EFFICIENT CACHE STATE DUMPING FOR POST-SILICON PROCESSOR VALIDATION

ANANT VISHNOI

AMAR NATH and SHASHI KHOSLA
SCHOOL OF INFORMATION TECHNOLOGY

INDIAN INSTITUTE OF TECHNOLOGY DELHI

Efficient Cache State Dumping for Post-silicon Processor Validation

by

Anant Vishnoi

Amar Nath and Shashi Khosla School of Information Technology

Submitted in fulfillment of the requirements of the degree of

Doctor of Philosophy

to the

Indian Institute of Technology Delhi
May 2010

Certificate

This is to certify that the thesis titled **Efficient Cache State Dumping for Post-silicon Processor Validation** being submitted by **Anant Vishnoi** for the award of **Doctor of Philosophy** in **Information Technology** is a record of bona fide work carried out by him under my guidance and supervision at the **Amar Nath and Shashi Khosla School of Information Technology, Indian Institute of Technology Delhi**. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Preeti Ranjan Panda
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Delhi

Acknowledgment

I am greatly indebted to my supervisors Dr. Preeti Ranjan Panda and Prof. M. Balakrishnan for their invaluable technical guidance and moral support. I would like to thank Prof. Anshul Kumar for his moral support and guidance during my stay at IIT Delhi, especially at the start of my PhD. I also like to thank Dr. Kolin Paul for his valuable feedback, suggestions and help. During my Phd journey I made several friends, their extended help made my PhD stay pleasant and taught me several technical and non-technical lessons. I would like to thank Aryabratta Sahu and Vikram Goyal, their fighting spirit always encouraged me to face tough situations and they made things around me very comfortable. I would like to thank Neeraj Goel, whose valuable IIT system guidance helps me in starting the research. I would like to thank Nagaraju Pothineni, Lava Bhargava, Smruti Padhy, G Krishnaih, BVN Silpa, Sonali Chouhan and Uma mudengudi for their cooperation and support. I would also like to thank the staff of Philips, FPGA and Intel laboratories, IIT Delhi for their help. I am very thankful to Post-silicon verification team at Intel Pvt. Ltd, India for being enabler of my work.

I like to thank my parents for their guidance, encouragement and patience that provided me great moral support during the course of my work. I would also like to thank my sisters for their moral support.

Anant Vishnoi

ABSTRACT

Post-silicon processor debugging is frequently carried out in a loop consisting of several iterations of the following two key steps: (i) processor execution for some duration, followed by (ii) dumping out of the processor's internal state into an external logic analyzer for further offline processing. Internal state of the processor is dominated by the L2 cache. During the process of dumping the cache content, the processor's execution is halted so that the state can be faithfully reproduced offline. In this thesis we propose three hardware mechanisms to reduce the processor debug time and amount of cache data transfer.

In our first work, we propose compression of the cache state on its way out to a logic analyzer in order to save transfer time and expensive logic analyzer space. 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.

In our second work, we propose to reduce the processor halt time, and indirectly processor debug time, with two *Online Cache Dumping* strategies, Retransmit Non-dumped Line (RNL) and Dump History Table (DHT), with the objective of transferring the cache contents while the processor is executing, and yet maintaining fidelity of the dumped data. For typical experimental debug scenarios, we observe that the effective dump times are reduced to between 0.01% and 3.5% of the original times. We also employ compression to reduce the cache content transfer time and logic analyzer space. Our experiments indicate an average compression ratio of 59.2%.

In our last work, we propose to reduce the amount of cache data transfer, which saves the cache content transfer time and logic analyzer space. We propose *Incremental Cache Dumping*: we dump only that portion of the cache which has interesting update information since the last cache dump. Based on the processor execution state, we present two incremental cache state dumping techniques; *Blocking Incremental Cache*

Dumping and *Non-blocking Incremental Cache Dumping*. Experimental results indicate that an average of 36% of the total cache lines are dumped. Further compression of the dumped cache lines reduce 84% of the logic space requirements.

Finally, our proposals enable the state-of-art post-silicon validation to dump the cache content faster and to reduce the amount of cache data transferred for off-line analysis. Our proposals can be extended in the future to enhance post-silicon validation of more complex processor debug systems such as systems with asynchronous peripherals.

Contents

Acknowledgement i

Abstract iii

1 Introduction 1

1.1 Background . . . 1

1.2 Overview of Processor Validation . . . 3

1.2.1 Processor Validation Procedure . . . 3

1.2.2 Processor Structure . . . 4

1.2.3 Post-silicon Validation Bottleneck . . . 5

1.3 Review of Previous Work . . . 6

1.4 Research Contributions . . . 8

1.4.1 Cache Aware Compression . . . 8

1.4.2 Online Cache Dumping . . . 9

1.4.3 Incremental Cache Dumping . . . 9

1.5 Thesis Organization . . . 10

2 Cache Aware Compression 11

2.1 Introduction . . . 11

2.2 Cache Aware Compression . . . 12

2.2.1 Tag . . . 12

2.2.2 Control Bits . . . 13

2.2.3 Data Field . . . 13

2.2.4 ECC (Error Correcting Codes) . . . 15
2.2.5 Parallel Compression . . . 16
2.3 Compressor . . . 17
2.3.1 LZW-SLU . . . 18
2.3.2 LZW-DC . . . 20
2.4 Hardware Design . . . 23
2.4.1 Input Unit . . . 23
2.4.2 Compressor . . . 26
2.4.3 Output Unit . . . 28
2.5 Experimental Validation . . . 29
2.5.1 Experimental Setup and Workload . . . 29
2.5.2 Compression Algorithm Results . . . 30
2.5.3 Compression Results . . . 33
2.5.4 Implementation and Synthesis Results . . . 38
2.5.5 Transfer Time . . . 40
2.6 Conclusion . . . 43

3 Online Cache Dumping 45
3.1 Introduction . . . 45
3.2 Online Processing . . . 46
3.2.1 Cache Partioning . . . 47
3.2.2 Handling Cache Update . . . 48
3.3 Architecture . . . 53
3.3.1 Modification in Cache Architecture . . . 53
3.3.2 RNL . . . 55
3.3.3 DHT . . . 55
3.3.4 Compressor . . . 55
3.4 Experimental Validation . . . 55
3.4.1 Additional Cache Lines Dumped . . . 55
3.4.2 Saved Debug Time . . . 57

3.4.3 Compression Results . . . 57
3.4.4 Implementation and Synthesis Results . . . 58
3.4.5 Limitations . . . 59
3.5 Conclusion . . . 59

4 Incremental Cache Dumping **61**
4.1 Introduction . . . 61
4.2 Incremental Cache Dumping . . . 62
4.2.1 Blocking Incremental Cache Dumping (BICD) . . . 63
4.2.2 Non-blocking Incremental Cache Dumping (NICD) . . . 66
4.3 Architectural Details . . . 70
4.3.1 Modification in Cache Architecture . . . 70
4.3.2 BICD Architecture . . . 70
4.3.3 NICD Architecture . . . 71
4.4 Experimental Evaluation . . . 72
4.4.1 Generation of Experimental Data . . . 72
4.4.2 Cache Lines Dumped . . . 73
4.4.3 Compression Results . . . 75
4.4.4 Effect on Processor Execution (For NICD) . . . 80
4.4.5 Dump Time . . . 81
4.4.6 Implementation and Synthesis Results . . . 84
4.4.7 Limitations . . . 85
4.5 Conclusion . . . 85

5 Conclusion and Future Work **87**
5.1 Main Contributions . . . 87
5.2 Future Directions . . . 89

Technical Biography of Author **97**

List of Figures

1.1 Moore's Law Trend (adapted from [2]) 2
1.2 State-of-the-Art Processor Debug 4

2.1 Compression of Cache Data 12
2.2 Cache Architecture 12
2.3 Effective Utilization of Cache Architecture 13
2.4 Column Wise Byte Extraction 14
2.5 Row Wise Byte Extraction 14
2.6 ECC Example 16
2.7 Parallel Compression of Data Field 17
2.8 Hardware Module 24
2.9 FSM for Handling Tag and Control bits 24
2.10 FSM for Handling ECC Field 26
2.11 Architecture of LZW-SLU 27
2.12 Block Diagram of LZW-DC 27
2.13 FSM controlling LZW-DC Dictionaries 28
2.14 Overall flow 29
2.15 Compression Ratio of Benchmarks 31
2.16 Average Compression Ratio for Parallel Compression 36
2.17 Compression Ratio for Benchmarks (LZW-DC Row-Wise) 36
2.18 Compression Ratio of Benchmarks (LZW-DC Col-Wise) 37
2.19 Compression Ratio of Benchmarks (LZW-SLU Row-Wise) 37
2.20 Compression Ratio Benchmarks (LZW-SLU Col-Wise) 38

