Mayur Patel
Kai Zhao
Zain Siddiqui

ECE 594 Project 3

Results

- Test compaction is able to reduce the number of test vectors by 94.95%.
- Our compaction algorithm lost very few don’t cares (x’s). 78.38% of the remaining bits are x’s.
- Our compaction algorithm is better than ATALANTA’s in terms of number of test vectors. Our compaction algorithm had 228 test vectors while ATALANTA had 254 (average of 10 trials) test vectors.
- Compacting test vectors retains 100% detectable fault coverage.

Results Continued

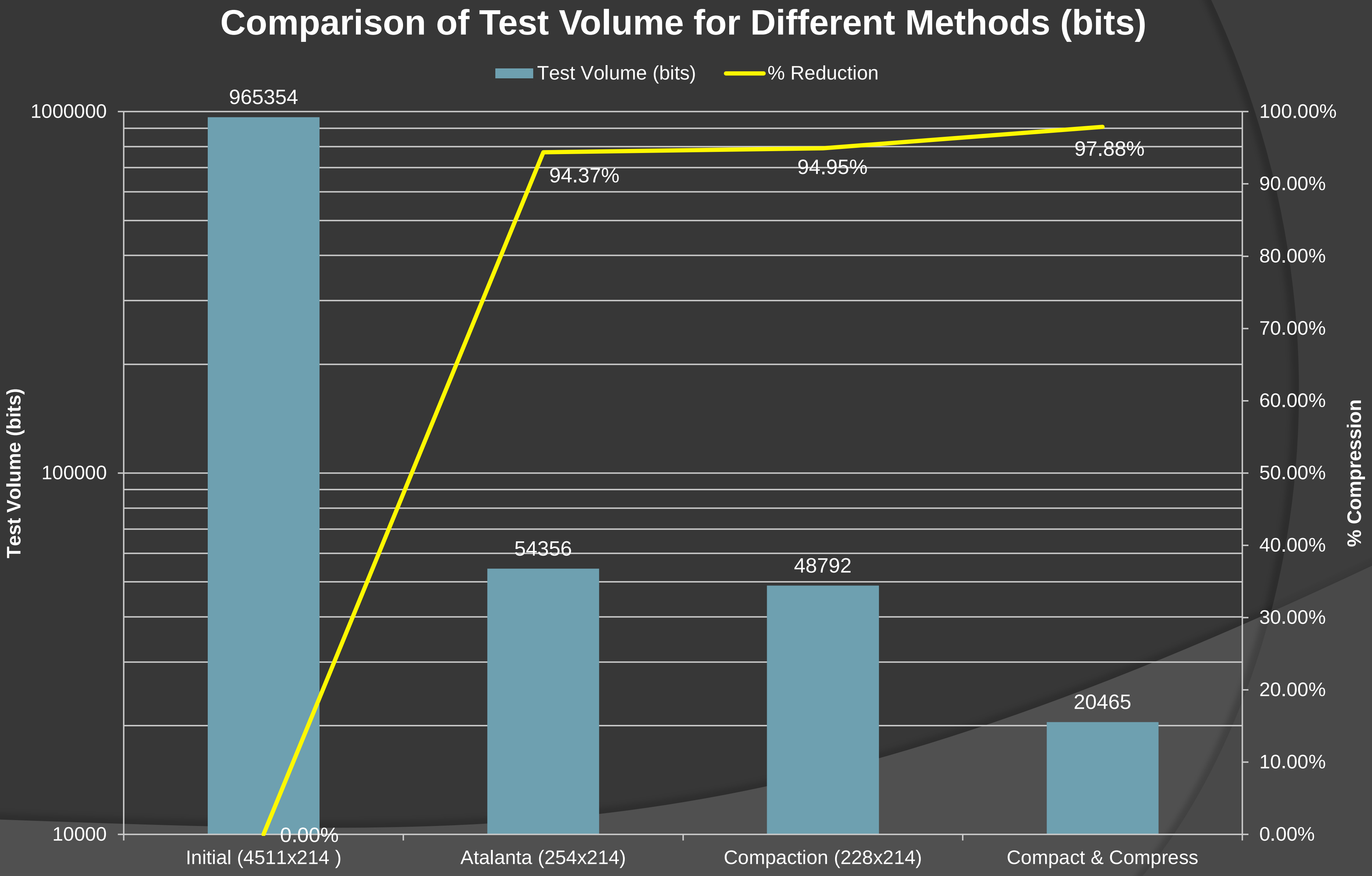
- We used vertical Huffman coding on our compacted test vectors and, on average, were able to represent 6 bits using only 2.5166 bits.
  - We picked 6 bits to do Huffman coding because that resulted in the best compression ratio for a short dictionary.
- We first replaced all the x’s in our test vectors with 1’s and then with 0’s. Replacing all x’s with 0’s results in a better reduction.
- Changing all x’s to 0’s saved 982 bits more than changing all x’s to 1’s.
- Our compression and compaction scheme combined resulted in a total of 97.93% reduction.
- A state machine could be used as a hardware de-compressor.

