Trading Time and Space in Catalytic Branching Programs

Ian Mertz

University of Toronto

July 23, 2022

Joint work with James Cook
Space-bounded computation

$BP(w, \ell)$: layered branching programs of width $w$ and length $\ell$
Space-bounded computation

\[ \text{BP}(w, \ell) \text{ looks like } \text{SPACETIME}(\log w, \ell) \]
Space-bounded computation

$BP(w, \ell)$ looks like $SPACETIME(\log w, \ell)$ (idea: for a fixed timestamp, $w$ nodes in a layer $\leftrightarrow$ log $w$ bits in memory)
Space-bounded computation

$BP(w, \ell)$ looks like $SPACETIME(\log w, \ell)$ (idea: for a fixed timestamp, $w$ nodes in a layer $\leftrightarrow$ log $w$ bits in memory), but...

$SPACETIME$ is uniform: machine is “easy to describe” for every $n$
**Space-bounded computation**

$BP(w, \ell)$ looks like $SPACETIME(\log w, \ell)$ (idea: for a fixed timestamp, $w$ nodes in a layer $\leftrightarrow \log w$ bits in memory), but...

$SPACETIME$ is *uniform*: machine is “easy to describe” for every $n$

$BP$ is *non-uniform*: no restrictions on the description
Space-bounded computation

Every $f$ can be computed by $BP(2^{n-1}, n)$
Every $f$ can be computed by $BP(2^{n-1}, n)$
Amortized space-bounded computation
Amortized space-bounded computation
Amortized space-bounded computation

\( mCBP(w, \ell, m) \): \( m \) different branching programs (one source \( \rightarrow \) two sinks) which can share states
Amortized space-bounded computation

$mCBP(w, \ell, m)$: $m$ different branching programs (one input node, two output nodes) which can share states

\[
\begin{align*}
1 & \quad \cdots \quad (1,0) \\
2 & \quad \cdots \quad (1,1) \\
\vdots & \quad \cdots \\
i & \quad f(x) = 0 \quad (i,0) \\
\vdots & \quad f(x) = 1 \quad (i,1) \\
m & \quad \cdots \quad (m,1)
\end{align*}
\]
Amortized space-bounded computation

\( mCBP(w, \ell, m) \): \( m \) different branching programs (one source → two sinks) which can share states
Catalytic computation

\( CSPACETIME(s, t, c) \): space-bounded Turing Machines with an extra worktape (\( c \) bits) of full memory

\[
\begin{array}{cccccc}
\chi_1 & \chi_2 & \cdots & \chi_n & \\hline \\
\text{input tape} & \text{output}
\end{array}
\]

\[
\begin{array}{ccc}
\hline \\
\text{work tape} & \cdots & \\
\hline
\end{array}
\]

catalytic tape

\[
\begin{array}{cccccc}
0 & 1 & 1 & \cdots & 0 & 1
\end{array}
\]
Catalytic computation

\textbf{CSPACETIME}(s, t, c): space-bounded Turing Machines with an extra worktape (c bits) of full memory
Catalytic computation

$CSPACETIME(s, t, c)$: space-bounded Turing Machines with an extra worktape ($c$ bits) of full memory

\[
\begin{array}{cccc}
\chi_1 & x_2 & \cdots & x_n \\
\text{input tape} & & & \\
\hline
\text{work tape} & & & \\
0 & 1 & 1 & \cdots & 0 & 1 \\
\text{catalytic tape} & & & \\
\end{array}
\]

\[ f \]

output
Catalytic computation

Again $m \text{CBP}(w, \ell, m)$ looks like non-uniform $\text{CSPACETIME}(\log w, \ell, \log m)$
Catalytic computation

Again $m \text{CBP}(w, \ell, m)$ looks like non-uniform $\text{CSPACETIME}(\log w, \ell, \log m)$

$m \cdot w$ nodes in a layer $\leftrightarrow \log m + \log w$ bits in memory
Catalytic computation

