Cache Aware Compression for Processor Debug Support

Anant Vishnoi, Preeti Ranjan Panda, and M. Balakrishnan
Department of Computer Science and Engineering, Indian Institute of Technology Delhi
Hauz Khas, New Delhi 110016, India
Email:{anant,panda,mbala}@cse.iitd.ac.in

***Abstract*—During post-silicon processor debugging, we need to frequently capture and dump out the internal state of the processor. Since internal state constitutes all memory elements, the bulk of which is composed of cache, the problem is essentially that of transferring cache contents off-chip, to a logic analyzer. In order to reduce the transfer time and save expensive logic analyzer memory, we propose to compress the cache contents on their way out. We present a hardware compression engine for cache data using a *Cache Aware Compression* strategy that exploits knowledge of the cache fields and their behavior to achieve an effective compression. Experimental results indicate that the technique results in 7-31% better compression than one that treats the data as just one long bit stream. We also describe and evaluate a parallel compression architecture that uses multiple compression engines, resulting in a 54% reduction in transfer time.**

I. Introduction

The increasing complexity of modern processors is accompanied by a corresponding increase in the complexity of their debug procedure. During post-silicon processor debugging we need to capture the internal state of the processor for further analysis. In a typical processor testing scenario millions of test cases are run with the internal state being dumped off-chip at regular intervals through a logic analyzer. When a failure is detected the processor is re-run from the previously dumped correct state and the internal state is now dumped at smaller intervals [1]. The process continues until external analysis isolates the problem. This requires a large amount of expensive logic analyzer memory for saving the internal state and costs a significant amount of valuable debug time in transferring the internal state. Since internal state constitutes all memory elements in the processor, the bulk of which is composed of cache, the problem in transferring the state is essentially that of transferring the cache contents off-chip.

We present a novel cache compression engine to compress the cache contents with the intention of saving state transfer time and logic analyzer memory, as shown in Figure 1. The engine compresses the cache contents on the way to the logic analyzer, where it is stored in compressed form. The compressed state is later downloaded from the logic analyzer and decompressed offline for further analysis. Our *Cache Aware Compression* technique compresses the cache contents by exploiting the knowledge of the cache architecture. Due to locality of reference in code and data, we observe correlation within the cache fields such as Tag, ECC, Control bits, etc. Our proposed cache aware compression compresses the different fields separately for higher compression. We selected a variant of LZW compression method for the cache compression hardware. In order to reduce the latency overhead caused by the compressor, we have designed LZW-SLU, an efficient hardware implementation of LZW. Furthermore, since most of the compression time is consumed by the bulky data field of the cache, we have proposed a parallel compression methodology to reduce the overall dump time.

This work was partially sponsored by a research grant from Intel

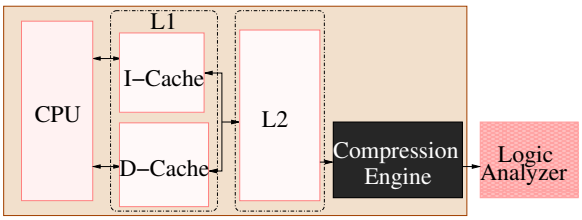

Fig. 1. Compression of Cache Data

II. Related Work

Related research can be classified into three categories: (i) Compression in processor debug/test, (ii) hardware implementation of data compression algorithms, and (iii) cache/memory compression for better performance and power.

A large body of work already exists in the area of test compression. Balakrishnan et al. [2] address the problem of faster loading of test vectors. Their proposed improvement uses a software compression mechanism to improve the loading time. Anis et al. [3] proposed a signature based technique to enhance the post silicon debug process. Cheng et al. [4] propose a test data compression technique, CacheCompress, to compress scan chain data. The above line of research is orthogonal to our proposed idea.

Many hardware compression techniques have been reported in the past targeting different domains. We first review the main compression techniques implemented in hardware, then compression techniques that target memory and cache. X-Match [5] is a dictionary based compression algorithm using a move-to-front strategy with Huffman coding. IBM's MXT (Memory Extension Technology) [6] uses Parallel Block-Referential Compression with directory sharing, a parallel derivative of the Lempel-Ziv algorithm [7]. Lin et al. [8] proposed a parallel architecture for LZW compression. Content Addressable Memory (CAM) based compressor implementations have been reported in [9], [10].

