Efficient Trace Data Compression using Statically Selected Dictionary

Kanad Basu and Prabhat Mishra
Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120, USA
email: {kbasu, prabhat}@cise.ufl.edu.

Abstract

Post-silicon validation and debug have gained importance in recent years to track down errors that have escaped the pre-silicon phase. Limited observability of internal signals during post-silicon debug necessitates the storage of signal states in real time. Trace buffers are used to store these states. To increase the debug observation window, it is essential to compress these trace signals, so that trace data over larger number of cycles can be stored in the trace buffer while keeping its size constant. In this paper, we propose several dictionary based compression techniques for trace data compression that takes account of the fact that the difference between golden and erroneous trace data is small. Therefore, the static dictionary selected based on golden trace data can provide notably better compression performance than the dynamic dictionaries selected in the current approaches. This will also significantly reduce the hardware overhead by reducing the dictionary size. Our experimental results demonstrate that our approach can provide up to 60% better compression compared to existing approaches, while reducing the architecture overhead by 84%.

I. Introduction

Increase in System-on-Chip (SoC) design complexity has made design verification an essential step before a chip is released in the market. However, with the time-to-market constraints, it is not possible to spend a lot of time in this process. The verification must be foolproof and should be accomplished as fast as possible. Pre-silicon debug has been a popular method for identifying the design errors before the chip is actually fabricated. Formal verification and extensive simulation are the two main approaches of pre-silicon verification. However, many physical parameters cannot be modeled correctly in early stages; as a result, pre-silicon validation and debug fail to actually detect and fix all the errors.

Manufacturing tests are used to detect manufacturing defects, like shorts and opens. However, they are not designed to capture any functional bugs that might have escaped the pre-silicon phase. Post-silicon validation techniques are used to capture these design errors that are present even after the chip is fabricated.

Post-silicon debug operates on a fabricated chip. Therefore, it is not possible to record the values of each and every internal signal. Various techniques have been proposed to observe some selected internal signals that will aid in the debug process [1], [2], [11]. These internal signal states are stored in a trace buffer during execution. During debug, the trace buffer content is analyzed, where it is matched against a set of ideal values to check for possible errors.

The trace buffer has two parameters, width and depth. Width refers to the total amount of debug data that can be stored per cycle, while depth refers to the total number of cycles over which debug data is to be stored. In order to keep the trace buffer size constant, while increasing the amount of trace data that can be stored, either the depth or the width has to be compressed. An efficient lossless trace data compression technique is necessary which can provide both fast compression and high compression efficiency, with minor impact on the architecture overhead. Different techniques of trace data compression, either by depth [3], [4] or by width [5] have been proposed. Depth compression approaches deal with selecting the cycles where the data are erroneous, and store the data for only those cycles. The problem with depth compression algorithms [3], [4] is that they assume rerunning the same set of tests on the same system produce the same output, that is, repeatable; which may not always be true. It would be better if the optimized trace data can be generated by running the tests only once. Width compression [5] utilizes this observation and run the tests just once in order to generate the trace data and compress them.

We have proposed a lossless dictionary based width compression scheme that operates on real-time to compress the trace data. Unlike [5], our method chooses the dictionary offline, which provides a better compression performance as well as huge reduction in compression architecture overhead. Three different compression algorithms have been proposed to trade-off between compression performance and architecture overhead.

We have used *Compression Ratio*, defined in Equation 1, as a metric to measure the efficiency of a compression algorithm. A higher compression ratio implies a better compression.

$$Compression\ Ratio = \frac{Uncompresssed\ Data\ Size}{Compressed\ Data\ Size} \tag{1}$$

The rest of the paper is organized as follows. Section II describes the related works in post-silicon debug and compression field. Section III describes our trace data compression techniques. Section IV describes the experimental results. Section V concludes the paper.

II. Related Work

The primary problem concerning post-silicon debug is the limited observability of the internal signals. Once the values of signals are known, they are analyzed using some algorithms like failure propagation tracing [6] to identify the errors in the circuit.

