Data Compression via Logic Synthesis

Luca Amarú¹, Pierre-Emmanuel Gaillardon¹, Andreas Burg², Giovanni De Micheli¹

Integrated Systems Laboratory (LSI), EPFL, Switzerland¹
Telecommunication Circuits Laboratory (TCL), EPFL, Switzerland²

Thursday, January 23, 2014
Data Compression

- Software and hardware applications are committed to reduce the footprint and resource usage of data.

- Standard data compression: data decorrelation + entropy encoding.
- EDA methods are powerful and scalable: they solve also non-EDA problems. Logic synthesis is a primary EDA application.

Can Modern Logic Synthesis Help Compressing Binary Data?
Outline

1. Introduction and Motivation
2. Data Compression via Logic Synthesis
3. Experimental Results
4. Conclusions
1 Introduction and Motivation

2 Data Compression via Logic Synthesis

3 Experimental Results

4 Conclusions
(Brief) Introduction on Data Compression

**(Lossless) Data Compression:** data decorrelation + entropy encoding

- **Data decorrelation:**
  - Reduces the autocorrelation of the input data.
  - Typically achieved via linear decorrelation transforms.
  - *Karhunen-Loeve Transform* (KLT), *Discrete Cosine Transform* (DCT) etc.

- **Entropy encoding:**
  - Compress an input data down to its entropy.
  - With exact probabilistic model, entropy encoding is optimum.
  - Huffman coding, arithmetic coding, etc.
Why Are We Interested in a Different Approach?

With the perfect data decorrelation, entropy encoding is optimal.

Unfortunately, perfect data decorrelation is intractable.

How to unlock ultimate lossless data compression?

Approach the problem from a new angle.

Logic synthesis shares similar optimization criteria.

Use logic synthesis as core data compression engine.
1 Introduction and Motivation

2 Data Compression via Logic Synthesis

3 Experimental Results

4 Conclusions
Data Compression via Logic Synthesis

**Logic synthesis**: Boolean function $\Rightarrow$ minimal logic circuit (size).

**Data compression**: Binary data $\Rightarrow$ minimal representation (# bits).

---

### Alternative Data Compression Flow

- **Binary data** (N bits)
  - $\downarrow$
  - **Function Description**
  
  - **Boolean function**
  - $\downarrow$
  - **Logic Synthesis**
  
  - **Optimized logic circuit** (M bits)
Data Compression via Logic Synthesis – Example

Prior art example: Binary data ⇒ Truth table ⇒ 2-level minimized form

Input binary data $B = 0001001111111111$

$B$ is the entry vector of a truth table for a 4 inputs Boolean function.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$w$</th>
<th>$y$</th>
<th>$z$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2-level logic synthesis: $B \Rightarrow x + yw + yz$
Data Compression via Logic Synthesis – Example

Data Decompression:

\[ B(0) = (x + yw + yz)@\{(x = 0, w = 0, y = 0, z = 0) = 0 \} \]
\[ B(1) = (x + yw + yz)@\{(x = 0, w = 0, y = 0, z = 1) = 0 \} \]
\[ B(2) = (x + yw + yz)@\{(x = 0, w = 0, y = 1, z = 0) = 0 \} \]
\[ B(3) = (x + yw + yz)@\{(x = 0, w = 0, y = 1, z = 1) = 1 \} \]

... 

In general:

\[
\text{for}(i=0; i< 2^{\#vars}; i++)
\]
\[ B(i) = (x + yw + yz)@\{BR(i)\} \]
\[
\text{endfor}
\]
Data Compression via Logic Synthesis – Scalability

Monolithic truth tables may hide compression opportunities.

Very often data to be compressed is generated sequentially.

Storing everything in a single output is not efficient.
New Logic Model for Data Compression

Partition the input in $M$ sub-blocks of fixed length $L = |B|/M$.

Describe a logic circuit that stimulated by $BR(i)$ generates $S_i$.

Simulating the logic circuit it is possible to build back $B$. 