Several memory compression techniques target the improvement of system performance through reduced memory traffic [11], [12]. Because compression and decompression are performed on-the-fly needing low latency, the compression

978-3-9810801-5-5/DATE09 © 2009 EDAA

ratios achieved are relatively low. Lekatsas and Wolf [13] used Semi Adaptive Markov Compression (SAMC) to store compressed code. The techniques used in [14]–[17] attempt to fit data into a smaller cache space using compression, giving the illusion of a larger cache. The technique in [14] compresses swapped-out virtual memory pages. The FPC cache compression algorithm [15] attempts to find patterns in cache line data and compresses it using a static coding technique. Lee et al. [16] proposed a compressed memory hierarchy model that selectively compresses L2 cache and memory blocks. Lekatsas et al. [17] use SAMC to compress the cache.

Most previous work on memory and cache compression focused on performance and power improvement, which is orthogonal to our targeted scenario. Since, in our case, decompression is not performed online, we can use more aggressive compression strategies. To the best of our knowledge, there is no prior research on the specific problem scenario we have outlined.

III. Cache Aware Compression

The knowledge that our input data has a cache structure allows us to exploit data correlation within cache fields (Tag, Control, Data, and ECC) to achieve better compression. We extract the fields and compress them separately, as shown in Figure 2.

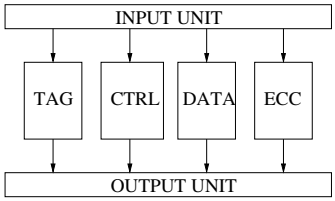

Fig. 2. Utilization of Cache Architecture

A. Tag Field

Spatial locality of memory references often causes the Tag fields (higher order address bits) of consecutive cache lines to be identical or differ only in a few LSB bits. This leads to significant compression opportunities when the tag sequence is compressed separately. Note that such possibilities are less explicit and may not be noticed by a compressor when the cache content is treated as just an unstructured byte stream.

B. Control Field

Each cache line is associated with a set of control bits such as Dirty, Reference, and Valid bits. We observed that these bits change less frequently and depend on the program segment (code, data) the cache line is representing. For example, if a cache line represents the code segment then the dirty bits may not change. Due to this correlation we separately compress the control bit fields.

C. Data Field

In a unified cache, depending on the program, the data field can be dominated by data or instructions or can be a mixture of both. To utilize such information, we propose the following two byte extraction methods.

1) Column-wise: It is well known that in applications dominated by integer data, the upper bytes of the integer variables change less frequently. Since most integers are of small magnitude, the higher order bits are usually all 0's or all 1's, depending on whether the integer is positive or negative. The stream of such bytes leads to high compression. If the corresponding higher order bytes are concatenated, compression possibilities are enhanced. This is illustrated in Figure 3. We divide 32-bit words of the cache line into bytes, labeled 1,2,3,4, and form our byte streams for compression by concatenating all the 1's, 2's, 3's, and 4's separately.

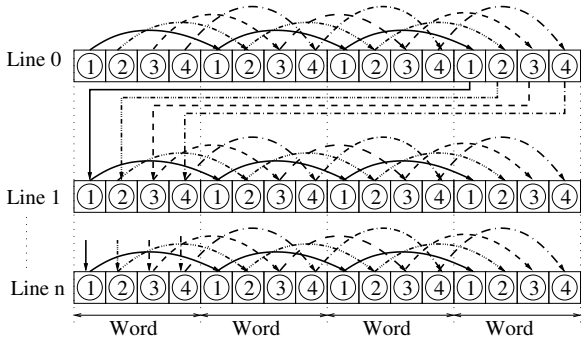

Fig. 3. Column-wise byte extraction

2) Row-wise: When the cache contents are different from integers, e.g., instructions, floating point, image data, etc, column-wise byte streams do not help in compression. In such cases, the input for the Data field compressor could be composed as a row-wise byte stream, as shown in Figure 4.