In order to obtain real-time observability of the internal chip signals, the obvious choice would be to use chip pins to observe them [7]. However, this method is difficult to implement in the presence of high frequency internal clocks and limitation of the chip pins. As a result, trace buffer based debugging techniques have been proposed [8] to counter this problem. In these techniques, the Embedded Logic Analyzer (ELA) acquires the signal sample and stores them in an on-chip trace buffer. The data from the trace buffer

* This work was supported in part by NSF grants CAREER-0746261 and CNS-0915376.

is then loaded to a processor via low bandwidth interface, which analyzes them to find out the error in the circuit. Since the trace buffer is used only for debugging, it is better to keep its size as small as possible to reduce the overall cost, area and energy requirements. Thus, to increase the amount of data that can be stored in a trace buffer, trace compression techniques have been proposed [3], [4], [5], which compress the trace data before storing them into trace buffer.

In order to compress the trace data, a compression technique is required which can provide a good compression with very low architecture overhead. In general, complex statistical algorithms produce best possible compression but introduce huge area overhead. On the other hand, simple dictionary-based techniques introduce acceptable area overhead but sacrifices compression efficiency. The dictionary based compression scheme was improved to remember mismatches with the help of bitmasks by Seong et al. [9] for code compression in embedded systems. Bitmask based compressions were also used in [10] for test compression. In this paper, we have explored the standard dictionary based compression, bitmask-based compression, as well as a variation of the BSTW compression scheme to provide much better performance in terms of compression and area overhead compared to existing approaches.

III. Trace Data Compression

The existing compression techniques compress the trace data by selecting a dictionary dynamically during execution. This not only results in inferior compression performance (due to non-optimal dictionary selection), but also increases the architecture overhead. This section describes our trace data compression techniques. The overview is shown in Figure 1.

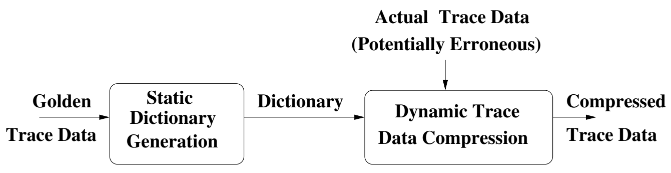

Fig. 1. Overview of our trace compression procedure

Our approach is based on an important observation. In any post-silicon debug environment, after the trace data is collected from the chip, it is validated by checking with a set of ideal trace data, that is obtained from a golden model. Since very few (2-5%) bugs actually remain to be tracked during the post-silicon debug phase, there are a few cycles which produce erroneous values [3], [4], that are different from the ideal ones. We utilize this information to design our approach. Since the difference between the ideal and the actual trace data is very small, the same dictionary applicable for ideal trace data compression can be reused for compression of the actual trace data. This takes care of the two problems by providing a better compression performance, and reducing the architecture overhead as well[1]. These compressed data are then read out through a channel to a debugger, where they are checked against the ideal trace data. Any discrepancy in the trace data is reported as error. As can be seen from our analysis in Section III-C, introduction of 2-5% error in trace data results in 2-6% penalty in compression performance, which is acceptable. It can be seen from the discussions in Section IV, even with the introduction of errors, our technique provides less compression penalty compared to the methods described by [5]. The remainder of this section describes our dictionary selection algorithms and also performs a theoretical analysis of the maximum penalty possible when the dictionary from the ideal trace data is used to compress the actual (potentially erroneous) trace data.

[1]no need to implement a dynamic dictionary selection algorithm

A. Dictionary Selection Algorithms

We have explored three compression algorithms for compression of the trace data, namely Dictionary based compression (DC), Bitmask based compression (BMC) and fixed Dictionary MBSTW (fMBSTW) based compression. All these three techniques use a dictionary for compression. The dictionary selection is extremely vital since it would be reused to compress the actual trace data. We will now describe how the dictionaries are selected in order to achieve the maximum compression performance.

1) Dictionary based compression (DC): Algorithm 1 outlines the dictionary selection method. In a dictionary based compression, the main aim is to include in the dictionary all the unique entries which have maximum repetitions in the dataset. Therefore, the first step determines all the unique entries in the dataset. We then find the number of repetitions for each entry. The unique entries are sorted in a descending order of the number of repetitions. The entries with the highest number of repetitions are included in the dictionary.

Algorithm 1 Dictionary selection algorithm for DC

```
$M = Number\ of\ unique\ entries$
$N = Number\ of\ Dictionary\ Entries$
$DIC = Dictionary$
for $each\ entry\ in\ M$ do
    Calculate the number of repetitions in the entire dataset
end for
Sort the $M$ entries in decreasing order of repetition count
Include the first $N$ entries in $DIC$
```

2) Bitmask Based Compression (BMC): The dictionary selection for bitmask based compression follows the same trend as the dictionary based compression, that is, select dictionary entries giving the maximum savings. However, there is a minor difference between the two. While savings for DC corresponds to just the repetitions, for BMC it includes those due to bitmask based matchings as well. Hence, the savings for each unique entry should be calculated based on the direct as well as bitmask based matches. The entries are then sorted in order of savings and included in the dictionary. The dictionary selection algorithm is shown in Algorithm 2.