Binary String

\[
B = 0001010101010001110101\ldots00010
\]

\[
S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_{M-1}
\]

\[
S_{000} \quad S_{001} \quad S_{010} \quad S_{011} \quad S_{100} \quad S_{BR(M-1)}
\]
New Logic Model for Data Compression – Example

M=8, L=3  

Binary String

\[ B=000001010011000001110111 \]

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad S_6 \quad S_7 \]

Focus on the first bit of the sub-blocks

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad S_6 \quad S_7 \]

\[ 000 \quad 001 \quad 010 \quad 011 \quad 000 \quad 001 \quad 110 \quad 111 \]

The first bit is logic 1 when

\[ I_0 I_1 \bar{I}_2 \text{ OR } I_0 I_1 I_2 = I_0 I_1 \]

Logic Circuit

\[ \text{Logic for } Si(1) \text{ Si}(2) \]
Describing the Logic Circuit: Algorithm

Algorithm 1 \( G \) function description.

**INPUT:** binary strings \( \{S_0, S_1, ..., S_{M-1}\} \) (\( L \)-bits per each)

**OUTPUT:** SOP representation for \( G \) function

**FUNCTION:** Construct \( G(\{S_0, S_1, ..., S_{M-1}\}) \)

\[
\text{for all } k = 0 : L - 1 \text{ do}
\]

\[
\quad \text{for all } i = 0 : M - 1 \text{ do}
\]

\[
\quad \quad \text{if } (S_i(k) == 1) \text{ then}
\]

\[
\quad \quad \quad \text{add cube } BR(i) \text{ to SOP for the } k\text{-th output of } G
\]

\[
\quad \quad \text{end if}
\]

\[
\quad \text{end for}
\]

\[
\text{end for}
\]
Data Compression Flow

Compression Flow

Binary data \( (N_o\) bits) 

\[ \Downarrow \] Partitioning

Partitioned binary data (M sub-blocks long \(|B|/M\) each) 

\[ \Downarrow \] SOP Description Algorithm

G Function Description 

\[ \Downarrow \] Multi-level Logic Synthesis

Optimized logic circuit for \( G (N_c\) bits)
Improving the Compression/Synthesis Efficiency

- Let us fix a decompression sense:
- The (compressed) logic circuit $G$ can be stimulated by $BR(i)$ to produce $S(i)$ iff it has been previously stimulated by $BR(i - 1)$ to produce $S(i - 1)$.
- This has no impact on the decompression performance.
- But $S(i - 1) = G(BR(i))$ can now be used as additional input to $G$.

- With this information, the logic synthesizer has more freedom.
- Also $S(i - 1), S(i - 2)$ etc. can be used.
Improving the Compression/Synthesis Efficiency – Motivation Example

- Suppose we want to compress a binary string generated by:
  \[ F_n = \left( \varphi^n - \psi^n \right) / \sqrt{5} \] with \( \varphi = 1.6180339887... \) and \( \psi = -1/\varphi \).

- Suppose we have no knowledge about \( S(i - 1), S(i - 2), \) etc.

- The logic synthesizer receives as inputs only \( BR(i) \).

- Even if the synthesizer is very powerful it is unlikely to recognize
  \[ F_n = \left( \varphi^n - \psi^n \right) / \sqrt{5}. \]
Improving the Compression/Synthesis Efficiency – Motivation Example

- Suppose we still want to compress a binary string generated by:
  \[ F_n = (\varphi^n - \psi^n)/\sqrt{5} \] with \( \varphi = 1.6180339887\ldots \) and \( \psi = -1/\varphi \).

- Suppose we have knowledge about \( S(i - 1), S(i - 2) \).
- The decompression has a fixed sense \( (S_0, S_1, S_2, \ldots, S_{M-1}) \).
- The logic synthesizer receives as inputs \( BR(i) \) and \( S(i - 1), S(i - 2) \).

