the 5-cycle $\alpha$ otherwise, and similarly, let $\tau_t$

**Theorem 4.1** *(HLS) If $S_2(n)$ is a fully space-constructible function,*

$$\lim_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0,$$

*and $S_1(n)$ and $S_2(n)$ are each at least $log(n)$, then there is a language in $DSPACE[S_2(n)]$ not in $DSPACE[S_1(n)]$.*

This theorem implies there is a dense hierarchy in deterministic space bounded computations.

An apparent weakening of the power of safe-storage machines is described in the following definition.

**Definition 4.2** *A language $L$ is logspace-serializable if there is a safe-storage machine accepting $L$ in which each phase's computation is Logspace. A class $\mathcal{L}$ of languages is logspace-serializable if every $L \in \mathcal{L}$ is logspace-serializable.*

The relationship between logspace-serializabilty and constant-width branching programs shows up when safe-storage computations are made relative to an oracle $A$. We omit the details, but point out that if a property $R$ of an oracle $A$ is recognizable by a safe-storage machine whose phases are Logspace computations, then $R$ can be computed by logspace-uniform, constant-width branching programs. Here logspace uniform means that given the number of transition $i$ it is possible to calculate the transition in space $O(\log i)$.

What problems are *logspace*-serializable? The set of Quantified Boolean Formulas ($QBF$) that provably requires more than $log(n)$ space will need that extra amount of space at some point in its computation. In light of the space hierarchy theorem, one might not expect such problems to be logspace serializable.

However, we prove the contrary:

**Theorem 4.3** *$PSPACE$ is logspace serializable.*

**proof** Let $L = L(M)$, where $M$ is a polynomial-space-bounded deterministic TM. Assume, without loss of generality, that $M$ is clocked by a polynomial $p$, such that the computation of $M$ on input $x$ takes precisely $2^{p(|x|)}$ steps. Also assume, without loss of generalitry, that when $M$ accepts an input, it accepts it by entering a unique configuration when the clock times out.

Consider the computation of $M$ on inputs of length $n$. An instantaneous description (ID) for $M$ consists of a polynomial $q(n)$ bits of information about state and tape [HU]. Define a graph as

be the identity or $\beta$. Then

$$\sigma_s \tau_t \sigma_s{}^{-1} \tau_t{}^{-1}$$

represents the truth of $s \vee t$. It is the identity $e$ if $s \vee t$ is true, and the cycle $c$ otherwise.

To represent logical NOT, set $\sigma_s$ to be $e$ if $s$ is true and $c^{-1} = (15432)$ if $s$ is false. Then the product $\sigma_s \cdot c$ represents $\neg s$. Logical AND is handled via OR and NOT.

Thus, the exponentially-large, polynomially-deep circuit for "$M$ accepts $x$" can be converted in a straightforward way to an exponentially-long product of permutation variables. Now the task is to show that the particular permutation variable that appears at position $i$ in this product can be computed in $O(\log i)$ space. The details follow.

In general, we say a permutation variable $\sigma$, $\gamma$-represents $s$, if the following conditions are satisfied: $\sigma = e$ if $s$ is true and $\sigma = \gamma$ if $s$ is false, where $\gamma$ is some fixed 5-cycle in $S_5$.

For any fixed 5-cycle $\gamma = (1\ 1\gamma\ 1\gamma^2\ 1\gamma^3\ 1\gamma^4)$, define

$$\delta = \delta(\gamma) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 1\gamma & 1\gamma^2 & 1\gamma^3 & 1\gamma^4 \end{pmatrix},$$

and

$$\begin{aligned} \alpha(\gamma) &= \delta(\gamma)^{-1} \alpha \delta(\gamma), \\ \beta(\gamma) &= \delta(\gamma)^{-1} \beta \delta(\gamma). \end{aligned}$$

Clearly

$$\alpha(\gamma)\beta(\gamma)\alpha(\gamma)^{-1}\beta(\gamma)^{-1} = \gamma.$$

It is straightforward to generalize the previous discussion on OR and NOT in terms of $\gamma$-representation for any fixed $\gamma$.