Algorithm 2 Dictionary selection algorithm for BMC

```
$M = Number\ of\ unique\ entries$
$N = Number\ of\ Dictionary\ Entries$
$DIC = Dictionary$
for $each\ entry\ in\ M$ do
    Calculate the savings due to repetition and bitmask based
    matching in the entire dataset
end for
Sort the $M$ entries in decreasing order of total savings
Include the first $N$ entries in $DIC$
```

3) Fixed Dictionary MBSTW compression (fMBSTW): The compression technique for fMBSTW algorithm follows the same technique as MBSTW compression [5]. The difference from MBSTW is that the dictionary is selected statically and the number of dictionary entries is limited. We would now explain the dictionary selection steps for fMBSTW in Algorithm 3. This algorithm is shown for a 2-fMBSTW (2-strings are encoded together, similar to 2-MBSTW). This can be further extended to 3-fMBSTW, where 3 strings are encoded together.

Algorithm 3 Dictionary selection algorithm for fMBSTW

```
$M = No.\ of\ unique\ entries$
$N = No.\ of\ Dictionary\ Entries$
$DIC = Set\ of\ Dictionaries$
$first\_entry = last\_entry = NULL$
Create a 2-tuple for each pair of entries in $M$
for each 2-tuple do
    Calculate the savings across the entire dataset assuming only this tuple is in the dictionary
end for
Find the 2-tuple with the highest savings and add it to $DIC$
$first\_entry$ = first entry of 2-tuple
$last\_entry$ = last entry of 2-tuple
$N = 2$
while Size of $DIC$ less than $N$ do
    find the 2-tuple that starts with $last\_entry$ and produces maximum savings
    if such a 2-tuple exists then
        Include the 2-tuple in $DIC$, $N = N + 1$
    else
        Find any 2-tuple (not containing an entry already in $DIC$) which has the highest savings and include it in $DIC$
        $last\_entry$ = last entry of the 2-tuple, $N = N + 2$
    end if
end while
```

Figure 2 shows an illustrative example for dictionary selection using Algorithm 3. In this example, the strings in the trace data are represented using p, q, r, s, t. The amount of savings for each 2-tuple is shown in Figure 2. We want to have a dictionary of size 4. As can be seen, the highest savings is obtained from the 2-tuple $< r, s >$. Both of these are now included in the dictionary. The $last\_entry$ is s here. Now, we proceed to see which 2-tuple with the first entry s has the maximum savings. $< s, p >$ is selected as the 2-tuple and included in the dictionary. When searching for the next 2-tuple, it is seen that $< p, r >$ gives the highest savings. However, r is already present in the dictionary. Hence, $< p, r >$ is avoided. The 2-tuple having the next highest savings is $< p, t >$. Therefore, t is selected for the dictionary. In this way, the dictionary is built up.

Tuple	<p,q>	<p,r>	<p,s>	<p,t>	<q,p>
Savings	12	25	4	17	11

Tuple	<q,r>	<q,s>	<q,t>	<r,p>	<r,q>
Savings	12	12	7	19	14

Tuple	<r,s>	<r,t>	<s,p>	<s,q>	<s,r>
Savings	28	21	15	9	12

Tuple	<s,t>	<s,p>	<s,q>	<s,r>	<s,t>
Savings	12	14	7	9	11

r
s
p
t

Savings for each tuple Final Dictionary

Fig. 2. Example of dictionary selection in fMBSTW

B. Dynamic Trace Data Compression

Our final goal is to debug the DUT, for which we need the trace data from it. Application of a set of tests produces the trace data from the DUT which are compressed to reduce the size of the trace buffer. The overview of the compression architecture is shown in

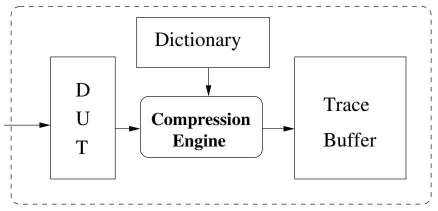

Fig. 3. Actual Trace Data Compression

Figure 3. As can be seen, the compression architecture consists of two parts, the dictionary and the actual compression engine. Depending on the design and associated constraints, a specific compression algorithm and its respective dictionary is used. For example, when BMC is most suitable for a design, the compression engine will have BMC in it and the dictionary will be the one selected for BMC. It should be noted, that the dictionary size is fixed here and not variable as in the case of dynamic dictionary selection [5]. Actually, [5] tried to include every single unique string in the dictionary. This increases the dictionary size, thereby introducing significant architecture overhead and also degrades the compression performance (since the number of bits used to index the dictionary increases with an increase in dictionary size). Our approach eliminates these disadvantages by keeping a limited number of profitable entries in the dictionary.