2.21 Compression Ratio of Benchmarks (X-Match Row-Wise) 38
2.22 Compression Ratio of Benchmarks (X-Match Col-Wise) 39
2.23 Compression Ratio of Benchmarks (LZW-PLRU Row-Wise) 39
2.24 Compression Ratio of Benchmarks (LZW-PLRU Col-Wise) 40
2.25 Average Compression Ratio 40
2.26 Dump Time T_{dump},t_1, t_2, t_3 41

3.1 Online Dumping with Cache Data Compression 46
3.2 Cache Logical Partition 47
3.3 Axis for Cache Partition 47
3.4 RNL Example 50
3.5 Retransmit Non-dumped Line (RNL) Architecture 51
3.6 DHT Example 52
3.7 Modified Cache Controller FSM 54
3.8 Dump History Table (DHT) Architecture 56
3.9 Average # Cache line Dumps (due to Cache Update) 57

4.1 Example of Blocking Incremental Cache Dumping (BICD) 64
4.2 Example of Non-blocking Incremental Cache Dumping (NICD) 67
4.3 Blocking Incremental Cache Dumping (BICD) Architecture 70
4.4 Non-blocking Incremental Cache Dumping (NICD) Architecture 72
4.5 % of Cache lines Dumped at Various Dump Intervals 73
4.6 Compression Ratio for BICD (LZW-SLU) 76
4.7 Compression Ratio for NICD (X-Match) 77
4.8 Processor Stalls with NICD 79
4.9 % of Dump Time Overhead using Parallel Compressors for BICD 82
4.10 % of Dump Time Overhead using Parallel Compressors for NICD 83

List of Tables

1.1 Cost of Mistakes at each Stage of the Design Cycle[1] 2

2.1 Benchmarks 30
2.2 Area, Critical Path, Processing Speed 32
2.3 Compression Ratio(%) of Cache Fields 34
2.4 Variation in ECC Compression Ratio with Error Rate 35
2.5 Area, Critical Path, and Processing Speed 41
2.6 Comparison for Parallel Engines 42

3.1 Dump Time and Compression Results 58

4.1 Average Compression ratio (%) for BICD 75
4.2 Average Compression ratio (%) for NICD 78
4.3 UHT Size, Area and Critical Path 85

Chapter 1

Introduction

1.1 Background

We are living in a digital era where a huge amount of work is performed with digital gadgets, which are used everywhere from bread makers to automobiles, desktops to supercomputers. The power behind such gadgets lies in the processing elements or processors. Processor is a circuit printed on silicon that is running under the control of some program. As the processing demand is increasing, the processor is becoming more complex with more available space on silicon. This growth over the last few decades has been exponential following the oft-quoted Moore's law (shown in Figure 1.1) [3].

A processor's success depends upon many metrics that include its functionality, speed, area, power, as well as correctness. The last parameter, that is, correctness is mainly based on the extent to which it is verified at design time. Processor verification is performed at all stages of the design cycle. At each stage the cost of fixing the bug found is different, as shown in Table 1.1 [1]. The cost of fixing the bugs found at early stages of the processor design cycle is many orders of times lower than that of bugs found at later stages. The situation is worse if the bug is found after the processor enters mass production. Hence, verification and test engineers are forced to devise new methodologies with the growing complexity of the modern processor.

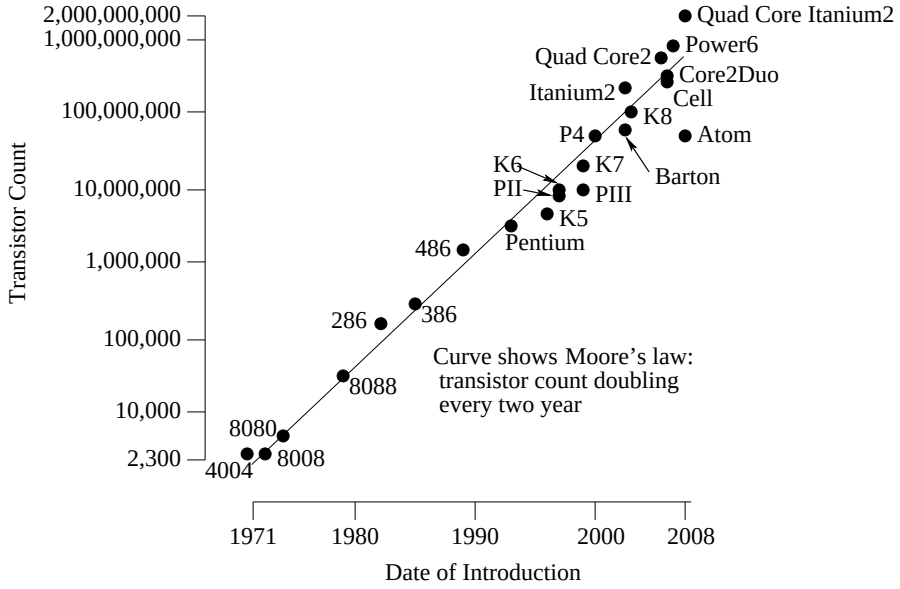


Figure 1.1: Moore's Law Trend (adapted from [2])

Design stage	Cost of Fix
Initial design	$10
Design review	$100
Layout	$1000
Tape release	$10,000-$100,000
Early silicon	$100,000-$1,000,000
Sampling	$1,000,000
Volume production	$10M-$500M

Table 1.1: Cost of Mistakes at each Stage of the Design Cycle[1]

As the market becomes competitive (Intel has released 14 different processors in year 2008 [4]), there is a need for shortening the design time of a processor. Previous studies show that 50% of the design time is spent in processor validation [5]. This directly translates into design cost. Hence, to reduce the design time, there is a need to develop techniques for reducing the validation time. In this thesis we study this problem and propose techniques for reducing the validation time.

The rest of this chapter is organized as follows. In Section 1.2, we give an overview of processor debugging and identify the bottleneck in the processor debug process. Section

1.3 discusses the previous work that is closely related to our work. The thesis contributions are discussed in Section 1.4. Section 1.5 describes the organization of the rest of the thesis.

1.2 Overview of Processor Validation

1.2.1 Processor Validation Procedure

Validation is a phase in the processor design cycle when we find possible faults and rectify them. After a fault is reported, the design team makes necessary changes and the process repeats until the processor chip passes the test on the desired parameters. This makes processor validation a time consuming process.

Processor validation is divided into two phases: 1) Pre-silicon validation, and 2) Post-silicon validation. Pre-silicon validation refers to early simulation based verification (prior to fabrication) of the processor components using Electronic Design Automation (EDA) tools. Post-silicon validation is applied to test chips after fabrication. Since simulation based techniques are very slow (24,000x slower than actual hardware [6]) post-silicon validation is widely used for testing complex processors. Post-silicon validation can be classified into two categories: 1) electrical validation and 2) functional validation.

Electrical validation ensures that the chip works according to the data sheet specification, i.e., chip operation should be robust across all frequency, voltage, temperature and other parameters. For validation, engineers study the changes in electrical property of the chip with changing temperature and voltage through plots known as Shmoo plots [7].

Functional validation confirms that all the components of the processor work as intended. The testing requirement of the validation is similar to that of the target production system. Large examples are run on the systems and the results are verified against simulation results. Such testing is often called postmortem [1].