- It is much easier for a synthesizer to recognize \( F_n = F_{n-1} + F_{n-2} \) (Fibonacci sequence).
Synthesis facilitated Logic Circuit Description

**Algorithm 2** Synthesis-facilitated description of $G$.

**INPUT:** binary strings $\{S_0, S_1, ..., S_{M-1}\}$ ($L$-bits per each)

**OUTPUT:** SOP representation for $G$ function

**FUNCTION:** Construct $G(\{S_0, S_1, ..., S_{M-1}\})$

\[
\begin{align*}
\text{for all } k &= 0 : L - 1 \text{ do} \\
&\quad \text{for all } i = 0 : M - 1 \text{ do} \\
&\quad \quad \text{if } (S_i(k) == 1) \text{ then} \\
&\quad \quad \quad \text{add cube BR}(i) \text{ to SOP for the } k\text{-th output of } G \\
&\quad \quad \quad \text{if } (S_{i-1} \text{ is unique in } \{S_0, S_1, ..., S_{M-1}\}) \text{ then} \\
&\quad \quad \quad \quad \text{add cube } S_{i-1} \text{ to SOP for the } k\text{-th output of } G \\
&\quad \quad \text{end if} \\
&\quad \text{end if} \\
&\text{end for} \\
&\text{end for}
\end{align*}
\]

$s_{i-1}$ can be used as alternative (logical or with BR($i$)) information to describe $G$
**Improved Data Compression Flow**

**Binary data** \((N_o \text{ bits})\)

\[\downarrow\] \hspace{1cm} Partitioning

**Partitioned binary data** \((M \text{ sub-blocks long } |B|/M \text{ each})\)

\[\downarrow\] \hspace{1cm} \(BR(i)/S(i-1)\) Description

**G Function Description**

\[\downarrow\] \hspace{1cm} Multi-level Logic Synthesis

**Optimized logic circuit for** \(G\) \((N_c \text{ bits})\)
What if the Synthesis is not Satisfactory?

- For hard functions logic synthesis may lead to very large circuits or too long runtime.
- But we want to be fast and at the same time efficient.

- Idea: consider one output bit of $S_i$ per time.
- If the synthesis of such output bit is too hard (timeout or not advantageous) – use entropy encoding for the corresponding bits.
- Otherwise keep the synthesis results.

- Merge synthesis results with entropy encoding results to get final compressed data.
**Final Data Compression Flow**

**Final Compression Flow**

**Binary data** \((N_o \text{ bits})\)

\[
\downarrow \quad \text{Partitioning}
\]

**Partitioned binary data** \((M \text{ sub-blocks long } |B|/M \text{ each})\)

\[
\downarrow \quad BR(i)/S(i - 1) \quad \text{Description}
\]

**G Function Description**

\[
\downarrow \quad \text{Multi-level Logic Synthesis}
\]

**Optimized logic circuit for } G\)

\[
\downarrow \quad \text{Entropy encoding of bits too hard to synthesize}
\]

**Compressed data** \((\text{synthesis + entropy encoding results}) \ (N_c \text{ bits})\)
Final Decompression Flow

- Use FSM to rebuild back part of the $S_i$.
- Entropy decoding of the hard to synthesize bits.
- Interleave the results (recalling back the hard bits position in $S_i$).
Final Decompression Flow – Example

• From the FSM ($M = 3$): $X = 000111010 = \{000, 111, 010\}$.
• Entropy decoding ($2^{nd}$ index in $S_i$): $Y = 101$.
• Interleaving $B = \{0100, 1011, 0110\} = 010010110110$. 
1 Introduction and Motivation

2 Data Compression via Logic Synthesis

3 Experimental Results

4 Conclusions
Experimental Setup 1/2

- Logic synthesis engine:
  - *ABC*: *resyn2* optimization script and *ABC* mapper (academic).

- Entropy encoding: ZIP tool.

- Algorithms implemented in *C* language.

- Interaction with external tools: *Perl* language.

- Comparison with:
  - ZIP tool.
  - DCT + ZIP tool.
  - bzip2 tool.
  - 7zip tool.
Experimental Setup 2/2

- Benchmarks deriving from casual processes:
  - Perfect line measurement.
  - Line measurement + white noise.
  - Parabolic measurement.
  - Simple computer (logic) program generating binary data.
## Experimental Results: Memory Footprint