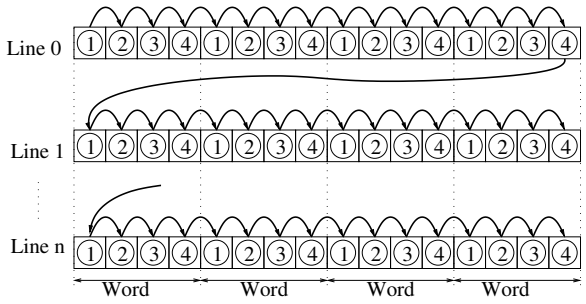

Fig. 4. Row-wise byte extraction

D. ECC Field

Error Correcting Code (ECC) bits are usually stored with each cache line and need to be dumped along with the other cache contents. We use a simple and effective compression strategy for the ECC field, which is based on the recognition that the ECC bits are redundant, unless there is actually an error. Since the ECC computation function is known to the compressor (and decompressor), we just re-compute the ECC bits for the cache line data in the compression engine. If the computed ECC matches the ECC stored in the cache, (which is expected for the most cache lines), we send a '0'. Since the de-compressor knows the ECC computation function, it can easily re-generate the ECC bits. If there is a mismatch, we send a '1' followed by the actual ECC bits. The generated bit stream is then compressed using a regular compression algorithm.

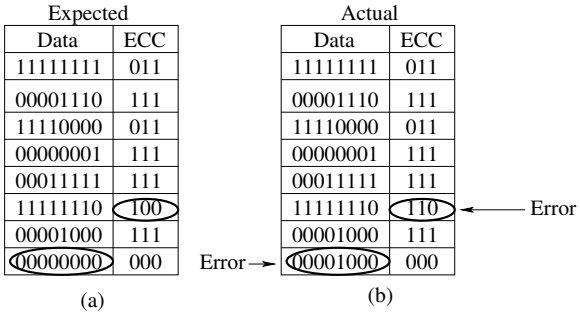

Fig. 5. ECC Example

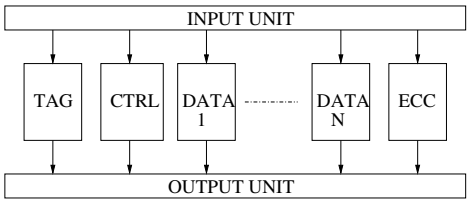

Fig. 6. Parallel Compression of data field

Example: In Figure 5, we have shown two sets of data with ECC bits. In the second set we have introduced errors in one of the ECC and data fields. We need to dump the whole ECC field, as shown below.

- 011,111,011,111,111,100,111,000

With our compression strategy we obtain the following output. The correct bits are replaced by '0' and error bits are prefixed with '1'.

- 0,0,0,0,0,0,1110,0,1000

Our compression strategy results in a saving of 10 bits (41.7%) in this example. The generated output stream is a promising candidate for further compression, since we expect long strings of '0's.

E. Parallel Compression

We observe that the compression time is dominated by the bulky data field of the cache. To reduce the compression time we explore architectures with parallel compression of the data field with multiple compressors, as shown in Figure 6.

IV. Hardware Design

Based on the functionality, the compression engine can be divided into following three sections: Input Unit, Compressor and Output Unit as shown in Figure 7.

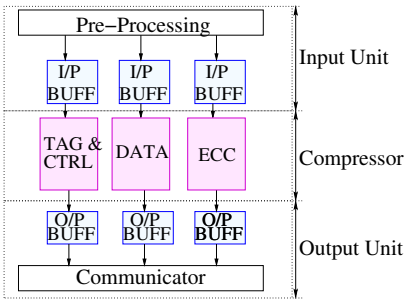

Fig. 7. Hardware Module

A. Input Unit

The Input Unit is responsible for communicating with the cache to receive the cache lines and segregate the line contents into different input buffers associated with the different compression engines. If necessary, different fields could also be merged and fed to the same compression engine. For example, in the cache configurations we studied, there were only 3 control bits; we could merge these 3 bits with the LSB bits of the Tag field (Figure 7). Not much difference was observed in the compression and this led to reduced area by dropping one compression engine.