C. Performance Analysis with Erroneous Trace Data

Our approach is promising due to use of statically selected dictionary. However, this dictionary will be used to compress actual (potentially erroneous) trace data. This section analyzes our procedure and determines the performance degradation that may occur when the dictionary obtained from ideal trace data is used to compress the actual trace data. We have kept the trace data length constant at 32 bits. We introduce a term *compression penalty*, which is the ratio of the number of extra bits needed for compression when error is introduced, compared to the original trace data length. Obviously, a lower compression penalty signifies less number of bits needed to accommodate the error, and hence, a better compression performance.

$$Compression\ Penalty\ (CP) = \frac{Number\ of\ extra\ bits\ needed}{Size\ of\ original\ trace\ data}$$

We first analyze the compression penalty for DC and BMC. Next, similar analysis is performed for fMBSTW.

1) Compression Penalty for DC and BMC: We try to obtain the compression penalties for the two methods DC and BMC. In this section, we make two important observations.

Theorem 1: When statically selected dictionary (based on golden trace data) is used, the compression penalty is bounded by the percentage of error introduced in the actual trace data.

Proof: Let there be x strings in the original trace data. Let the percentage of error in case of actual trace data be l, expressed as a fraction ($l < 1$). The introduction of error changes $l \times x$ strings. In the worst case, all these $l \times x$ strings will be among the strings originally compressed, and these will now be uncompressed due to contamination. Let the number of bits required to compress the rest (that is $(1 - l) \times x$ strings) in the dataset be M[2]. It should be noted that these strings are not affected due to error injection and hence, the value of M remains constant in both cases. Let the number of dictionary entries be 2^d, so that d bits are needed to represent the

[2]Some of the strings may be compressed, while the rest uncompressed

dictionary. The $l \times x$ strings were compressed in the ideal case using $(1+d)$ bits each. If y_{ideal} be the number of bits after compression for the ideal trace data, it can be rewritten as,

$$y_{ideal} = M + l \times x \times (1+d) \qquad (1)$$

Now, let's analyze the actual trace data. In the worst case, all of the $l \times x$ strings remain uncompressed. Each of these strings will require 33 bits[3] to be represented. The M bits required to represent the $(1-l) \times x$ strings will remain the same. If y_{faulty} is the number of bits needed to represent the strings now, it can be represented as

$$y_{faulty} = M + l \times x \times (33) \qquad (2)$$

which implies,

$$y_{faulty} = y_{ideal} + l \times x \times (32-d) \qquad (3)$$

Therefore, number of extra bits needed, represented as y_{extra}, is

$$y_{extra} = y_{faulty} - yideal = l \times x \times (32-d) \qquad (4)$$

If CP_{DC} is the compression penalty for DC, then from the definition,

$$CP_{DC} = \frac{l \times (32-d)}{32} \qquad (5)$$

As can be seen CP_{DC} is always less than l, and hence is bounded by it. ∎

For example, with 8 dictionary entries, we have $d = 3$, and assuming the error rate is 5%, (which is the maximum error rate in these scenarios [3], [4]), we get

$$CP_{DC} = 4\% \qquad (6)$$

Thus, we see that a very slight compression penalty is introduced in DC even in the worst case. It can be seen from Equation (5) that increase in dictionary size can lessen this degradation.

Theorem 2: Compared to the ideal case (if dictionary was selected using erroneous trace data), the compression penalty using statically selected dictionary (using golden trace data) will be bounded by the twice the percentage of error.

Proof: We would like to see if the actual trace data were compressed without the help of ideal dictionary, how much compression would be obtained. In this case, the dictionary entries might differ from the ideal dictionary. If n is the extra number of strings that can be compressed in the actual case and m is the number of strings that were compressed in the ideal case, then the total number of strings compressed are $m + n - l \times x$. It is obvious that the maximum value of n can be $l \times x$, otherwise, these new strings would have been compressed in case of ideal trace compression, that is, these new strings would have been represented in the ideal dictionary. Therefore, the maximum number of strings compressed is m, which is the same case as in golden trace data.