In postmortem technique, break points are created at certain intervals in a controlled environment [8]. At these break points, the processor is halted and its internal state is

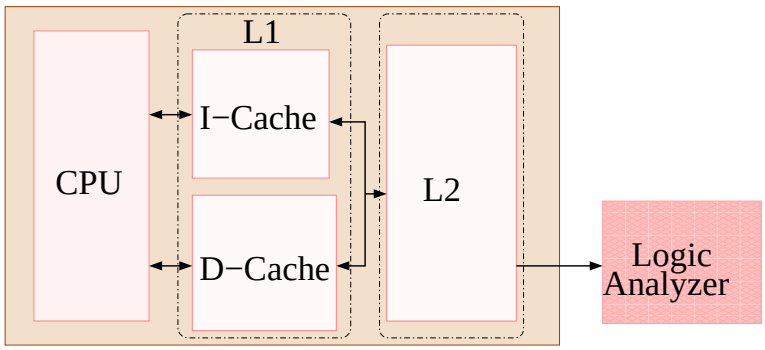


Figure 1.2: State-of-the-Art Processor Debug

dumped to an equipment known as logic analyzer, where it is stored and downloaded for offline analysis. This process is shown in Figure 1.2. The state of the processor typically feeds a more detailed cycle-accurate simulator for detailed analysis. If some bugs are found, then the processor is re-run from the last known good state and the break points are created at smaller intervals. This process is repeated until the source of the bug is located.

The above off-line debug methodology is usually employed in commercial settings for high-end processors because of the considerable setup time overhead involved in exercising the debug infrastructure to extract individual state dumps. During the typical process of analysing a single bug, the designer may have to sift through a large number of processor state dumps. Automated tools aid the designer in navigating through the dumps and extracting useful information. One can visualise an adhoc pipelined operation, where the next series of dumps are taken while the current scenario is being analysed by the designer.

1.2.2 Processor Structure

The typical processor, shown in Figure 1.2, consists of a processor core and on-chip memory. The processor core includes all functional units and other processor units such as instruction decoder, pipeline registers, branch predictor, TLB, reservation station etc. On-chip memory includes all of on-chip cache. Recent trends show that with the shrinking transistor size, cache sizes are increasing rapidly in comparison to other processor

components [4].

The processor state includes all the memory elements present on chip. We observe that for processors such as Alpha [9], with 512KB L2, 16KB L1-I and L1-D cache, only 7% of the processor state is contributed by processor core memory elements and the rest is contributed by on-chip cache [10]. The majority of the cache state is contributed by the bigger L2 cache (we use the term L2 for both on chip L3 and L2 cache).

1.2.3 Post-silicon Validation Bottleneck

Detailed processor simulation is extremely slow and it is well known that cycle-accurate simulators require hours for a detailed simulation of a few milliseconds of processor time. Hence, for reasonable debugging, the state dump has to be taken at least once in a few milliseconds. 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 it off-chip. Since the internal state constitutes all memory elements in the processor, the bulk of which is composed of cache, the problem in transferring the state information to the logic analyzer is essentially that of transferring the cache contents off-chip. For modern high-end processors, with a few megabytes of L2 cache, the dump time through the memory bus is also a few milliseconds. That is, the execution time is comparable to the dump time. Since this cycle repeats millions of times during a typical debug run, the dump time could actually be of the order of 50% of the total run time. Since the debug hardware platform is also very expensive and always busy, time saved in extracting dumps directly corresponds to time saved in the overall debug cycle.

The above scenario motivates us to employ lossless compression techniques to reduce the amount of data dumped during debugging. Let us assume the execution time and dump time are approximately equal to T ms each in one cycle iteration, with a total of n iterations. If the data size could be reduced by 50%, then in addition to the 50% savings in the required logic analyzer memory, the total debug cycle time ideally reduces from $n(T + T) = 2nT$ to $n(T + T/2) = 1.5nT$, i.e., a saving of 25%. Thus, two important benefits follow from state compression:

1. The amount of memory space required in the logic analyzer reduces, leading to cost savings in the logic analyzer memory. Conversely, a larger number of state snapshots can be captured with the same logic analyzer memory.

2. The debug cycle time reduces, leading to faster validation. Conversely, a more comprehensive validation can be performed within the same debug time.

1.3 Review of Previous 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. [11] 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. [12] proposed a lossy compression technique to enhance the post silicon debug process. They capture signatures at intervals which are later offloaded to the debugging software. A scheme for recording the flow of instructions in additional buffers to aid bug localisation is described in [13]. Fang et al. [14] propose a test data compression technique, CacheCompress, to compress scan chain data. The above line of research is complementary 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 and then follow it up with compression techniques that target memory and cache. X-Match [15] is a dictionary based compression algorithm using a move-to-front strategy. It replaces each input data element (of fixed size – four bytes) with a shorter code in case of total and partial match with a dictionary entry. The code word is further compressed using Huffman coding. IBM's MXT (Memory Extension Technology) [16] uses Parallel Block-Referential Compression with directory sharing, a parallel derivative of the Lempel-Ziv [17]. The dictionary-based LZW [18] algorithm has been the basis of several hardware implementations. Lin et al. [19] proposed the architecture of Parallel

Dictionary LZW (PDLZW) compression algorithm and approximate Huffman Coding. Parallel dictionaries of various word sizes are maintained in PDLZW. The input character string is searched simultaneously in all the dictionaries. PDLZW gives good results when we have repeated long sub-strings present in the input stream. However, in our context we have cache data where the repeating string length is smaller. In the CAM based LZW design proposed by Su et al. [20], a second rounding algorithm is used for approximation of LRU policy that is employed in LZW for dictionary updation. Lin and Wu [21] proposed a low power Content Addressable Memory (CAM) design for the LZ77 compression algorithm. Samanta et al. [22] proposed an enhanced CAM cell design for fast execution of LZW compression algorithm.

Several memory compression techniques [23, 24], target the improvement of system performance through reduced memory traffic. Because compression and decompression are performed on-the-fly, needing low latency, the compression ratios achieved are relatively low. Lekatsas and Wolf [25] proposed a code compression technique, using Semi Adaptive Markov Compression (SAMC) [26]. This is used to store program code in a compressed form in memory. A software memory compression mechanism, Compressed RAM for Embedded Systems (CRAMES) was proposed by Yang et al. [27]. The above are not directly applicable to our scenario because we need to handle L2 cache (which contains both instructions and data) through hardware compression. Moreover, the reported compression for these techniques is poorer than the LZ algorithm variants.

The techniques used in [28, 29, 30] attempt to fit multiple cache line data into a single cache line. The FPC cache compression algorithm [28] attempts to find patterns in cache lines and compresses the line using a static coding technique. Alameldeen et al. [31] report that FPC decompresses fast but the compression ratio is about 40% poorer than *gzip*, a variant of LZ. Lee et al. [29] proposed a compressed memory hierarchy model that selectively compresses L2 cache and memory blocks. Their selective compressed memory system (SCMS) uses a hardware implementation of the X-RL compression algorithm [15], a variant of X-Match. Lekatsas et al. [30] use SAMC to decompress CPU instructions that are stored in compressed form in L1 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.

1.4 Research Contributions

The thesis obtains the motivation from the fact that about 93% of the total processor state is contributed by the cache state [10]. A large amount of debug time is spent in transferring the cache state off chip. In this thesis, we propose the following hardware mechanisms to reduce the dump time and logic analyzer space requirements for the cache content.

1.4.1 Cache Aware Compression

Cache Aware Compression utilizes the fact that the input to the compression engine has a cache structure, consisting of well defined fields such as Tag, Control bits, Data field and ECC (Error Correcting Codes) bits. Since each cache field exhibits certain well-defined properties, Cache Aware Compression extracts each cache field separately and, after pre-processing, compresses them separately.

The compression has its own overhead and to overcome it, we propose parallel compression. We noticed that the compression engine spent most of its time in compressing the data field of the cache. Therefore, we propose a parallel compression methodology for the cache data field.

Hardware Compressor

We select LZW, a dictionary based compression algorithm, for cache data compression. LZW searches for unique strings found in the input stream and builds a dictionary with such strings. Since dictionary size is limited, several dictionary pruning policies have been proposed, out of which Least Recently Used (LRU) policy is widely accepted [32].

LRU implementation uses a doubly-linked list structure. To work around the hardware complexity of true LRU implementation, we propose an approximate LRU: Single bit LRU with Update bit (SLU) and its hardware implementation for LZW dictionary pruning.

For better compression, we have also proposed a LZW modification and its hardware implementation, LZW with Dictionary Caching (LZW-DC). LZW-DC uses two dictionaries of different sizes instead of the traditional single LZW dictionary.

1.4.2 Online Cache Dumping

Based on the observation that the L1 cache serves most of the processor requests, and the L2 cache is infrequently accessed by the processor, we propose *Online Cache Dumping*. Thus, instead of halting the processor, we dump the cache state with processor execution overlap. The main challenge in online cache dumping is to maintain the fidelity of the cache state content. For maintaining this fidelity, we propose two alternatives: *Retransmit Non-dumped Line* and *Dump History Table*, with a cost-performance trade-off. Following this we take the cache data through a compression engine, based on Cache Aware Compression, to reduce the data transfer time and logic analyzer memory space.