B. Compressor

The Compressor is the heart of the compression engine. We observed that dictionary based LZW compression algorithm works well for cache data. The X-Match strategy is also powerful; a comparison is presented in Section V. For practical implementation of LZW, an appropriate dynamic dictionary update policy has to be defined. Since the dictionary space is limited, we would like to store only those entries that are likely to be referenced in the near future. Since, as mentioned earlier, we expect locality of reference in the input data, a Least Recently Used (LRU) policy is appropriate. The input string suffix is searched in the dictionary, and the index corresponding to the longest match is output. The dictionary entry structure of LZW is shown in Figure 8. The index field stores the pointer of the prefix string location, and the data field contains the string suffix. We have designed and evaluated two LZW implementations: LZW-SLU and LZW-DC.

Fig. 8. Dictionary Entry of LZW algorithm

LZW-SLU: We have designed a CAM-based circuit for a time-efficient realization of the dictionary search function. The CAM based LZW algorithm returns the address of the dictionary where the input data is stored. An ideal LRU implementation requires expensive counters associated with each CAM entry, so appropriate approximations are necessary. Ideas such as Pseudo-LRU (PLRU) have been used to implement cache associativity [18]. We present a comparison of our proposed technique with this in Section V.

We introduce an LRU approximation that uses two bits per entry: one bit for capturing recency of reference (L) and one bit representing updation, i.e., write operation (U). We call this approximation LZW with **S**ingle bit **L**RU and **U**pdate (LZW-SLU). When a dictionary entry is referenced the corresponding L bit is set to '1'. Similarly, when a dictionary entry is updated the U bit is set to '1'. To update the dictionary we search for the dictionary entry having L and U bits equal to "00". Since, the dictionary is built from top to bottom, the probability of the LRU entry lying closer to the top of the dictionary is high. When there is no entry with L,U bits = "00" then the L bits associated with all entries are copied to the corresponding U bits, and L bits are reset.

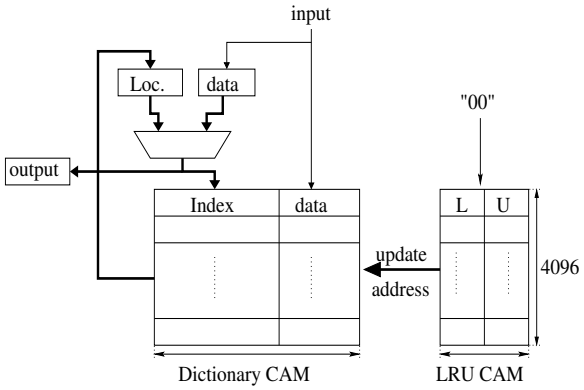

Fig. 9. LZW-SLU Architecture

LZW-SLU performs two simultaneous searches: (i) input string in the dictionary, and (ii) dictionary address to be updated if the string search fails. We have implemented LZW-SLU with CAM-based structures, shown in Figure 9. The first, a *Dictionary-CAM*, maintains the dictionary of the LZW algorithm. The second, an *LRU-CAM*, stores the L and U bits. If there are multiple matches in the LRU CAM, the first matching address is returned using the CAM's priority encoding mechanism. A simple modification of the generic CAM cell was necessary to achieve the simultaneous copying of the L to the U bits.

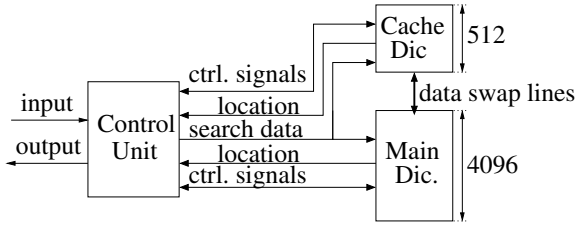

Fig. 10. LZW-DC Architecture Block Diagram

LZW-DC: To exploit locality in the input data, we propose a variant of LZW called LZW with Dictionary Caching (LZW-DC). LZW-DC stores recently referenced and LZW initialization data in the Cache Dictionary (CD) and the remaining data in Main Dictionary (MD). The cache dictionary being smaller than main dictionary, requires fewer bits to address it. Since LZW's compressed output is an address sequence, the output can be smaller if we hit in CD most of the time. The hardware structure is shown in Figure 10. CD and MD have 512 and 4096 entries, requiring 9 and 12 bits to address them. By using one additional bit (ID) to identify the dictionary, we require 10 and 13 bits addresses respectively. If the hit ratio in CD exceeds 33%, then the average number of bits in the ouput address sequence is less than 12.