As an example, consider a hypothetical trace data set of 20 entries. Suppose we choose the best 2 entries in the dictionary, each of which can compress a total of 5 entries. Therefore the total number of compressed entries will be 10. Corresponding to the symbols described above, $x = 20$, $m = 10$ and $d = 2$. Let the error rate be 10%, that is $l = 0.1$. When error is introduced, the number of strings contaminated is $l \times x$, that is, 2. In the worst case, both these strings were part of m and are now left uncompressed due to errors. The number of compressed strings now are $m - l \times x$, which is equal to 8. Now, if we try to compress these erroneous data with a different set of dictionary, let the number of extra strings being compressed be n. It is obvious that if n is greater than 2, the new dictionary would have been selected in the first place, so that the value of m would be different. So, the maximum value of n is bounded by $l \times x$.

However, in the best case, these contaminated strings can be all compressed using some other entry, which is not part of the dictionary now. Let us reiterate our previous example to explain this. For example, all of the $l \times x$ contaminated entries can be compressed using some other entry. Now, if that entry has high enough frequency, it will be included in the dictionary. In this example, the maximum frequency (original, without contamination) that an entry can have is 5; otherwise, it would have been included in the original dictionary. Therefore, the maximum number of strings that can be compressed with the new dictionary is $m + l \times x$, that is, 12 in this case. Hence, the maximum number of strings that can be compressed extra using the dynamically selected dictionary is $(m + l \times x) - (m - l \times x)$, that is, $2 \times l \times x$, which means the difference in compression ratio should be $2 \times l$. Therefore, the difference in compression efficiency between the dictionary based on golden data and dictionary based on actual data, will be bounded by twice the error rate in the data. ∎

It can be noted that the analysis for BMC will be similar to DC. This is because, even for BMC, the worst case comes when some strings which were completely compressed (not using bitmasks) change to uncompressed due to error introduction.

2) Compression Penalty for fMBSTW: To find the compression penalty, we analyze the worst case condition for 2-fMBSTW here. The worst case scenario can be divided in two parts. The first part is similar to that of DC and BMC, that is, the worst part comes when some completely compressed strings become uncompressed. The second part of the condition is explained as follows. Suppose, two consecutive strings correspond to two consecutive dictionary entries a, b. Therefore, all the two strings will be compressed using the 11 prefix, followed by the dictionary entry corresponding to a. However, if either a or b gets contaminated by error, in the worst case, one of them is uncompressed and the other one gets compressed separately, which requires more bits to compress the trace data. We now investigate the compression penalty in this approach.

Let the error rate and the number of strings be l and x as before. Let d be the number of bits to represent the dictionary index. Therefore, the number of such strings changed is $l \times x$. Each of these string corresponds to a $< a, b >$ tuple which is broken due to perturbation. Before the introduction of error, the number of bits required to compress these is given as y_{ideal} in Equation (7)

$$y_{ideal} = l \times x \times (2+d) \qquad (7)$$

Here, 2 bits are needed to represent the prefix 11 and d bits for the dictionary index of a. After perturbation (of b), in the worst case, a is independently compressed as single bits using the prefix 01[4]. Therefore, the number of bits needed to represent are $(36+d)$[5]. There will be $l \times x$ such occurances. Therefore, the total number of bits needed to represent the erroneous tuples is given by y_{faulty} as

$$y_{faulty} = l \times x \times (36+d) \qquad (8)$$

As before, let M be the number of bits required to compress the other $(1-l) \times x$ strings. Since M is unchanged in either case, the number of extra bits needed, is given by

$$y_{extra} = y_{faulty} - y_{ideal} = l \times x \times 34 \qquad (9)$$

[3]32 bits (original size), plus one bit to indicate not compressed

[4]as discussed in Section III-A.3

[5]$(2+d) + (2+32)$, where $2+d$ bits are needed to compress a and $2+32$ bits are needed to represent the uncompressed string b

Therefore, the compression penalty is given by,

$$CP_{fMBSTW} = \frac{l \times 34}{32} \qquad (10)$$

With a 5% error rate we can see that,

$$CP_{fMBSTW} = 5.31\% \qquad (11)$$

Thus, even with an introduction of 5% error, the compression penalty is small. These analysis will be later verified with experimental results in Section IV-C.