1.4.3 Incremental Cache Dumping

Since L2 sees a relatively small amount of activity, relatively few cache lines are updated between two consecutive dumps, especially when the dump interval is small. Only the recently updated cache lines are likely to hold some interesting information since the last cache dump. Thus, dumping the entire L2 cache content leads to wastage of precious dumping time. Hence, the obvious choice to reduce the dump time is *Incremental Cache Dumping*: dump only the portion of the cache that was modified since the last dump. Based on the processor execution status, we present two strategies for incremental cache dumping: 1) *Blocking Incremental Cache Dumping (BICD)* and 2) *Non-blocking Incremental Cache Dumping (NICD)*.

1.5 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 discusses the *Cache Aware Compression* methodology and hardware implementation along with the implementation of LZW algorithm variants. Chapter 3 presents our *Online Cache Dumping* mechanism to reduce the cache transfer time with different two strategies considering the hardware cost and performance. Chapter 4 elaborates on the idea and implementation of *Incremental Cache Dumping* for further reducing the amount of cache data transferred per dump. Finally, Chapter 5 summarizes our research contributions and presents possible future directions of our work.

Chapter 2

Cache Aware Compression

2.1 Introduction

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 2.1. 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.

The rest of this chapter is organized as follows: Section 2.2 explains the details of our Cache Aware Compression Engine. Section 2.3 discusses the algorithms and implementation details of the LZW algorithm variants. Architectural details of the compression engine are described in Section 2.4. The experimental validation is discussed in Section

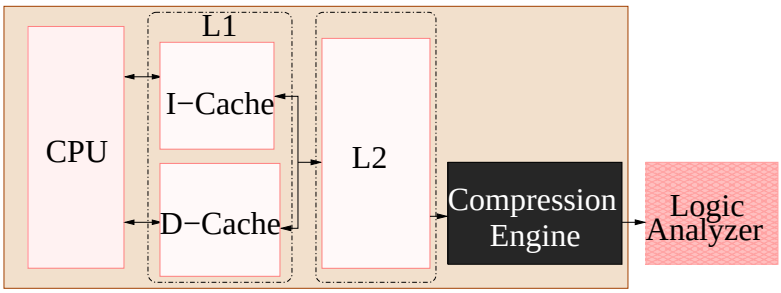


Figure 2.1: Compression of Cache Data

2.5. Finally Section 2.6 concludes this chapter with the findings and observations.

2.2 Cache Aware Compression

Tag	Ctrl	Data	ECC
•	•	•	•
•	•	•	•
•	•	•	•

Figure 2.2: Cache Architecture

The knowledge that our input data has a cache structure allows us to exploit data correlation within cache fields (Tag, Control, Data, and ECC, shown in Figure 2.2) to achieve better compression. We extract the fields and compress them separately, as shown in 2.3.

2.2.1 Tag

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

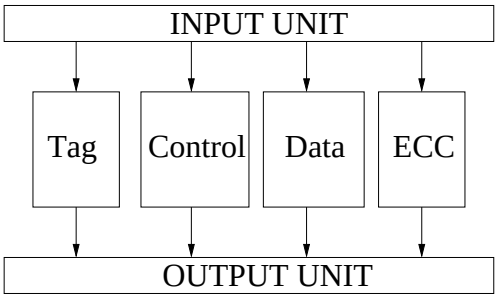


Figure 2.3: Effective Utilization of Cache Architecture

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.

2.2.2 Control Bits

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 at all. Due to this correlation, we separately compress the control bit fields.

2.2.3 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.

Column-Wise Byte Extraction

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

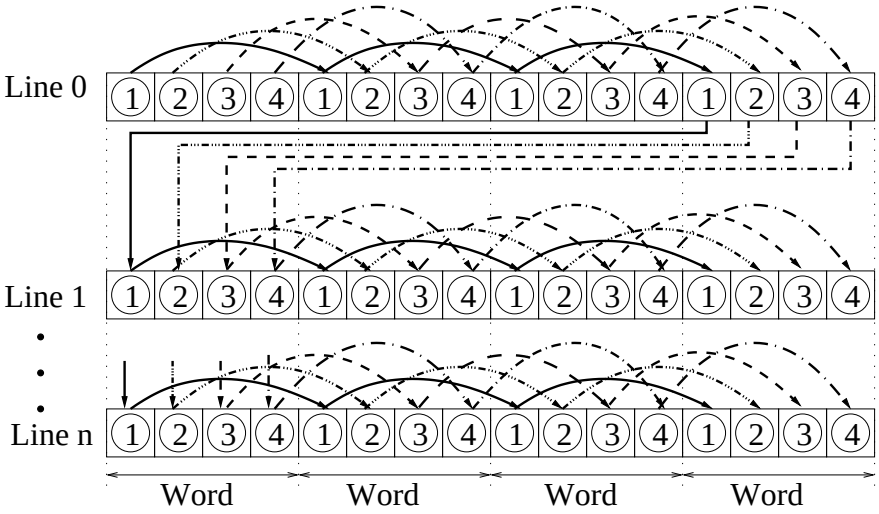


Figure 2.4: Column Wise Byte Extraction

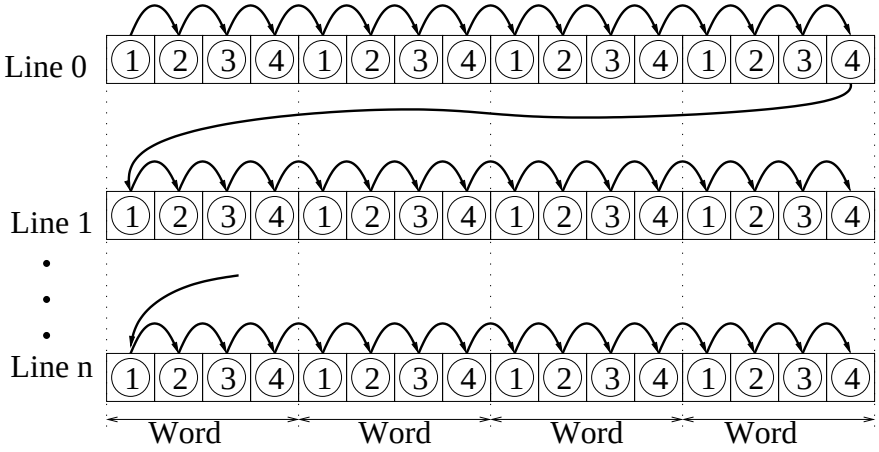


Figure 2.5: Row Wise Byte Extraction

illustrated in Figure 2.4. 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.

Row-Wise Byte Extraction

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 2.5.

Other orders

We also investigated the effect of concatenating data in other orders such as nibble-wise (4 bits of one word concatenated with the corresponding 4 bits of the next word). Such orderings resulted in inferior compression to row- and column-wise, and are not considered further.

2.2.4 ECC (Error Correcting Codes)

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 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.