We use a local variable *pad* to accumulate the "boundary effect" encountered in the context switching from AND to OR. The following function Alpha-Beta takes as inputs a 5-cycle $\gamma$ and two bits $b_1 b_2$, and returns a 5-cycle $\gamma$ and *pad*.

**procedure** Alpha-Beta ( $\gamma$, $b_1 b_2$, *pad* )
**begin**
case 1: $b_1 b_2 = 00 \rightarrow \gamma = \alpha(\gamma)$
case 2: $b_1 b_2 = 01 \rightarrow \gamma = \beta(\gamma)$
case 3: $b_1 b_2 = 10 \rightarrow \gamma = \alpha(\gamma)^{-1}$

case 4: $b_1b_2 = 11 \rightarrow \gamma = \beta(\gamma)^{-1}$

if $b_1b_2 \neq 11$ then $pad = e$

**end**

Alpha-Beta is obviously a finitary function and thus can be stored in a table.

The following procedure uses the function Alpha-Beta and $get2bits(l)$ which returns the next 2 leading bits of its argument $l$. Also it uses a precedure *getnewid* to update $ID_1$ and $ID_2$; we will explain that shortly. Procedure Mission ( $\gamma$, $l$, $ID$, $ID'$, $pad$ ) takes its input $l$, a binary integer of $2p(n)(q(n)+1)$ bits, and returns a 5-cycle $\gamma$, two instantaneous descriptions $ID^1$, $ID^2$, and $pad$.

**procedure** Mission ( $\gamma$, $l$, $ID_1$, $ID_2$, $pad$ )
**begin**
$\gamma = (12345)$, $ID_1 = ID^0$, $ID_2 = ID^Y$
$pad = e$
For $t = 1$ to $p(n)$ do
For $i = 1$ to $q(n)$ do
$b_1b_2 = get2bits(l)$
Alpha-Beta ( $\gamma$, $b_1b_2$, $pad$ )
$pad = \gamma \cdot pad$
$\gamma = \gamma^{-1}$
$b_1b_2 = get2bits(l)$
Alpha-Beta ( $\gamma$, $b_1b_2$, $pad$ )
$pad = \gamma \cdot pad$
$\gamma = \gamma^{-1}$
$getnewid(ID_1, ID_2)$
**end**

The procedure *getnewid* is used to update the two current instantaneous descriptions for which the existence of a path from $ID_1$ to $ID_2$ is to be examined next. Initially they equal to $ID^0$ and $ID^Y$ respectively. Let $ID_{it_1}$ and $ID_{jt_2}$ be the current values for $ID_1$ and $ID_2$. Upon completion of the inner loop, a new $ID$ on the $\frac{t_1+t_2}{2}$ th column will be selected, $ID_{new} = ID_{k\frac{t_1+t_2}{2}}$— midway between column $t_1$ and $t_2$. The particular $k$ depends on the previous $2(q(n)+1)$ bits $b_1^1b_2^1b_1^2b_2^2 \cdots b_1^{q(n)}b_2^{q(n)}b_1^{q(n)+1}b_2^{q(n)+1}$ used from $l$. It requires a moment's reflection, but it is not hard

to see that $k$ is precisely $b_2^1 b_2^2 \cdots b_2^{q(n)}$. And $b_2^{q(n)+1}$ decides whether $ID_1$ or $ID_2$ is to be replaced by $ID_{k\frac{t_1+t_2}{2}}$:

$$(ID_1, ID_2) = \begin{cases} (ID_1, ID_{new}) & \text{if } b_2^{q(n)+1} = 0 \\ (ID_{new}, ID_2) & \text{otherwise} \end{cases}$$

