Data Compression via Logic Synthesis

Luca Amarú[1], Pierre-Emmanuel Gaillardon[1], Andreas Burg[2], Giovanni De Micheli[1]

Integrated Systems Laboratory (LSI), EPFL, Switzerland[1]

Telecommunication Circuits Laboratory (TCL), EPFL, Switzerland[2]

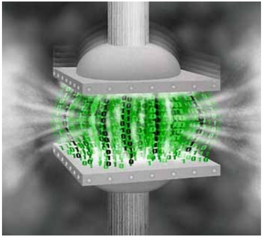
Thursday, January 23, 2014

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Integrated Systems Laboratory

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.

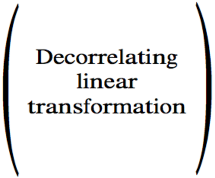
Can Modern Logic Synthesis Help Compressing Binary Data?

Outline

(Brief) Introduction on Data Compression

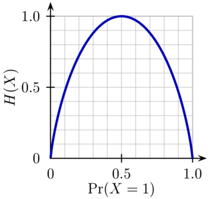
(Lossless) Data Compression: data decorrelation + entropy enconding

- Data decorrelation:



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

- Entropy enconding:



  - Compress an input data down to its entropy.
  - With exact probabilistic model, entropy enconding 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.

Data Compression via Logic Synthesis

Logic synthesis: Boolean function ⇒ minimal logic circuit (size).

Data compression: Binary data ⇒ minimal representation (# bits).

Alternative Data Compression Flow

Binary data (N bits)

⇓ **Function Description**

Boolean function

⇓ **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.

x	w	y	z	B
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	**1**
1	1	1	1	**1**

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:

for(i=0;i$< 2^{\#vars}$**;i++)**

$$B(i) = (x + yw + yz)@(BR(i))$$

endfor

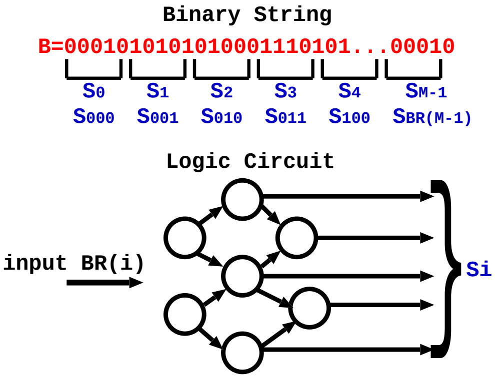
Data Compression via Logic Synthesis – Scalability

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.

New Logic Model for Data Compression – Example

M=8, L=3 **Binary String**

B=000001010011000001110111

S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
000	001	010	011	000	001	110	111

Focus on the first bit of the sub-blocks

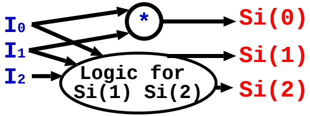
$\bar{I_0}\bar{I_1}\bar{I_2}$	$\bar{I_0}\bar{I_1}I_2$	$\bar{I_0}I_1\bar{I_2}$	$\bar{I_0}I_1I_2$	$I_0\bar{I_1}\bar{I_2}$	$I_0\bar{I_1}I_2$	$I_0I_1\bar{I_2}$	$I_0I_1I_2$
S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
000	001	010	011	000	001	110	111

The first bit is logic 1 when

$$I_0I_1\bar{I_2} \text{ OR } I_0I_1I_2$$

$$=I_0I_1$$

Logic Circuit



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($\{S_0, S_1, ..., S_{M-1}\}$)

```
for all k = 0 : L − 1 do
   for all i = 0 : M − 1 do
      if (S_i(k) == 1) then
         add cube BR(i) to SOP for the k-th output of G
      end if
   end for
end for
```

Data Compression Flow

Compression Flow

Binary data (N_o bits)

$\Downarrow$ **Partitioning**

Paritioned binary data (M sub-blocks long $|B|/M$ each)