The input string to LZW-DC is searched in both dictionaries simultaneously. If the search succeeds in CD, then there is no action. If the search fails in CD but succeeds in MD, we swap the MD entry with the LRU entry from CD, in anticipation of temporal locality for this entry, which would then hit in CD. In case of search failure we output the index of the last string suffix found in the dictionaries, prefixed by identifier ID. Following this, CD is updated with the string suffix, and the LRU entry of CD replaces the LRU entry of MD. The LZW-DC hardware implementation is based on LZW-SLU. Both dictionaries use the SLU approximation of LRU policy.

C. Output Unit

The Output unit buffers the output of the compressor and communicates with the logic analyzer. Whenever data is ready for dumping, it initiates the protocol to transfer compressed data to the logical analyzer. When the output buffer is empty, we prefix some identification bits to the compressed stream. These bits are required to identify the fields during construction of the cache from compressed stream. We omit the detailed working and FSM for Input and Output units due to lack of space.

V. Experiments

Our evaluation framework for dumping the cache state is built around the Simplescalar processor simulator [19] and Dinero cache simulator [20]. Appropriate modifications were made to facilitate the stopping of simulation at arbitrary times (representing the break points during debugging) and dump the cache tags, data, control, and ECC bits.

We generated experimental data by running applications from Mediabench and CPU-SPEC 2000 benchmarks on our framework. To create more realistic scenarios where one application runs after another, we also invoked the benchmarks sequentially as subroutines of the main program. All our experiments were carried out on the following L2 unified cache configuration: 512 KB, 4-way, 16-byte line, 3 control bits, 7 ECC bits. All compression results reported in this paper are the average compression ratios of all cache dumps taken periodically at intervals of 10 million instructions. We use the definition of compression ratio as the percentage of bytes saved, i.e., $(1 - o/i) \times 100\%$, where o is the number of output bytes and i is the number of input bytes.

A. Compression Results

Table I shows the compression ratios of all the cache components when they are compressed separately. The Tag and Ctrl. fields (Columns 3 and 4) show good compression, which validates our claim about the correlation within them in different cache lines. Columns 5 and 6 show compression ratios of the data field of the cache using row-wise and column-wise byte extraction respectively. The significant variation in compression ratio can be attributed to the different characteristics of applications and their data. The data field of Bzip2 does not compress much, as it being a compression algorithm, the cache contains both uncompressed and compressed data; the latter cannot be effectively compressed further. The Bzip2 example, which depends on input data, and Milc/Hmmer, which are applications using mostly integer data, give better compression with column-wise byte extraction. The ECC column shows the compression of the ECC field assuming error rate of 0.01%. We have above 96% compression for ECC field for every benchmark, because, as expected, there are long runs of 0's

resulting from our ECC compression strategy. The last three columns of Table I show the total compression ratio for the benchmarks. The 8^{th} and 9^{th} columns show the compression ratio when the data field is compressed row-wise and column-wise respectively. The last column shows the compression ratio when the cache content is compressed by considering it as a stream of bytes i.e, by ignoring the cache's internal structure completely. This case can be considered the baseline case for the comparisons. We obtained compression improvements of 7% to 31% with our methodology. The last row of the table shows the average compression of the various fields of the cache and the average total compression ratio. Row-wise processing of the Data field gives better results on an average, but it should be possible, as a future enhancement, to program the dumping mechanism with a hint indicating the type of data likely to be present in the cache.