IV. Experiments

We have compared the compression performance of our approach with the algorithms proposed by Anis et al. [5] (MBSTW and WDLZW). We have also investigated our compression performance when the number of dictionary entries are varied. We have shown that our methods require much less compression architecture overhead compared to those in [5]. Finally, in Section IV-C, we have also analyzed the effect of introduction of errors on compression ratio and validated the equations developed in Section III-C. We have applied all the algorithms on the 5 largest ISCAS 89 benchmarks.

A. Compression Performance

First, we compare the compression performance of our algorithms with the algorithms in [5] using the traces obtained from ISCAS 89 benchmark circuits. The traces were obtained by following the approach outlined in [11]. The results are reported in Figure 4. We have fixed the dictionary entry to be 8 in each of the two compression algorithms, DC and BMC. For MBSTW, we have used the 2-MBSTW algorithm[6]. For the fMBSTW algorithm, the number of dictionary entries is floored to the nearest integer which is a power of 2. It can be seen that the fMBSTW approach works best in all cases except $s38584$. This is because the traces of $s38584$ has very less number of unique entries. As a result, even with 8 dictionary entries, a large portion of the circuit can be compressed using DC. DC works better than MBSTW in most cases and worse only in some cases ($s9234$ and $s35932$). The reason for this is the large number of unique entries in those trace data, which are effectively captured by MBSTW, but not by the 8-entry dictionary used in DC. If the number of bits needed to represent the compressed data is analyzed, it can be seen that fMBSTW provides up to 60% reduction in compressed data size compared to MBSTW and 70% compared to WDLZW. WDLZW provides worst performance for almost all the benchmarks. The high redundancy in the trace dataset is responsible for its somewhat good performance in $s38584$ and $s38417$.

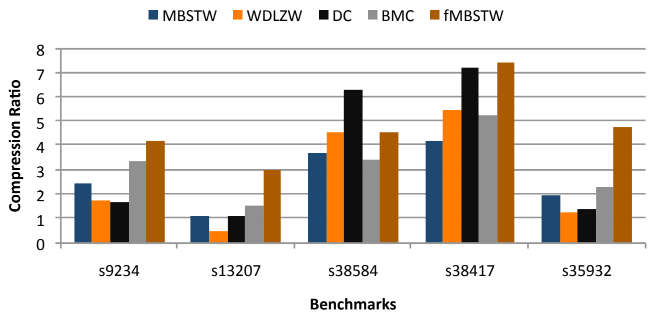

Fig. 4. Comparison of compression performance

Next, we vary the dictionary size of DC to see the effect on compression ratio. The results are shown in Figure 5. We have varied the number of dictionary entries from 8, 16, 32 and 64. As can be seen from Figure 5, the variation is not uniform for all the benchmarks. For s9234, s13207 and s35932, the compression ratio increases with increase in dictionary entries. On the other hand, for s38584 and s38417, increase in number of dictionary entries worsens the compression ratio and the optimal compression is achieved at 8 dictionary entries. Once we reach an optimal compression ratio, any increase in the number of dictionary entries will add to the total compressed data size both due to the increased number of entries in the dictionaries and increase in the number of bits representing the dictionary index.

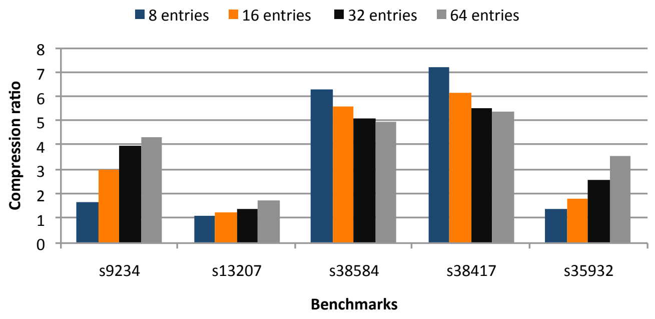

Fig. 5. Compression performance with dictionary entries

B. BRAM Requirement (Hardware Overhead)

The dictionary for compression has to be stored in on-chip 32-bit BRAMs. We have computed the total size of BRAMs needed for compression using each of these algorithms. Figure 6 compares the requirements for each of these approaches. It can be seen that since the dictionary size is always fixed (8 entries) for DC and BMC, the number of BRAMs required in these two algorithms is significantly less than any other approaches. WDLZW has the highest number of BRAM requirements since it captures all the double symbol repetitions (which worsens the compression performance). For fMBSTW, the number of BRAMs is kept floored to the nearest higher power of 2 for the number of unique entries in the stream[7]. From Figure 6, it can be seen that our two methods (DC and BMC) provides almost 96% less compression architecture overhead compared to MBSTW and almost 99% less than WDLZW.