Example

In Figure 2.6 we have shown two sets of data with ECC fields. In the second set, we have introduced an error each in an ECC and data field. For analysis purpose 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

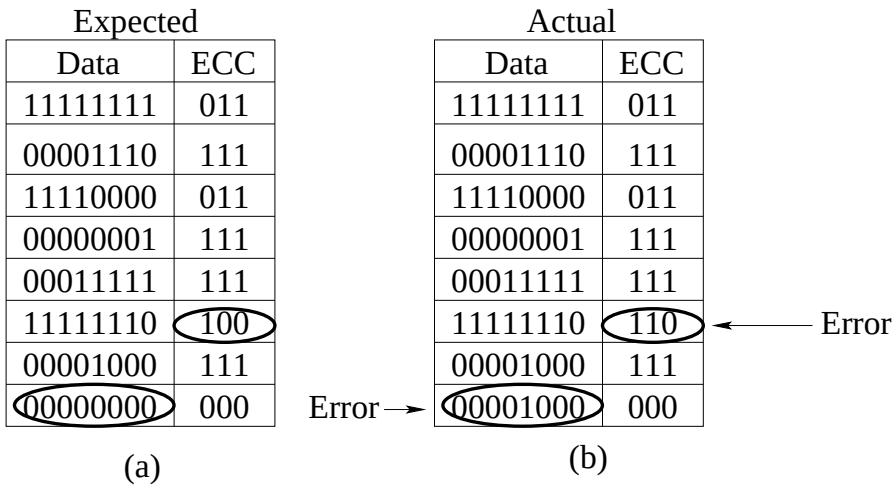


Figure 2.6: ECC Example

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

2.2.5 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 through multiple compressors, as shown in Figure 2.7. Parallel compression reduces the overall compression time, but incurs the overhead of additional area. As discussed in Section 2.5.3, parallel compression improves compression very marginally for a small number of units (up to 4), but for larger numbers, degrades the compression. The degradation occurs because LZW compression produces output streams of varying size for different input streams of the same length. For efficient handling, the compressor outputs are buffered and are transmitted only when the buffers are full. This leads to unpredictable buffer orderings in the output, requiring additional identification bits to output data. A large number of compressors incurs a visible overhead due to these extra bits. Even though parallel compression may not improve the compression ratio, it does reduce dump time by increasing the processing speed, ensuring that the compressor does

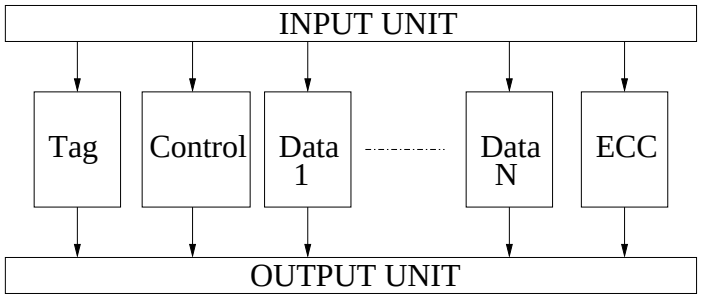


Figure 2.7: Parallel Compression of Data Field

not become a bottleneck in determining the dump time.

2.3 Compressor

Shetty [33] has studied the effect of various compression algorithms on cache data. He observed relatively poor compression ratio for simple techniques such as Run Length Coding [34] and BSTW [35]. Algorithms of Lempel-Ziv family compress the cache data better. We select the LZW [18] compression algorithms for our compression engine.

The Lempel-Ziv class of compression algorithms has been found to be very suitable for hardware implementation [20, 19]. Here, the input stream is parsed and replaced by a set of code words. The LZW compression algorithm works by maintaining a dynamic dictionary. It searches input sub-strings in the dictionary and outputs the dictionary location. The dictionary is updated with the input sub-string not found in the dictionary, with the expectation that locality will cause future references to result in successful searches. Since dictionary space is limited, the management of this space directly impacts the resulting compression.

LZW compression has evolved in the past, with various improvements being proposed to improve dictionary management. Initially, LZW froze the dictionary when it was full. This led to poor compression because the dictionary contents were biased towards patterns found in the beginning of the input stream. The LZC technique clears the dictionary when it is full, causing compression to deteriorate. Another dictionary management strategy, used in UNIX *compress* utility, doubles the dictionary size when it becomes full, until the dictionary size reaches a saturation value, after which it fol-

lows LZC dictionary management. Various pruning mechanisms such as *First In First Out* (FIFO), *Least Frequently Used* (LFU), *Least Recently Used* (LRU), etc., have been proposed [32]. Among these, LRU is known to be most effective dictionary management technique [32].

We present two hardware implementation strategies for LZW. The first, Single bit LRU with Update bit (LZW-SLU), uses a two bit approximation of the LRU policy in its dictionary update. The second, LZW with Dictionary Caching (LZW-DC), exploits locality in the input stream by augmenting the main dictionary with a smaller cache dictionary. We also compare our work with other reported hardware LZW implementations.

Two major implementations of the LZW compression have been reported in the past. Chauchin et al. [20] proposed LZW-WSC, a hardware implementation of LZW using the Window Second Chance updating technique using a one bit LRU approximation. In their approximation, they gave the same preference to both dictionary entry update and reference. However a study by Miller et al. [32] shows that, in general, referenced nodes have lower priority than updated nodes in the LRU list. Our two bit approximation, SLU, explores such priority differentiation and gives better compression than LZW-WSC. PDLZW [19] uses multiple dictionaries of different word sizes to compress the data and uses FIFO policy to update the dictionaries. A comparative study is presented in Section 2.5.2.

2.3.1 LZW-SLU

A variant of the LZW algorithm is outlined in Algorithm 1. For an input word size of 8 bits, locations 0-255 of the dictionary are initialized by data words 0 through 255 respectively. In general, each dictionary element corresponds to a previously encountered string. The algorithm reads the input element and checks if this character, when concatenated with previous search data, exists in the dictionary. If yes, then there is no output. If the new string is not present, the dictionary is updated and the location corresponding to this new string is output. In the latter case, if the dictionary is full, an existing location corresponding to a previously occurred string has to be evicted. Several

Algorithm 1 LZW

```
Input: Input Data Stream
Output: Compressed Data Stream
 1: P_loc ⇐ Read (inp)
 2: while NOT END of STREAM (inp) do
 3:    K ⇐ Read (inp)
 4:    loc ⇐ search (K, P_loc)
 5:    if loc == NULL then
 6:       // NO MATCH
 7:       update (P_loc, K)
 8:       outp ⇐ Write (P_loc)
 9:       P_loc ⇐ K
10:    else
11:       P_loc ⇐ loc
12:       lru_update (loc)
13:    end if
14: end while
15: return
```

update policies are possible.

When an LRU update policy is chosen, the least recently used entry of the dictionary is updated. Keeping track of LRU information requires maintaining a doubly linked list of as many nodes as dictionary elements and some associated complications because dictionary elements are interconnected. Miller et al. [36] suggested maintenance of the LRU list leaf nodes, but implementation of such a structure increases the hardware complexity.

We propose a simple and effective **S***ingle bit* **L***RU with* **U***pdate bit (SLU)* LRU approximation, which also attempts to sidestep the above mentioned problem with a true LRU policy. SLU uses two bits: an 'L' bit approximates LRU behavior and represents recency, and a 'U' bit captures updation information. The L bit is set to '1' when the dictionary entry is referenced (Algorithm 1, line 12). Similarly, the U bit is set to '1' when the dictionary entry gets updated (Algorithm 2, line 11). To update the dictionary we search for the first dictionary entry having L and U bits "00" from the top of the dictionary. As it fills from top to bottom the LRU entry is likely to be at the top. This is shown in Algorithm 2, line 1-3 (0-255 locations of the dictionary are initialization data,

Algorithm 2 LZW-SLU Update Subroutine

Input: P_Loc, data
Output: Updated Dictionary

```
1: dic_loc ⇐ 256
2: while dic[dic_loc].lru_bit != 0 || dic[dic_loc].update_bit != 0 do
3:    dic_loc ⇐ dic_loc + 1
4: end while
5: if dic_loc == Dic_size then
6:    copy_lru_update()
7:    reset_lru()
8: else
9:    dic[dic_loc].loc ⇐ P_loc
10:   dic[dic_loc].data ⇐ data
11:   dic[dic_loc].update ⇐ 1
12: end if
13: return
```

considering input word size of 8 bits). If there is no entry in the dictionary with L, U bits equal to "00", then the L bits are copied to the associated U bits, (*copy_lru_update()*), and the L bits are reset (*reset_lru()* in Algorithm 2). In this way each referenced dictionary entry gets one more chance before removal and a newly created leaf node gets removed if not referenced.

2.3.2 LZW-DC

The compressed output in LZW is a stream of dictionary addresses; hence the dictionary size is a crucial determinant of the output size. If it is small, then each address output is smaller, but since fewer patterns match, the frequency of output is higher. If it is large, then there are more matches, leading to fewer outputs, but each output is larger because more bits are required to address the larger dictionary. To exploit temporal locality, we propose a hierarchically organized dictionary. In our design, which we call LZW with Dictionary Caching (LZW-DC), we have a smaller Cache Dictionary (CD) comprising the initial and recently accessed data, and a larger Main Dictionary (MD) containing the remaining data. If a sufficiently large fraction of accesses is into the CD, then we can save on the overall number of bits required for addressing. For example, instead of a

dictionary with 8K entries, which requires 13 bits to specify each location, we could use a CD with 512 entries and MD with 8K entries. A reference in CD needs 9+1 = 10 bits and a reference in the MD needs 13 + 1 = 14 bits with the extra bit indicating which dictionary is referenced. If accesses to CD comprise more than 25% of the accesses, then the average number of bits is less than 13.