Again $m\text{CBP}(w, \ell, m)$ looks like non-uniform $\text{CSPACETIME}(\log w, \ell, \log m)$

- $m \cdot w$ nodes in a layer $\leftrightarrow \log m + \log w$ bits in memory
- $m$ sources plus source-sink pairing requirement $\leftrightarrow$ resetting $\log m$ catalytic memory)
Catalytic computation

Two interpretations of reducing $w$ and $m$ (non-uniform):

1) amortized space: reducing the amortized space ($w = (w \cdot m) / m$) needed to compute $f$, or the number of copies ($m$) needed for amortization to help

2) catalytic space: reducing the amount of space ($\log w$) and catalytic space ($\log m$) needed to compute $f$
Catalytic computation

Two interpretations of reducing $w$ and $m$ (non-uniform):

1) *amortized space*: reducing the amortized space
\[(w = (w \cdot m)/m)\] needed to compute $f$, or the number of copies
($m$) needed for amortization to help
Catalytic computation

Two interpretations of reducing $w$ and $m$ (non-uniform):

1) *amortized space*: reducing the amortized space $(w = (w \cdot m)/m)$ needed to compute $f$, or the number of copies $(m)$ needed for amortization to help

2) *catalytic space*: reducing the amount of space ($\log w$) and catalytic space ($\log m$) needed to compute $f$
Known results

[Potchin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$. 

Counting argument: almost every function $f$ requires branching programs to have either non-amortized width or length $2\Omega(n)$. In contrast, [Potchin’17] gives (asymptotically) optimal amortized width $w = O(1)$ and length $\ell = O(n)$ simultaneously...but we need $m = 2^{2n-1}$ to get it!
[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$.

Counting argument: almost every function $f$ requires branching programs to have either non-amortized width or length $2^{\Omega(n)}$. 
Known results

[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$.

Counting argument: almost every function $f$ requires branching programs to have either non-amortized width or length $2^\Omega(n)$.

In contrast, [Potechin’17] gives (asymptotically) optimal amortized width $w = O(1)$ and length $\ell = O(n)$ simultaneously.
[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$.

Counting argument: almost every function $f$ requires branching programs to have either non-amortized width or length $2^{\Omega(n)}$.

In contrast, [Potechin’17] gives (asymptotically) optimal amortized width $w = O(1)$ and length $\ell = O(n)$ simultaneously.

...but we need $m = 2^{2^n-1}$ to get it!
Our results

[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{2^m-1}$.
Our results

[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{2^n-1}$.

Main result 1: for any $\epsilon > 0$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $O_\epsilon(n)$, where $m = 2^{2^\epsilon n}$. 

[Potechin’17]’: every function \( f \) can be computed by a read-4 permutation branching program of width \( 2^{2n+1} \).

Main result 1’: for any \( \epsilon > 0 \), every function \( f \) can be computed by a read-\( O_\epsilon(1) \) permutation branching program of width \( 2^{2\epsilon n} \).
[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{2^n-1}$. 

**Setup:** catalytic space $\log m = 2^{2^n-1}$ in some initial state $\tau_1 \ldots \tau_{2^n-1}$, plus $\log 4 = 2$ bits of free work space $(00 \ldots 0)$, $(00 \ldots 1)$, $\ldots$, $((\tau_1 \tau_2 \ldots \tau_{2^n-1}), 0)$, $((\tau_1 \tau_2 \ldots \tau_{2^n-1}), 1)$, $\ldots$, $((11 \ldots 1), 1)$.
[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{2^n-1}$.