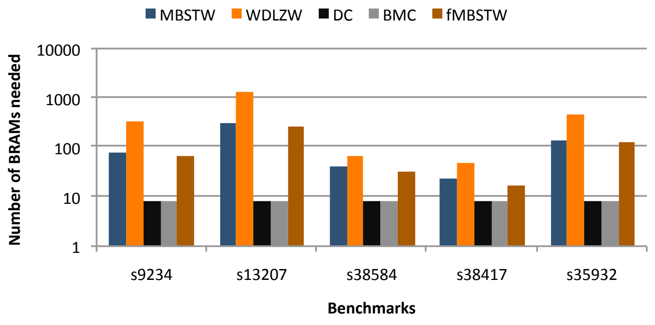

Fig. 6. BRAM requirements

It can be seen that there is a tradeoff between better compression ratio and lower architecture overhead. As can be seen from Figure 4 and Figure 6, either of the two techniques BMC or fMBSTW can be applied based on priority - BMC can be used for least area overhead (up to 96% reduction) with reasonable compression improvement (10%) compared to MBSTW, whereas fMBSTW should be used for best possible compression (up to 60%) while providing reasonable (up to 84%) reduction in BRAM requirement.

[6]Provides better performance than the 3-MBSTW algorithm

[7]Results in Figure 4 are also reported using this configuration

C. Compression Performance with Erroneous Trace Data

We now like to validate the analysis done in Section III-C. Errors have been inserted randomly at a rate of 2% to 10% in steps of 2% in the trace data, and the same is compressed using DC, BMC and fMBSTW. Figure 7 shows the comparison of compression penalty in DC with varying percentage of error. It can be seen that the change in compression penalty complies with Equation (5) in Section III-C. For example, putting a value of $l = 2\%$ in Equation (5) will result in a compression penalty of less than 2%, which matches in the figure for all the benchmarks. We have conducted similar experiments for BMC based compression technique as well. The results in Figure 8 shows that the compression penalty also follows Equation (5).

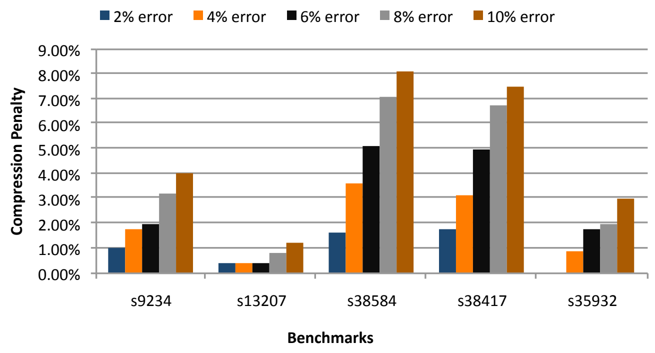

Fig. 7. Comparison of compression penalty for DC

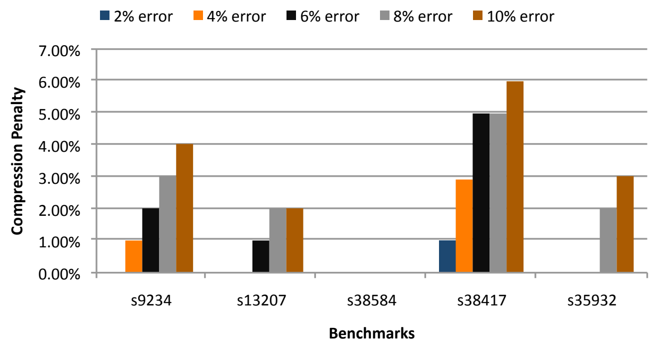

Fig. 8. Comparison of compression penalty for BMC

Now, we would like to verify the last part of the discussion in Section III-C, that is, the change in compression penalty with error rate for fMBSTW. We have conducted similar experiments and the results are shown in Figure 9.