LZW-DC's basic functionality (Algorithm 3) is similar to the LZW algorithm. The LZW-DC tuple consists of the input word, location and the dictionary identifier. LZW-DC searches for this tuple in both dictionaries simultaneously and outputs the location of the longest sub-string matched, along with the dictionary identifier. The dictionary identifier is required to identify the dictionary of the parent node. LZW-DC maintains a "MOVE stack" which stores the locations to be moved from MD to CD. A successful search returns the location and the dictionary identifier. If the tuple is found in MD then the tuple location and subsequent tuple locations are stacked on the MOVE stack (line 6-11) until the stack is full or the tuple search is unsuccessful. In LZW-DC we restrict the size of the MOVE stack to reduce hardware. In a software implementation the stack can grow upto the size of MD. On unsuccessful search or if the MOVE stack is full, LZW-DC generates outputs. If the stack is empty, LZW-DC updates the dictionary and outputs the previously found tuple location and dictionary identifier. In the non-empty stack scenario, LZW-DC calls the *move_MD_CD()* subroutine to move the stacked MD entries to CD. If the stack is full and the last tuple searched is found, then the current found location and dictionary identifier without dictionary update are output. Otherwise, the dictionary is updated with the searched tuple and the last found tuple location is output with dictionary identifier.

LZW-DC updates both the dictionaries with SLU approximation. The move_MD_CD subroutine (Algorithm 4) is an important task in LZW-DC. It moves the MD entries to the CD using the location pointers stored in MOVE stack. The MD pointer pointed to by the top of the stack is swapped with the first CD entry having LRU bit '1'. If no such LRU bit is found in CD then all LRU bits are reset and the CD swap location is set to 256 (initial location). After swapping the entries, both dictionaries update the parent pointer of the children of the swapped node by calling *update_child()* functions.

This loop continues until the stack is empty.

Algorithm 3 LZW-DC

Input: Input Data Stream
Output: Compressed Data Stream

```
1: move_num ⇐ 0
2: P_loc ⇐Read(Input)
3: while NOT END of STREAM do
4:    K ⇐Read(Input)
5:    loc ⇐ search(K, P_loc, P_dic) {search function also returns Found & curr_dic}
6:    if Found & move_num < max_no_move then
7:       P_loc ⇐ loc
8:       P_dic ⇐ curr_dic
9:       if move_num ! = 0 || curr_dic == 2 then
10:         MOVE[move_num].loc ⇐ loc
11:         MOVE[move_num].dic ⇐ curr_dic
12:         move_num ⇐ move + 1
13:      end if
14:   else
15:      if move_num == 0 then
16:         update(P_loc, P_dic, K)
17:         output(P_loc, P_dic)
18:      else
19:         move_MD_CD(MOVE, move_num)
20:         if move_num == max_no_move & Found == 1 then
21:            output(loc, curr_dic)
22:         else
23:            update(P_loc, P_dic, K)
24:            output(P_loc, P_dic)
25:         end if
26:      end if
27:      P_loc ⇐ K
28:      P_DIC ⇐ 1
29:      move_num ⇐ 0
30:   end if
31: end while
32: return
```

The update_child subroutine of the LZW-DC algorithm is similar to the update subroutine of LZW-SLU (Algorithm 2), the main difference being that the update subroutine of LZW-DC first fills the cache dictionary. When CD is full it continues with updating MD using the CAM's LRU policy.

Algorithm 4 move_MD_CD

Input: MOVE, num_move
Output: updated dictionaries

```
1: i ⇐ num_move
2: while i ! = 0 do
3:    decrement(i)
4:    dic_loc ⇐ 256
5:    while cache_dic[dic_loc].lru_bit ! = 0  do
6:       dic_loc ⇐ dic_loc + 1
7:    end while
8:    if dic_loc == Cache_dic_size then
9:       dic_loc ⇐ 256
10:      reset_lru(cache_dic)
11:   end if
12:   if MOVE[i].dic == 2 then
13:      K ⇐ MOVE[i].loc
14:      swap(cache_dic[dic_loc], main_dic[k])
15:      update_child(cache_dic, dic_loc, 1)
16:      update_child(main_dic, dic_loc, 1)
17:      update_child(cache_dic, K, 2)
18:      update_child(main_dic, K, 2)
19:   end if
20: end while
21: return
```

2.4 Hardware Design

Based on the functionality, the compression engine can be divided into the following three sections: Input Unit, Compressor and Output Unit as shown in Figure 2.8.

2.4.1 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. The input unit consists of the following sections.

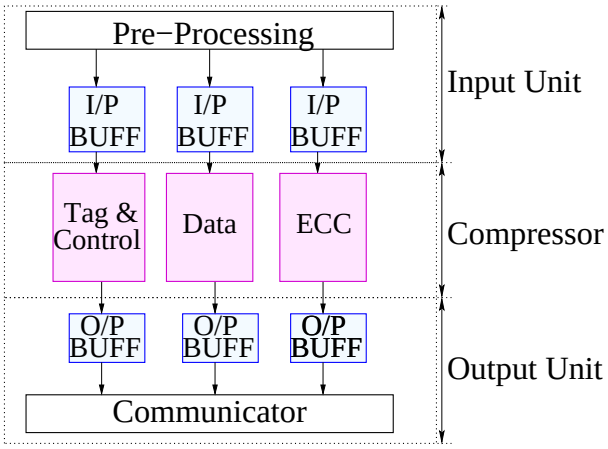


Figure 2.8: Hardware Module

Tag and Control

This section of the input unit processes the Tag and Control bits of the cache line. If necessary, different fields of the cache line could be merged and fed to the same compression engine. 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 2.8). Not much difference was observed in the compression and this led to reduced area by dropping one compression engine.

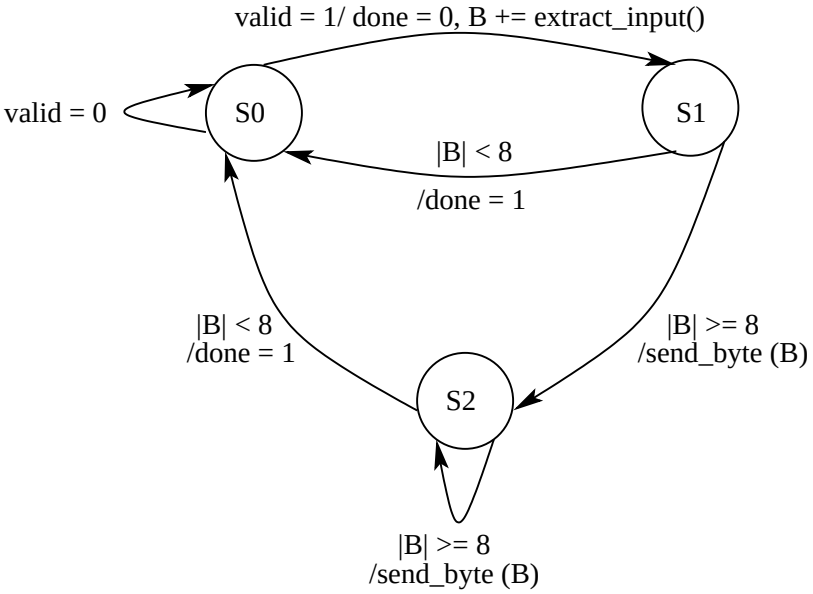


Figure 2.9: FSM for Handling Tag and Control bits

Figure 2.9 shows the simple FSM for forwarding tag and control bits into the corresponding input unit buffer. The tag and control bits in a cache line together usually do not amount to an exact multiple of 8, leading to fractional bytes which are concatenated across multiple cache lines. The idle state is $S0$, from where the FSM proceeds to $S1$ when *valid* is HIGH, indicating a new cache line arrival. Tag and control data are extracted from the cache line and bytes are pushed into the buffer. If fractional bytes are left over, it proceeds to state $S2$, concatenates the data with any bits carried over from previous cache lines, and generates additional bytes. The FSM then proceeds to the idle state $S0$.

Data

The data section of the input processing unit sends the data field of the received cache line into the input buffers of the compressor. Bytes are extracted from the cache line either row-wise or column-wise depending on the value of a configuration bit, and deposited into the buffers. If parallel compressors are present, the cache line data is spread across the respective multiple input buffers.