Setup: catalytic space $\log m = 2^n - 1$ in some initial state $\tau_1 \ldots \tau_{2^n-1}$, plus $\log 4 = 2$ bits of free work space

\[
\begin{align*}
(00 \ldots 0) & \quad (00 \ldots 0), 0 \\
(00 \ldots 1) & \quad (00 \ldots 1), 0 \\
\vdots & \quad \vdots \\
(\tau_1 \tau_2 \ldots \tau_{2^n-1}) & \quad f(x) = 0 \\
\vdots & \quad \vdots \\
(11 \ldots 1) & \quad (11 \ldots 1), 1
\end{align*}
\]
0) First free bit: $\vec{0}$ entry of $g$

$g(0) = 0$
1) \( g(\alpha_1 \ldots \alpha_i \ldots \alpha_n) \rightarrow g(\alpha_1 \ldots \alpha_i^{x_i} \ldots \alpha_n) \)

\[ g^{\oplus x}(x) = 0 \]

\( x_1 \ldots x_n \)
**Potechin'17** in two slides

2) \( g(y) \rightarrow g(y) + f(y) \)

\[
(g^{\oplus x} + f)(x) = f(x)
\]

(no reads)
3) $g(\alpha_1 \ldots \alpha_i^x \ldots \alpha_n) \rightarrow g(\alpha_1 \ldots \alpha_i \ldots \alpha_n)$

$$(g \oplus_t + f) \oplus_t (0) = f(x)$$
4) Second free bit (output): copy the answer from first free bit

\[(g \oplus x + f) \oplus x(0) = f(x)\]
5) Undo steps 1-3 (do steps 3-1)

\[ f(x) = 1 \quad f(x) = 0 \]

\[ g(0) = 0 \]
Trading time and space

*Truth table* representation [Potechin’17]:

\[
f(x) = \sum_{\alpha \in \{0,1\}^n} f(\alpha) \cdot [x = \alpha]
\]
Trading time and space

Truth table representation [Potechin’17]:

\[ f(x) = \sum_{\alpha \in \{0,1\}^n} f(\alpha) \cdot [x = \alpha] \]

Monomial representation [Cook-Mertz’20,21]:

\[ f(x) = \sum_{S \subseteq [n]} f_{\text{mon}}(S) \cdot \prod_{i \in S} x_i \mod 2 \]
Trading time and space

*Truth table* representation [Potechin’17]:

\[ f(x) = \sum_{\alpha \in \{0,1\}^n} f(\alpha) \cdot [x = \alpha] \]

*Monomial* representation [Cook-Mertz’20,21]:

\[ f(x) = \sum_{S \subseteq [n]} f_{mon}(S) \cdot \prod_{i \in S} x_i \mod 2 \]

Catalytic algorithms give us a way to compute \( f \) over the monomial basis only using catalytic memory.
Trading time and space

Monomial representation [Cook-Mertz’20,21]:

\[ f(x) = \sum_{S \subseteq [n]} f_{mon}(S) \cdot \prod_{i \in S} x_i \mod 2 \]

Two algorithms for monomial rep., different types of efficiency:
Trading time and space

*Monomial representation* [Cook-Mertz’20,21]:

\[
    f(x) = \sum_{S \subseteq [n]} f_{\text{mon}}(S) \cdot \prod_{i \in S} x_i \quad \text{mod 2}
\]

Two algorithms for monomial rep., different types of efficiency:

1) *Potechin algorithm (monomial basis edition)*
   
   - compute each monomial into separate memory in parallel
   - linear time, exponential space
Trading time and space

Monomial representation [Cook-Mertz’20,21]:

\[ f(x) = \sum_{S \subseteq [n]} f_{\text{mon}}(S) \cdot \prod_{i \in S} x_i \mod 2 \]

Two algorithms for monomial rep., different types of efficiency:

1) *Potechin algorithm (monomial basis edition)*
   - compute each monomial into separate memory in parallel
   - linear time, exponential space

2) *Cook-Mertz algorithm (branching program edition)*
   - compute each monomial directly into the output register in series
   - exponential time, linear space
Trading time and space

Main result 1: for any $\epsilon > 2/n$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$, where $m = 2^{n + \frac{1}{\epsilon} \cdot 2^{\epsilon n}}$. 

Proof idea: use time-efficient algorithm to compute monomials only up to degree $\epsilon n$, then use space-efficient algorithm to combine them to get the higher degree monomials.