Benchmark	*Comp. Algo.*	*Tag*	*Ctrl*	*Data (R-W)*	*Data (C-W)*	*ECC*	*Total (R-W)*	*Total (C-W)*	*Byte Stream*
CHESS	DC	58.60	97.21	79.43	75.11	96.98	78.15	74.62	62.38
	SLU	57.14	96.96	80.60	76.19	96.69	78.92	75.32	62.50
LBM	DC	90.85	97.19	88.22	84.11	96.96	89.22	85.86	73.04
	SLU	91.87	96.93	90.73	90.71	96.67	91.38	91.36	76.85
MCF	DC	77.52	97.22	53.61	49.14	96.96	59.39	55.74	40.01
	SLU	79.38	96.97	55.57	49.51	96.67	61.20	56.26	40.14
MILC	DC	74.22	97.22	**38.34**	**41.99**	96.96	**46.54**	**49.51**	**22.95**
	SLU	80.28	96.97	**35.45**	**42.20**	96.67	**44.91**	**50.41**	**18.34**
HMMER	DC	87.38	94.67	**52.39**	**54.79**	96.57	**59.53**	**61.49**	**42.34**
	SLU	88.55	94.57	**51.61**	**54.79**	96.27	**59.04**	**61.63**	**42.09**
BZIP2	DC	85.73	97.04	**18.52**	**21.26**	96.94	**31.77**	**34.00**	**12.93**
	SLU	87.47	96.80	**14.81**	**15.60**	96.65	**28.95**	**29.59**	**8.10**
ENC_ MPEG	DC	88.26	95.96	49.63	37.60	96.96	57.42	47.61	38.73
	SLU	89.25	95.75	50.09	35.92	96.67	57.91	46.35	38.18
LAME_ ENC	DC	83.16	94.99	41.48	39.10	96.93	50.14	48.20	34.35
	SLU	83.61	94.81	37.56	37.11	96.93	46.99	46.62	30.94
MPEG_ LAME_ENC	DC	85.80	96.34	41.97	32.95	96.94	50.88	43.53	32.33
	SLU	86.41	96.08	39.81	30.78	96.65	49.19	41.82	29.96
Average	DC	81.28	96.43	51.51	48.45	96.91	58.11	55.62	39.89
	SLU	82.66	96.20	50.69	48.09	96.65	57.61	55.49	38.55

TABLE I
COMPRESSION RATIO(%) OF CACHE FIELDS

Figure 11 shows a comparison of the compression performance of compression algorithms. In the interest of brevity we have reported only the average compression ratio of different configurations, where the data field is compressed in parallel with the number of parallel compression units varying in powers of 2, from 2 to 128. We have compared our LZW implementation and modification with the best known hardware compression algorithm X-Match and the popular Pseudo LRU [18], approximation of LRU policy for LZW algorithm. We notice that, except for Bzip2, X-Match gives worse compression results than other implementations of the LZW algorithms. Our LRU approximation implementation of the LZW algorithm, LZW-SLU, gives, on an average, 9.2% more than X-Match, and 1.2% more than PLRU. We also observe that LZW-DC gives slightly better (0.5%) compression than LZW-SLU.

Figure 12 shows the average compression ratio of all the benchmarks for LZW-SLU (row-wise). The best compression is observed for 4 parallel compression units. Beyond that, the overhead of the identification bits leads to worse compression. Similar trends are noticed for other compression algorithms.

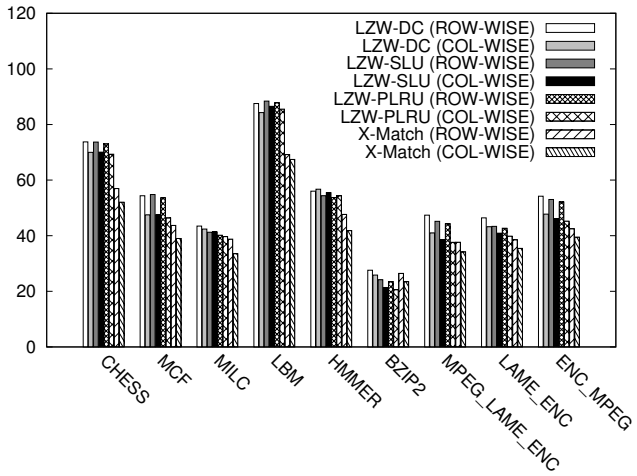

Fig. 11. Average Compression ratio

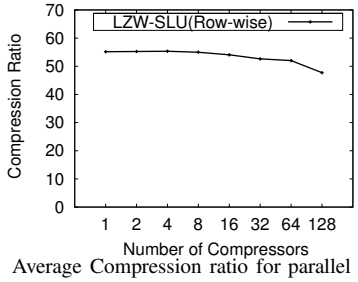

Fig. 12. Average Compression ratio for parallel compression

B. Implementation Details

We have designed our hardware implementation to be roughly compatible with Intel Pentium 4 class of processors – with 2 GHz clock, 32 bit bus, 533FSB, implemented on 180nm technology. Our implementation consists of VHDL models synthesized into a 180nm ASIC library with Synopsys Design Compiler. CACTI [21] was used to estimate the area and delay of memory components.