We finally describe the *Logspace* local computation. To start, the safe-store has the number 1. In general, the local machine uses the global clock as binary number $l$. Suppose Mission returns $\gamma_0$, $pad_0$, $ID_1$ and $ID_2$. By repeatedly taking 2 bits from $l$, the *Logspace* machine can figure out the right $\gamma_0$ and $pad_0$. Since the various $ID$'s are all practically written on the global clock $l$, the *Logspace* machine needs only to keep 2 pointers with $O(log(n))$ bits. ( The initial $ID^0$ and $ID^Y$ can be remembered in $O(log(n))$ bits. ) Given pointers to $ID_1$ and $ID_2$, it is a *Logspace* computation to check if $ID_2$ follows from $ID_1$ in a single move of the $PSPACE$ machine $M$.

Now the local machine will "behave" like $e \cdot pad_0$ if the transition exists from $ID_1$ to $ID_2$, and $\gamma_0 \cdot pad_0$ otherwise. This "behavior" is effected by applying the appropriate permutation to the number stored in the safe-storage ( a three bit number between 1 to 5 ).

It is easy to see that the entire series of local computations will behave like the identity $e$ sending 1 to 1, *if* there is a path from $ID^0$ to $ID^Y$; and behave like the 5-cycle (12345) sending 1 to 2 otherwise.

The proof is completed.

## 5 Conclusions

We have introduced the notions of safe-storage, or bottleneck machines as a way of further understanding the structure of what may lie between P and PSPACE. We have pointed out that there is a natural relationship between the safe-storage hierarchy and constant-width branching programs, which parallels the natural relationship between the polynomial-hierarchy and constant-depth circuits. This connection with constant-width branching programs led to the notion of serializability of computation and to the theorem that PSPACE is Logspace serializable. It seems remarkable that any PSPACE computation can be "strung out" in such a fashion. It is certainly stronger than the observation that PSPACE equals parallel Logspace. The connection also establishes the importance, at the level of P versus PSPACE, of lower bounds for constant-width branching programs.

## Acknowledgement

The authors thank Professor Mike Fischer for many valuable discussions. They thank Neil Immerman for first suggesting that the local computations might be done in Logspace instead of $P$. The authors also wish to acknowledge James Saxe for discussions several years ago that helped lead to the formulations presented here.

Thanks are due to the National Science Foundation whose grants MCS-8308805 and PYI grant DCR-8352081 helped fund this work.

## 6 References

[B] Barrington, D., *Bounded-width branching programs,* Ph.D. Thesis, Department of Mathematics, M.I.T, May 1986.

[C] Cai, J., "With probability one, a random oracle separates PSPACE from the polynomial-time hierarchy," Proc. 18th ACM Sym. on Theory of Computation, 1986, pp 21-29.

[CH] Cai, J., Hemachandra, L., "The Boolean hierarchy: hardware over NP," Proc. Structure in Complexity Theory, Springer-Verlag *Lecture Notes in Computer Science #223*, 1986, pp 105-124.

[FSS] Furst, M., Saxe, J., Sipser, M., "Parity, circuits and the polynomial-time hierarchy," Mathematical Systems Theory, 17, 1984, pp 13-27.

[HLS] Hartmanis, J., Lewis, P., Stearns, R., "Hierarchies of memory limited computations," IEEE Conference on Switching Circuit Theory and Logical Design, 1965, pp 179-190.

[HU] Hopcroft, J., Ullman, J., *Introduction to automata theory, languages, and computation,* Addison Wesley, 1979.

[S] Sipser, M., "Borel sets and circuit complexity," Proc. 15th ACM STOC, 1983, pp 61-69.

[SM] Stockmeyer, L. and Meyer, A, "Word problems requiring exponential time, preliminary report." Fifth Annual ACM Symposium on Theory of Computing, 1973.

[Y] Yao, A., "Separating the polynomial-time hierarchy by oracles," Proc. 26th IEEE Foundations of Computer Science, 1985, pp 1-10. ——