Small monomials: $(n \leq \epsilon n)$ monomials $\rightarrow$ space $n \epsilon n$ better: split variables into $1/\epsilon$ groups $\rightarrow$ space $1/\epsilon \cdot 2^{\epsilon n}$ ($+ n$)

Large monomials: degree $1/\epsilon$ $\rightarrow$ time $2^{1/\epsilon} \cdot 2^{\epsilon n}$.
Main result 1: for any $\epsilon > 2/n$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$, where $m = 2^{n + 1/\epsilon \cdot 2\epsilon n}$.

Proof idea: use time-efficient algorithm to compute monomials only up to degree $\epsilon n$, then use space-efficient algorithm to combine them to get the higher degree monomials.
Trading time and space

Main result 1: for any $\epsilon > 2/n$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$, where $m = 2^{n + \frac{1}{\epsilon} \cdot 2\epsilon n}$.

Proof idea: use time-efficient algorithm to compute monomials only up to degree $\epsilon n$, then use space-efficient algorithm to combine them to get the higher degree monomials.

- Small monomials: $\left( \sum_{\leq \epsilon n} \right)$ monomials $\rightarrow$ space $n^{\epsilon n}$
Main result 1: for any $\epsilon > 2/n$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$, where $m = 2^{n+\frac{1}{\epsilon} \cdot 2\epsilon n}$.

Proof idea: use time-efficient algorithm to compute monomials only up to degree $\epsilon n$, then use space-efficient algorithm to combine them to get the higher degree monomials.

- Small monomials: $\binom{n}{\leq \epsilon n}$ monomials $\rightarrow$ space $n^{\epsilon n}$
  - better: split variables into $\frac{1}{\epsilon}$ groups $\rightarrow$ space $\frac{1}{\epsilon} \cdot 2^{\epsilon n} (\pm n)$
Main result 1: for any $\epsilon > 2/n$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$, where $m = 2^{n+\frac{1}{\epsilon} \cdot 2\epsilon n}$.

Proof idea: use time-efficient algorithm to compute monomials only up to degree $\epsilon n$, then use space-efficient algorithm to combine them to get the higher degree monomials.

- Small monomials: $\binom{n}{\leq \epsilon n}$ monomials $\rightarrow$ space $n^\epsilon n$
  - better: split variables into $\frac{1}{\epsilon}$ groups $\rightarrow$ space $\frac{1}{\epsilon} \cdot 2^{\epsilon n} (+n)$
- Large monomials: degree $1/\epsilon \rightarrow$ time $2^{1/\epsilon} \cdot 2\epsilon n$
Trading time and space

Main result 1: for any $\epsilon > 2/n$, every function $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$, where $m = 2^{n+\frac{1}{\epsilon} \cdot 2\epsilon n}$.

Proof idea: use time-efficient algorithm to compute monomials only up to degree $\epsilon n$, then use space-efficient algorithm to combine them to get the higher degree monomials.

- Small monomials: $\left(\binom{n}{\leq \epsilon n}\right)$ monomials $\rightarrow$ space $n^{\epsilon n}$
  - better: split variables into $\frac{1}{\epsilon}$ groups $\rightarrow$ space $\frac{1}{\epsilon} \cdot 2^{\epsilon n} (+n)$
- Large monomials: degree $1/\epsilon$ $\rightarrow$ time $2^{1/\epsilon} \cdot (2\epsilon n)$
Extending to easier functions

[Potechin’17]: every function \( f \) can be computed by an \( m \)-catalytic branching program of width \( 4m \) and length \( 4n \), where \( m = 2^{2^n - 1} \).
Extending to easier functions

[Potechin’17]: every function $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{2n-1}$.

[Robere-Zuiddam’22]: if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{(\leq d)-1}$. 
[Robere-Zuiddam’22]: if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{(\leq d)-1}$.