<table>
<thead>
<tr>
<th>Bench</th>
<th>Size</th>
<th>ZIP</th>
<th>DCT+ZIP</th>
<th>bzip2</th>
<th>7zip</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>2.2 MB</td>
<td>208 KB</td>
<td>868 KB</td>
<td>316 KB</td>
<td>60 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>25 MB</td>
<td>2.1 MB</td>
<td>8.3 MB</td>
<td>3.1 MB</td>
<td>888 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>287 MB</td>
<td>21 MB</td>
<td>81 MB</td>
<td>31 MB</td>
<td>3.4 MB</td>
<td>302 KB</td>
</tr>
<tr>
<td>Linear + Noise</td>
<td>2.2 MB</td>
<td>264 KB</td>
<td>872 KB</td>
<td>258 KB</td>
<td>212 KB</td>
<td>80 KB</td>
</tr>
<tr>
<td></td>
<td>25 MB</td>
<td>2.7 MB</td>
<td>8.4 MB</td>
<td>2.6 MB</td>
<td>2.4 MB</td>
<td>700 KB</td>
</tr>
<tr>
<td></td>
<td>287 MB</td>
<td>27 MB</td>
<td>84 MB</td>
<td>30 MB</td>
<td>23 MB</td>
<td>7.1 MB</td>
</tr>
<tr>
<td>Quadratic</td>
<td>3.3 MB</td>
<td>484 KB</td>
<td>816 KB</td>
<td>532 KB</td>
<td>272 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>39 MB</td>
<td>5.3 MB</td>
<td>7.6 MB</td>
<td>6.1 MB</td>
<td>3.3 MB</td>
<td>16 KB</td>
</tr>
<tr>
<td></td>
<td>449 MB</td>
<td>59 MB</td>
<td>71 MB</td>
<td>67 MB</td>
<td>40 MB</td>
<td>566 KB</td>
</tr>
<tr>
<td>Program</td>
<td>1.6 MB</td>
<td>116 KB</td>
<td>304 KB</td>
<td>124 KB</td>
<td>44 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>20 MB</td>
<td>1.2 MB</td>
<td>3.2 MB</td>
<td>1.5 MB</td>
<td>796 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>230 MB</td>
<td>12 MB</td>
<td>31 MB</td>
<td>15 MB</td>
<td>3.8 MB</td>
<td>234 KB</td>
</tr>
</tbody>
</table>
Experimental Results: Memory Footprint

<table>
<thead>
<tr>
<th>Bench</th>
<th>Size</th>
<th>ZIP</th>
<th>DCT+ZIP</th>
<th>bzip2</th>
<th>7zip</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>2.2 MB</td>
<td>208 KB</td>
<td>868 KB</td>
<td>316 KB</td>
<td>60 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>25 MB</td>
<td>2.1 MB</td>
<td>8.3 MB</td>
<td>3.1 MB</td>
<td>888 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>287 MB</td>
<td>21 MB</td>
<td>81 MB</td>
<td>31 MB</td>
<td>3.4 MB</td>
<td>302 KB</td>
</tr>
<tr>
<td>Linear + Noise</td>
<td>2.2 MB</td>
<td>264 KB</td>
<td>872 KB</td>
<td>258 KB</td>
<td>212 KB</td>
<td>80 KB</td>
</tr>
<tr>
<td></td>
<td>25 MB</td>
<td>2.7 MB</td>
<td>8.4 MB</td>
<td>2.6 MB</td>
<td>2.4 MB</td>
<td>700 KB</td>
</tr>
<tr>
<td></td>
<td>287 MB</td>
<td>27 MB</td>
<td>84 MB</td>
<td>30 MB</td>
<td>23 MB</td>
<td>7.1 MB</td>
</tr>
<tr>
<td>Quadratic</td>
<td>3.3 MB</td>
<td>484 KB</td>
<td>816 KB</td>
<td>532 KB</td>
<td>272 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>39 MB</td>
<td>5.3 MB</td>
<td>7.6 MB</td>
<td>6.1 MB</td>
<td>3.3 MB</td>
<td>16 KB</td>
</tr>
<tr>
<td></td>
<td>449 MB</td>
<td>59 MB</td>
<td>71 MB</td>
<td>67 MB</td>
<td>40 MB</td>
<td>566 KB</td>
</tr>
<tr>
<td>Program</td>
<td>1.6 MB</td>
<td>116 KB</td>
<td>304 KB</td>
<td>124 KB</td>
<td>44 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>20 MB</td>
<td>1.2 MB</td>
<td>3.2 MB</td>
<td>1.5 MB</td>
<td>796 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>230 MB</td>
<td>12 MB</td>
<td>31 MB</td>
<td>15 MB</td>
<td>3.8 MB</td>
<td>234 KB</td>
</tr>
</tbody>
</table>

Data compression via logic synthesis presents best results.