Table II shows a comparison between the area, critical path delay, CPU cycle, and the processing speed of compression algorithms. The comparison shows X-Match to be superior in area and delay. However, as seen in Figure 11, its compression ratio is worse. The computation of total dump time is a function of different parameters. This is discussed in Section V-C. We have used a 32 byte input buffer and 256 byte output buffer, The larger output buffer is necessary to reduce the overhead of the identification bits. At the same time, the buffer sizes cannot be too large because the unused space at the end of the output streams leads to overheads. The input and output units have only 0.5% impact on the area of the compression engine.

C. Transfer Time

Transfer time is an important metric for comparison of the hardware compression strategies. The scenario is shown in Figure 13. The time taken to transfer the cache contents with no compression is:

Compression Algorithm	*Area (mm2)*	*Critical Path(ns)*	*# CPU Cycle*	*Byte Processed*	*CPU cycles/ Byte*
LZW-DC	2.066	12.11	25	0.32	78.13
LZW-SLU	1.701	5.11	11	1	11
P-LRU	1.825	5.62	12	1	12
X-Match	.460	15.09	31	4	7.75

TABLE II
AREA, CRITICAL PATH AND PROCESSING SPEED

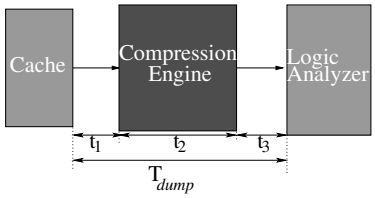

Fig. 13. Dump Time T_{dump},t_1, t_2, t_3

$$T_{dump} = \text{num\_bytes / bus\_speed}$$

For a cache with 512 KB, 4-way, 16 byte line, we have: T_{dump} = 6.12 $\times$ 10^5 CPU Cycles.

The compression engine compresses the data and communicates with the cache and the logic analyzer simultaneously. Therefore, when the compression engine is placed between the cache and the logic analyzer, the transfer time is determined by the slower of the two interfaces, i.e., the maximum of the following durations:

- t_1 Time to transfer data from Cache to Compression Engine; this is constant at 0.5 x10^5 CPU cycles
- t_2 Time to Compress Data

$$t_2 = \frac{\text{num\_bytes (Data Field)} \times \text{num\_cycles}}{\text{num\_comp\_engines} \times \text{process\_speed}}$$

- t_3 Time to transfer compressed data

$$t_3 = \text{num\_output\_bytes / bus\_speed}$$

Table III shows the effect on transfer time, of increasing numbers of parallel compression units using LZW-SLU and X-Match. Column 1 lists the compression algorithm. Column 2 shows the number of parallel compression engines. A monotonic increase is observed for both algorithms, in column 3, which shows the data rate at which compression engine compresses the data in parallel. Column 4 shows the rate at which the Bus can send data to the logic analyzer; this is constant. Column 5 shows the CPU cycles consumed by the Bus to transfer the compressed data. The compression ratio decreases with increase in number of compression engines due to overheads. The 6^{th} column, showing t_2 values, depends on column 4 and the data to be compressed. Compression time decreases with the number of compression units. The 7^{th} column shows the transfer time, which is the maximum of t_1, t_2 and t_3, in number of CPU cycles. We observe that the transfer time decreases with increasing number of parallel units (upto 32 for LZW-SLU and upto 16 for X-Match, amounting to about 54% of T_{dump}), but starts increasing beyond that due to increase in overheads. X-Match leads to lower transfer times until 16 units; for 32 and beyond, LZW-SLU performs better.