Challenges

- We started this project by attempting compaction from the fully specified test vectors generated from ATALANTA. We were able to save very few bits doing so.
- We tried horizontal Huffman Coding without any luck because there were equal frequencies of 0's and 1's.
  - The only factors of 214 (number of inputs for s5378) were 1, 2, 107, and 214. This made it nearly impossible to do horizontal Huffman code.
- We tried merging test vectors based on compatibility. However, we were losing too many don't cares after every merge.
  - It boiled down to 560 test vectors, which is twice as much as ATALANTA. This is when we decide to merge test vectors based on the number of x's that we have to specify for each merge.
- Then we tried removing all the don't cares on the right hand side.
  - This does reduce test time. However, this method was not as good as using vertical Huffman coding in terms of test volume.

Data

(we assume test time is proportional to test volume)



Huffman Decoding 3 Bit Example

1	0	1	0	0	0	1	0	1	1	0	0	0
0	0				1				0	0		
0	1				1				0	0		
0	1				1				0	1		

$$0.7949 * 1 + 0.041 * 3 + 0.0432 * 3 + 0.0413 * 4 + \cdots$$
$$= 1.56\ bits$$

We are able to represent every 3 bits using only 1.56 bits on average.

This 3 bit Huffman coding is an example only. The compression is better when using 6 bits. We were able to represent every 6 bits using only 2.52 bits.

Huffman Dictionary

Input	Output	Occurrence
1	000	79.49%
001	010	4.41%
000	001	4.32%
0111	100	4.13%
0101	111	2.88%
0100	011	1.72%
01101	110	1.71%
01100	101	1.34%

Technical Details Steps 1-3

- Step 1: Start with 4511 test vectors (965354 bits) from using atalanta-M -D 1 -t s5378.pat -W 2 s5378.bench
- Step 2: Import the 2330 unique test vectors (498620 bits)
- Step 3: Check if any test vector is a subset of any other. If so, then remove the least specified test vector.

  1529 test vectors (327206 bits) remaining.
  - E.g. if we have the following 2 test vectors
  - 1) x1x01x
  - 2) x1xx1x
  - then we can remove the second test vectors

Step 4

- Step 4: For every test vector, attempt to merge it with every other test vector.

  Keep track of the best possible merge (the merge that will require the least amount of x's to be specified) and the best possible merging partner.
  - E.g. if we have the following 4 test vectors
  - 1) 1x1x1
  - 2) 11010
  - 3) x1xxx
  - 4) xx101
  - then
  - 1) 1x1x1's best merge specifies 2 x's with either x1xxx or xx101.
  - 2) 11010's best merge specifies 4 x's with x1xxx.
  - 3) x1xxx's best merge specifies 2 x's with 1x1x1.
    4) xx101's best merge specifies 2 x's with 1x1x1.

Step 5

- Step 5: Compute the number of compatibilities for each test vectors.
  - E.g. if we have the following 4 test vectors
  - 1) 1x1x1
  - 2) 11010
  - 3) x1xxx
  - 4) xx101
  - then
  - 1) 1x1x1's compatibility is 2, since it is compatible with x1xxx and xx101
  - 2) 11010's compatibility is 1, since it is compatible with only x1xxx
  - 3) x1xxx's compatibility is 3, since it is compatible with 1x1x1, xx101, and 11010
  - 4) xx101's compatibility is 2, since it is compatible with 1x1x1 and x1xxx

Step 6

- Step 6: Take the test vector with the least number of x's specified for their best merge and the lowest compatibility (compatibility is a tiebreaker).

  Merge that test vector with its merging partner (merging partner is computed in step 4).
  - E.g. if we have the same following 4 test vectors
  - 1) 1x1x1
  - 2) 11010
  - 3) x1xxx
  - 4) xx101
  - then
  - 1x1x1, x1xxx, and xx101 all specifies 2 x's with their best merge. However, from those 3 test vectors, only 1x1x1 and xx101 have the lowest compatibility. Therefore, merge either 1x1x1 or xx101 with its merging partner, depending with whichever comes first.

Steps 7-9

- Step 7: If a merged has occurred, then go back to step 4.

  Else if no merge is possible, then move onto the next step.

- Step 8: 228 test vectors (48792 bits) remaining at the end of compaction and the beginning of compression.

  Replace all x's with 0's.

  38241 bit replacements had occurred.

- Step 9: Sort of test vectors in lexicographic (dictionary) order.

Step 10

- Step 10: Using a factor of the number of test vectors (6 in our s5378 case), do vertical Huffman coding. Count the number of occurrence of each word.

  Create a dictionary using frequency occurrence of each word (Huffman coding)

  Encode the words using the dictionary

  38 (228 divided by 6) lines of variable length (20465 bits) remaining

Steps 11-12

- Optional Step 11: (Increases % reduction to 97.93% (20015 bits), but NOT implemented due to decoding/decompression hardware complexity) Omit opposite bits at the end of each line.
  - E.g. given
  - 1) 1111111111111001111110010010001110011 0011111111
  - 2) 101011111111111111100111111111111011101111111111111111101110 11111
  - 3) 111111111101111110101000011111111001111111111111111111111111111110000001000111 0 00000
  - 4) 111111111111111001111 0011111
  - 5) 011111111110111111111111111111111111111111111111110 111111
  - Then omitting the opposite at the end will result in
  - 1) 111111111111100111111001001000111001100
  - 2) 101011111111111111100111111111111011101111111111111111101110
  - 3) 1111111111011111101010000111111110011111111111111111111111111111100000010001111
  - 4) 11111111111111100111100
  - 5) 011111111110111111111111111111111111111111111111110

  For decompression, add back in the opposite bits until there is enough input bits (214)
- Step 12: Print out all remaining test vectors.