Logic synthesis identifies the function correlating a data set.
## Experimental Results: Memory Footprint

<table>
<thead>
<tr>
<th>Bench</th>
<th>Size</th>
<th>ZIP</th>
<th>DCT+ZIP</th>
<th>bzip2</th>
<th>7zip</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>2.2 MB</td>
<td>208 KB</td>
<td>868 KB</td>
<td>316 KB</td>
<td>60 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>25 MB</td>
<td>2.1 MB</td>
<td>8.3 MB</td>
<td>3.1 MB</td>
<td>888 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>287 MB</td>
<td>21 MB</td>
<td>81 MB</td>
<td>31 MB</td>
<td>3.4 MB</td>
<td>302 KB</td>
</tr>
<tr>
<td>Linear + Noise</td>
<td>2.2 MB</td>
<td>264 KB</td>
<td>872 KB</td>
<td>258 KB</td>
<td>212 KB</td>
<td>80 KB</td>
</tr>
<tr>
<td></td>
<td>25 MB</td>
<td>2.7 MB</td>
<td>8.4 MB</td>
<td>2.6 MB</td>
<td>2.4 MB</td>
<td>700 KB</td>
</tr>
<tr>
<td></td>
<td>287 MB</td>
<td>27 MB</td>
<td>84 MB</td>
<td>30 MB</td>
<td>23 MB</td>
<td>7.1 MB</td>
</tr>
<tr>
<td>Quadratic</td>
<td>3.3 MB</td>
<td>484 KB</td>
<td>816 KB</td>
<td>532 KB</td>
<td>272 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>39 MB</td>
<td>5.3 MB</td>
<td>7.6 MB</td>
<td>6.1 MB</td>
<td>3.3 MB</td>
<td>16 KB</td>
</tr>
<tr>
<td></td>
<td>449 MB</td>
<td>59 MB</td>
<td>71 MB</td>
<td>67 MB</td>
<td>40 MB</td>
<td>566 KB</td>
</tr>
<tr>
<td>Program</td>
<td>1.6 MB</td>
<td>116 KB</td>
<td>304 KB</td>
<td>124 KB</td>
<td>44 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>20 MB</td>
<td>1.2 MB</td>
<td>3.2 MB</td>
<td>1.5 MB</td>
<td>796 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td></td>
<td>230 MB</td>
<td>12 MB</td>
<td>31 MB</td>
<td>15 MB</td>
<td>3.8 MB</td>
<td>234 KB</td>
</tr>
</tbody>
</table>

AWGN is identified in the flow – bits hard to synthesize.

Entropy encoding handle AWGN (anyway not compressible).

Significant compression for the remaining bits.
Experimental Results: Runtime

- 1\textsuperscript{st} place: ZIP.
- 2\textsuperscript{nd} place: bzip2 - 1.5×ZIP.
- 3\textsuperscript{rd} place: 7zip - 8×ZIP.
- 4\textsuperscript{th} place: this work - 12×ZIP.

- ZIP is the fastest tool - based on very fast algorithms.
- Our proposal involves logic synthesis - a time consuming technique.
- Speed-up is possible by integrating logic synthesis and entropy encoding techniques in the same code.
1 Introduction and Motivation

2 Data Compression via Logic Synthesis

3 Experimental Results

4 Conclusions
Conclusions

- **Software and hardware** applications are committed to **reduce the footprint** and resource usage of data.
- In this work we **use logic synthesis** to compact the size binary data.
- **Data compression via logic synthesis**: create a Boolean function describing the binary data + minimize such Boolean function.

- An **expressive logic model** is key to find the underlying logic function generating the input data.
- Our proposal is intended for **highly-correlated data sets**.
- Our proposal generates the **best results** as compared to state-of-art compression tools at the price of runtime overhead.
Questions?

Thank you for your attention.