ECC

This unit calculates the ECC of each new cache line and compares with the ECC field. If both are equal then it dumps '0' else it dumps '1' along with the actual ECC bits. These bits are concatenated, bytes are extracted from them, and pushed to the buffer for compression.

The FSM of the ECC unit, shown in Figure 2.10, has 3 states with state $S0$ as the initial state. It transitions to state $S1$, when new data is available, computes ECC of the cache line data and compares with the stored ECC field. On a match, it adds a '0' to the output stream, else a '1' and the ECC value, and transitions to $S2$. If the number of bits to be transmitted has reached or exceeded 8, then it extracts one byte at a time and sends it to the buffer for compression (function send_byte). When fewer than 8 bits remain, the FSM returns to the idle state.

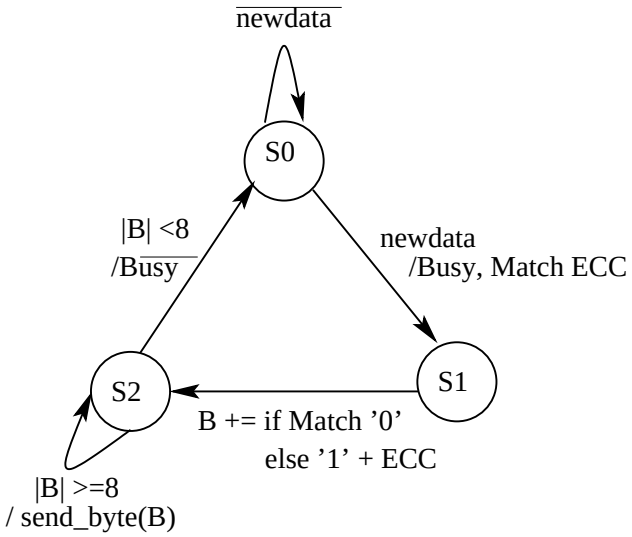


Figure 2.10: FSM for Handling ECC Field

2.4.2 Compressor

The LZW algorithm requires the searching of the input word and location tuple in the entire dictionary. A Content Addressable Memory (CAM) structure is amenable to such functionalities.

Hardware Implementation of LZW-SLU

LZW-SLU requires two CAMs – a Dictionary CAM (D-CAM) and an LRU CAM (L-CAM), as shown in Figure 2.11. D-CAM maintains the LZW dictionary and L-CAM maintains the L and U bits. For dictionary size N, the D-CAM and the L-CAM are organised as $N \times (\log_2 N + 8)$ and $N \times 2$ respectively. In a D-CAM entry, the first $\log_2 N$ index bits are the bits of the parent location of the stored trie [18] node. The next 8 bits of the D-CAM entry are the data bits. The tuple location and the current input data are searched in the D-CAM. On successful search, the location is output to the register *Loc*. Simultaneously, the current input data is also stored in register *data*. On successful search the next location field of the tuple comes from *Loc*. Otherwise, on the next search, the location field of the tuple is filled with the value stored in *data*. On unsuccessful search, the *output* register generates the location of the last tuple and the D-CAM gets updated with the location provided by L-CAM.

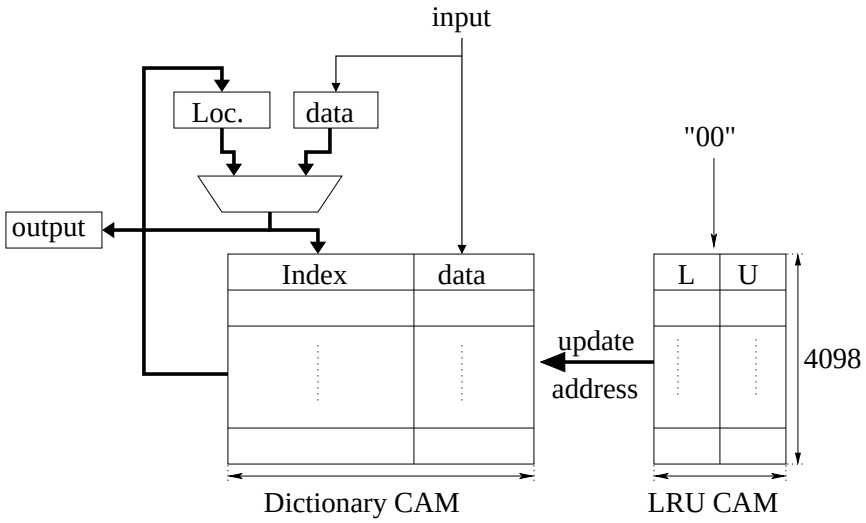


Figure 2.11: Architecture of LZW-SLU

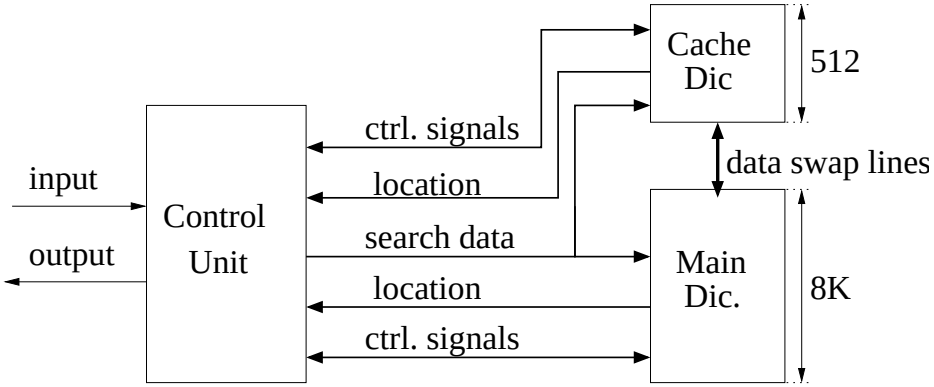


Figure 2.12: Block Diagram of LZW-DC

According to SLU, the update address is the first location having L and U bits equal to "00" from the top (location 256). Since multiple matches are possible in L-CAM, a priority encoder is needed to resolve between them [37]. The L-CAM requires some further modifications to a standard CAM structure for implementing copying of the L bits to U bits and resetting for L bits. For this we need one additional transistor which acts as a switch between the two CAM cells, L and U. The reset logic of the L-CAM cell is similarly straightforward.

Hardware Implementation of LZW-DC

Figure 2.12, the block diagram of the LZW-DC, shows the signals between the control unit and the dictionaries. The control unit takes the input and output of the longest

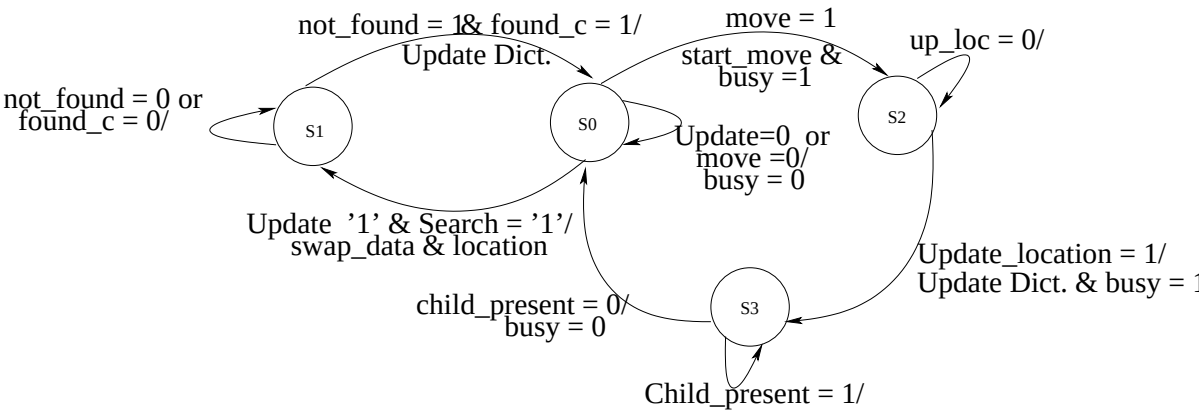


Figure 2.13: FSM controlling LZW-DC Dictionaries

input sub-string match. It generates the control signals and the tuples to be searched in the dictionaries. It also maintains the MOVE stack, for the swap data between Cache Dictionary and Main Dictionary.