Algo.	*# of Comp. Engine*	*Rate t2 bytes/ cycle*	*Rate t_3 bytes/ cycle*	*t_3 $\times 10^5$ cycles*	*t_2 $\times 10^5$ cycles*	*Transfer Time $\times 10^5$ cycles*
LZW-SLU	2	0.20	1	2.61	28.81	28.81
	4	0.40	1	2.56	14.42	14.42
	8	0.80	1	2.53	7.21	7.21
	16	1.60	1	2.64	3.60	**3.60**
	32	3.20	1	2.76	1.80	**2.76**
	64	6.40	1	2.94	0.90	**2.94**
	128	12.80	1	3.15	0.45	3.15
X-Match	2	0.27	1	3.15	20.31	20.31
	4	0.54	1	3.06	10.16	10.16
	8	1.08	1	3.11	5.08	5.08
	16	2.16	1	3.12	2.25	**3.12**
	32	4.32	1	3.19	1.31	**3.19**
	64	8.64	1	3.22	0.63	3.22
	128	17.28	1	3.32	0.32	3.32

TABLE III
COMPARISON FOR PARALLEL ENGINES

VI. CONCLUSION

Transferring internal state of a processor off-chip is a key function during post-silicon debug and testing. Since on-chip cache accounts for bulk of the processor state, we attempt to reduce memory space and time required for the state transfer by introducing a hardware compression engine that is aware of the cache architecture. Knowledge and exploitation of the cache fields enables an efficient compression that is 7-31% better than a compression that is unaware of the structure. We presented and evaluated designs for LRU approximations in the hardware implementation of dictionary-based compression, including a parallel compression architecture that uses multiple compression engines.

REFERENCES

[1] B. Vermeulen, M. Z. Urfianto, and S. K. Goel, "Automatic generation of breakpoint hardware for silicon debug," in *DAC 2004*.
[2] K. J. Balakrishnan, N. A. Touba, and S. Patil, "Compressing functional tests for microprocessors," in *ATS*, 2005.
[3] E. Anis and N. Nicolici, "Low cost debug architecture using lossy compression for silicon debug," in *DATE*, 2007.
[4] H. Fang et al., "CacheCompress: a novel approach for test data compression with cache for IP embedded cores," in *ICCAD 2007*.
[5] M. Kjelso et al., "Design and Performance of a Main Memory Hardware Data Compressor," in *22nd EUROMICRO*, 1996.
[6] R. B. Tremaine et al., "IBM Memory Expansion Technology (MXT)," *IBM Journal of Res. and Dev.*, vol. 45, no. 2, 2001.
[7] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. on Information Theory*, May 1977.
[8] M.-B. Lin, "A hardware architecture for the LZW compression and decompression algorithms based on parallel dictionaries," *J. VLSI Signal Process. Syst.*, 2000.
[9] C. Su, C.-F. Yen, and J.-C. Yo, "Hardware efficient updating technique for LZW CODEC design," *ISCAS*, 1997.
[10] K.-J. Lin and C.-W. Wu, "A low-power CAM design for LZ data compression," *IEEE Trans. Computers*, 2000.
[11] L. Benini et al., "An adaptive data compression scheme for memory traffic minimization in processor-based systems," *ISCAS02*.
[12] Y. Zhang and R. Gupta, "Data compression transformations for dynamically allocated data structures," in *Proceedings of the International Conference on Compiler Construction (CC)*, 2002.
[13] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *DAC*, 1998.
[14] L. Yang et al., "CRAMES: compressed RAM for embedded systems," in *CODES+ISSS*, 2005.
[15] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, 2004.
[16] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and evaluation of a selective compressed memory system," in *ICCD*, 1999.
[17] H. Lekatsas, J. Henkel, and W. Wolf, "A decompression architecture for low power embedded systems," in *ICCD*, 2000.
[18] H. Ghasemzadeh, S. S. Mazrouee, and M. R. Kakoee, "Modified pseudo LRU replacement algorithm," in *ECBS*, 2006.
[19] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," Technical Report cs-tr-97-1342, June 1997.
[20] J. Edler and M. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator." http://www.cs.wisc.edu/ markhill/DineroIV/.
[21] P. Shivakumar and N. Jouppi, "CACTI 3.0: An integrated cache timing, power and area model," WRL Research Rep., August 2001.