Proof idea (original): for low degree $f$, the Potechin algorithm has many isomorphic disjoint components based on the symmetries of the polynomial associated with $f$. 
Extending to easier functions

[Robere-Zuiddam’22]: if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{(\leq d)} - 1$.

Proof idea (original): for low degree $f$, the Potechin algorithm has many isomorphic disjoint components based on the symmetries of the polynomial associated with $f$.

Proof idea (new): monomial version of Potechin algorithm again, but now only compute monomials which actually appear in $f$ $\left(\binom{n}{\leq d}\right)$ by assumption.
Extending to easier functions

[Robere-Zuiddam’22]: if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{\left(\frac{n}{d}\right)-1}$. 

Main result 2: for any $\epsilon > \frac{2}{d}$, if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $2^m$ and length $2^{\frac{1}{\epsilon} \cdot 2n}$, where $m = 2^{n+1} \cdot \epsilon \cdot \left(\frac{n}{\epsilon d}\right)$. 

Proof idea: same time-space tradeoff as before, now with $\epsilon$ instead of $\epsilon n$. 
Extending to easier functions

[Robere-Zuiddam’22]: if \( f \) is a degree \( d \) polynomial over \( \mathbb{F}_2 \), then \( f \) can be computed by an \( m \)-catalytic branching program of width \( 4m \) and length \( 4n \), where \( m = 2^{<d}-1 \).

Main result 2: for any \( \epsilon > 2/d \), if \( f \) is a degree \( d \) polynomial over \( \mathbb{F}_2 \), then \( f \) can be computed by an \( m \)-catalytic branching program of width \( 2m \) and length \( 2^{1/\epsilon} \cdot 2n \), where \( m = 2^{n+1/\epsilon} \cdot (n^{<d}) \).
Extending to easier functions

[Robere-Zuiddam’22]: if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $4m$ and length $4n$, where $m = 2^{\left\lfloor \frac{n}{d} \right\rfloor - 1}$.

Main result 2: for any $\epsilon > 2/d$, if $f$ is a degree $d$ polynomial over $\mathbb{F}_2$, then $f$ can be computed by an $m$-catalytic branching program of width $2m$ and length $2^{\frac{1}{\epsilon}} \cdot 2n$, where $m = 2^{n + \frac{1}{\epsilon} \cdot \left\lfloor \frac{n}{\epsilon d} \right\rfloor}$.

Proof idea: same* time-space tradeoff as before, now with $\epsilon d$ instead of $\epsilon n$.  

*
Saving time

All results are linear time, which is optimal up to a constant factor. But how small can we get the constant?
All results are linear time, which is optimal up to a constant factor. But how small can we get the constant?

Main result 3: every $f$ can be computed by an $m$-catalytic (or even permutation) branching program of length $4n - 4$ and width $4m$, where $m = 2^{2^n-1}$. 
All results are linear time, which is optimal up to a constant factor. But how small can we get the constant?

**Main result 3:** every $f$ can be computed by an $m$-catalytic (or even permutation) branching program of length $4n - 4$ and width $4m$, where $m = 2^{2^n - 1}$.

**Main result 4:** any permutation* branching program calculating the AND function which reads any variable less than three times requires length at least $4n - 4$. 


Open problems

Save on *either* time or space (while keeping other optimal)
▶ would give better tradeoff algorithm
Open problems

Save on either time or space (while keeping other optimal)
  ▶ would give better tradeoff algorithm

Show that for some $f$, $m$ must be at least $2^n$ to get linear amortized size
  ▶ counting only gives $m \geq 2^n/O(n)$
Open problems

Save on *either* time or space (while keeping other optimal)
  ▶ would give better tradeoff algorithm

Show that for some $f$, $m$ must be at least $2^n$ to get linear amortized size
  ▶ counting only gives $m \geq 2^n/O(n)$

Optimal permutation branching program length for any function
  ▶ somewhere between $3n^*$ and $4n - 4$
  ▶ can get a read-3 program for $AND(x_1, x_2, x_3)$