$\Downarrow$ **SOP Description Algorithm**

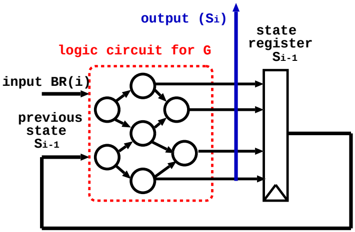
G Function Description

$\Downarrow$ **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:
- $F_n = (\varphi^n - \psi^n)/\sqrt{5}$ 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 $F_n = (\varphi^n - \psi^n)/\sqrt{5}$.

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 $\varphi = 1.6180339887...$ and $\psi = -1/\varphi$.

- Suppose we have knowledge about $S(i-1)$, $S(i-2)$.
- The decompression has a fixed sense $(S_0, S_1, S_2, ..., S_{M-1})$.
- 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($\{S_0, S_1, ..., S_{M-1}\}$)

```
for all k = 0 : L − 1 do
  for all i = 0 : M − 1 do
    if (S_i(k) == 1) then
      add cube BR(i) to SOP for the k-th output of G
      if (S_{i−1} is unique in {S_0, S_1, ..., S_{M−1}}) then
        add cube S_{i−1} to SOP for the k-th output of G
      end if
    end if
  end for
end for
```

S_{i-1} can be used as alternative (logical or with BR(i)) information to describe G

Improved Data Compression Flow

Improved Compression Flow

Binary data (N_o bits)

$\Downarrow$ **Partitioning**

Paritioned binary data (M sub-blocks long $|B|/M$ each)

$\Downarrow$ $BR(i)/S(i-1)$ **Description**

G Function Description

$\Downarrow$ **Multi-level Logic Synthesis**

Optimized logic circuit for G (N_c 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 enconding 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 bits)

⇓ **Partitioning**

Paritioned binary data (M sub-blocks long $|B|/M$ each)

⇓ $BR(i)/S(i-1)$ **Description**

G Function Description

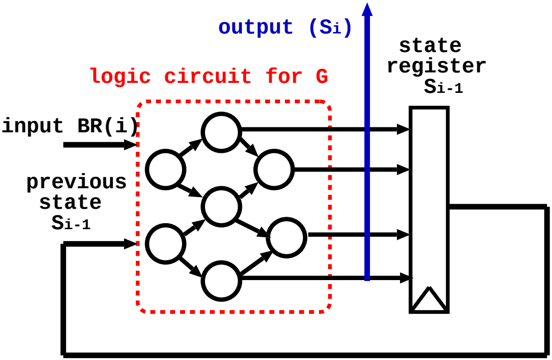
⇓ **Multi-level Logic Synthesis**

Optimized logic circuit for G

Entropy encoding of ⇓ **bits too hard to synthesize**

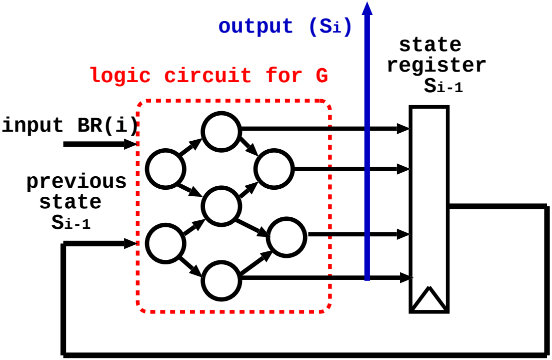
Compressed data (synthesis + entropy encoding results) (N_c bits)

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

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

Experimental Results: Memory Footprint

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

Data compression via logic synthesis presents best results.

Logic synthesis identifies the function correlating a data set.

Experimental Results: Memory Footprint

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

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^{st} place: ZIP.
- 2^{nd} place: bzip2 - 1.5×ZIP.
- 3^{rd} place: 7zip - 8×ZIP.
- 4^{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.

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.