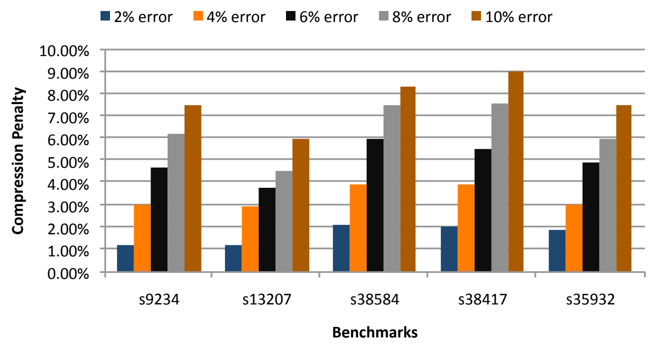

Fig. 9. Comparison of compression penalty for fMBSTW

An important observation here is that the change in penalty is sharper than the case of Figure 7 or Figure 8. This is quite obvious as per the discussion in Section III-C, since in fMBSTW, 2 strings are affected when an error is introduced, whereas in DC or BMC, only 1 string is affected.

Finally, we compare how the introduction of errors affect the compression performance in cases of MBSTW and fMBSTW. The results are shown in Figure 10. We have introduced 2% error for every benchmark's trace data. It can be seen that the compression penalty obtained using fMBSTW is always less than MBSTW, the maximum difference being 4% for $s38417$. The reason for higher penalty in MBSTW is that if error gets introduced early, MBSTW cannot benefit from a profitable sequence. In summary, our approach (fMBSTW) will perform significantly better irrespective of the percentage of errors in the dataset.

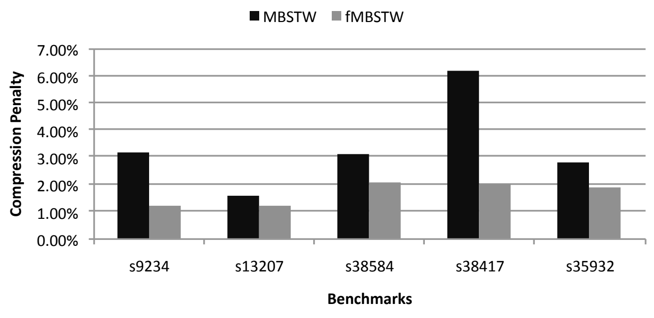

Fig. 10. Comparison of compression penalty

V. Conclusions

Post-silicon validation is extremely complex and time consuming in overall design methodology. To aid in debug, trace data obtained from the chip are stored in the trace buffer. However, the trace buffer size is limited due to area/cost constraints. Trace data compression schemes have been popular which deals with dynamic dictionary based compression that enables to store larger traces. We have proposed a trace data compression technique, which employs a statically computed dictionary. We have used three compression algorithms for compressing the trace data. Our approaches can produce up to 60% better compression performance, and reduce the compression architecture overhead up to 84% compared to best-known existing approaches.

References

[1] H. Ko and N. Nicolici, "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug," *IEEE TCAD*, vol. 28, no. 2, pp. 285–297, Feb. 2009.

[2] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *DATE*, 2009.

[3] J. Yang and N. Touba, "Expanding trace buffer observation window for in-system silicon debug through selective capture," in *VTS*, 2008.

[4] E. Anis and N. Nicolici, "Low cost debug architecture using lossy compression for silicon debug," in *DATE*, 2007, pp. 225–230.

[5] ——, "On using lossless compression of debug data in embedded logic analysis," in *ITC*, 2007, pp. 1–10.

[6] O. Caty, P. Dahlgren, and I. Bayraktaroglu, "Microprocessor silicon debug based on failure propagation tracing," in *ITC*, 2005, pp. 10–293.

[7] B. Vermeulen and S. Goel, "Design for debug: Catching design errors in digital chips," *IEEE Des. Test*, vol. 19(3), pp. 37–45, 2002.

[8] R. Leatherman and N. Stollon, "An embedded debugging architecture for socs," *IEEE Potentials*, vol. 24, no. 1, pp. 12–16, February 2005.

[9] S. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," *IEEE TCAD*, vol. 27(4), pp. 673–685, April 2008.

[10] K. Basu and P. Mishra, "Test Data Compression Using Efficient Bitmask and Dictionary Selection Methods," *IEEE TVLSI*, vol. 18, no. 9, pp. 1277–1286, 2010.

[11] K. Basu and P. Mishra, "Efficient Trace Signal Selection for Post Silicon Validation and Debug," in *International Conference on VLSI Design*, 2011.