The FSM controlling the LZW-DC dictionaries is shown in Figure 2.13. $S0$ is the initial/idle state of the FSM. After receiving the *Search* and *Update* signals, it transitions to $S1$. It remains in $S1$ as long as either of the signals *not_found* and *c_found* is LOW. Signal *not_found* is generated by CD; it is set to HIGH if the searched data is not found in CD. Similarly, *c_found* signal is generated by MD. If the searched data is not found in either dictionary then the FSM updates MD and transitions back to $S0$. It jumps to $S2$ if the searched data is found in MD and the *move* signal is set. In state $S2$ we swap the data between MD and CD and transition to state $S3$. In $S3$ all the parent pointer fields of the children of swap entries are updated.

2.4.3 Output Unit

The Output unit buffers the output of the compressor and communicates with the logic analyzer. Whenever data is ready for dumping, either because the output buffer of the corresponding compression unit is full or because the input stream has ended, it initiates the protocol to transfer compressed data to the logical analyzer. Identification bits are prefixed to the compressed stream to indicate the source compression unit during reconstruction of the cache content from the compressed stream.

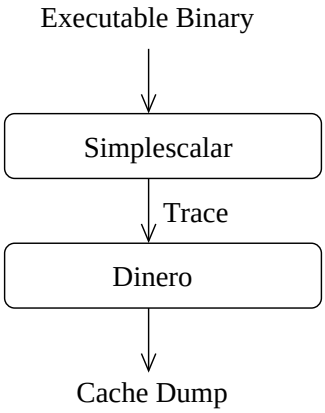


Figure 2.14: Overall flow

2.5 Experimental Validation

2.5.1 Experimental Setup and Workload

Our evaluation framework for dumping the cache dump is built around two public domain tools, Simplescalar (processor simulator)[38] and Dinero (cache simulator)[39], shown in Figure 2.14. Our modification of Simplescalar accepts executable binaries as input and generates the memory access trace file. The trace file forms the input to Dinero which can be triggered at arbitrary points to dump the cache content.

Simplescalar is a processor simulator that can be configured in various ways. Simplescalar supports many instruction sets but for our experiments we have used PISA (Portable Instruction Set Architecture). Simplescalar does not maintain data in its cache to avoid data duplication. It directly accesses the memory location from the memory array where the executable binary is loaded. The cache structure is used only for generating the statistics. We modified the simulator for generating the trace file consisting of both address and data.

Dinero is a trace driven cache simulator. The basic idea is to simulate a memory hierarchy consisting of various caches connected as one or more trees, with reference sources (the processors) at the leaves and a memory at each root. The various parameters of each cache can be set separately (architecture, policy, and statistics). As in Simplescalar, Dinero also does not maintain the data during the cache simulation; it only keeps track

of the tags. We modified Dinero to maintain the data, so that we are ready with the cache dump when necessary.

Currently the cache dump consists of the tags, data, control bits and ECC bits. We generated experimental data by running the Mediabench and CPU-SPEC 2000 benchmarks (shown in Table 2.1), on our frame work. To create realistic scenarios, where one application runs after another, we called the benchmarks as a subroutine of a program.

Benchmark Suite	Benchmarks	Domain
SPEC-2000	CHESS	Graph
	MCF	Public Transport
	LBM	Fluid Mechanics
	MILC	Computational Biology
	HMMER	Computational Biology
	BZIP2	Compression
Media	MPEG_LAME_ENC	Concatenate Media Benchmarks
	LAME_ENC	
	ENC_MPEG	
	MPEG_LAME	

Table 2.1: Benchmarks

The cache dump was taken periodically after execution of the 10 million instructions. These dumps were compressed separately.

2.5.2 Compression Algorithm Results

We have tested our proposed compression algorithms on several Benchmark suites: Corpus [40], Audio and Exe [41]. We compare our implementations with previously published compression hardware implementations (PDLZW [19] and LZW-WSC [20]), and LZW using the PLRU approximation [42]. We use the definition of compression ratio as the percentage of bytes saved, i.e., $(1 - o/i) \times 100\%$, where o is number of output bytes and i is number of input bytes.

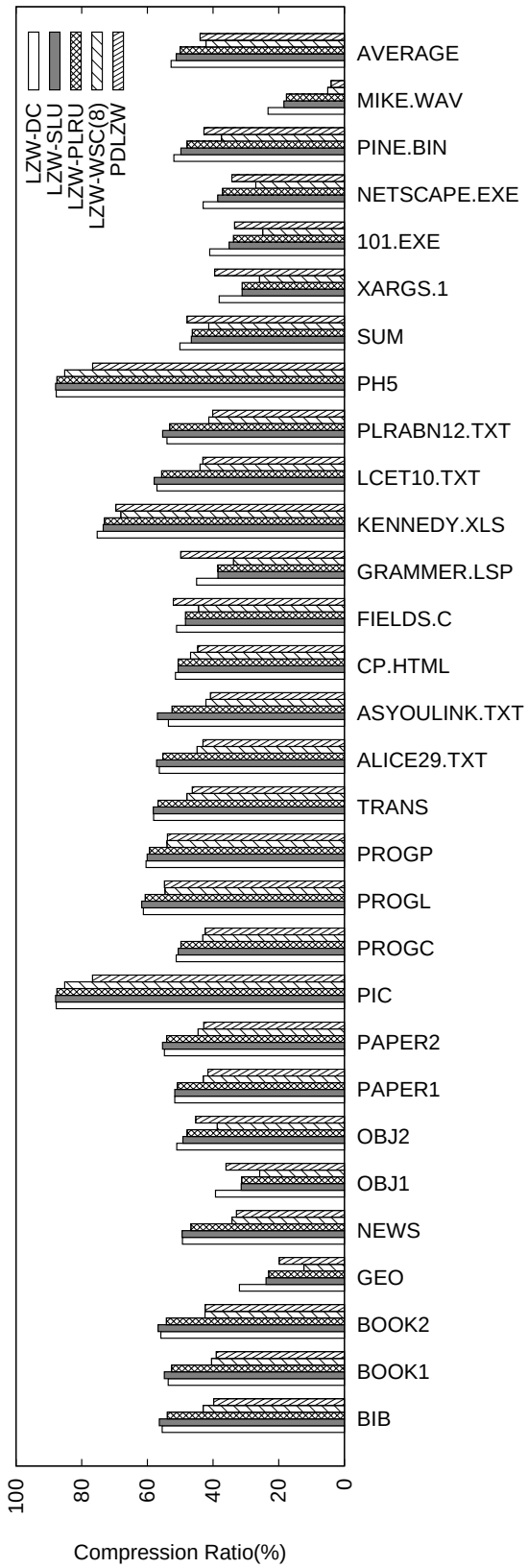


Figure 2.15: Compression Ratio of Benchmarks

The compression results for the different implementations are shown in Figure 2.15. An 8K dictionary size is used for LZW-SLU, LZW-PLRU, and LZW-WSC. For LZW-DC, CD and MD have 512 and 8K entries respectively. For PDLZW we used the best configuration[19]: 1K address space with 8 parallel dictionaries. We observe that LZW-SLU gives 2-15% more compression (average 9.07%) than LZW-WSC and outperforms PDLZW by upto 16.59% (average 7.26 %). LZW-PLRU is marginally worse (average 1.2%) than the LZW-SLU. LZW-DC overall performs marginally better (average 1.58%) than LZW-SLU. The best compression results are obtained by LZW-DC.

Hardware Implementation Details

Our implementation consists of VHDL models synthesised with an 180nm ASIC library with Synopsys Design Compiler. CACTI [43] was used to estimate the area and delay of memory components. We limit the size of LZW-DC "MOVE" stack to 32. Table 2.2 shows the area, critical path and processing speed comparison of LZW hardware implementations. LZW-DC takes an average of 3 cycles to process a byte. The comparison shows PDLZW to be better than other LZW implementations in terms of processing rate. PLRU is normally effective for cache associativity logic, but does not scale effectively for larger set of data such as in the LZW dictionary.

Algorithm	*Area* (mm^2)	*Critical path* (*ns*)	*Processing speed* (*Bytes/cycle*)	*Compression rate* (*MB/sec*)
PDLZW	2.16	2.78	1-8	360 - 2880
LZW-WSC 8	3.75	4.95	1	202
LZW-SLU	3.42	5.21	1	192
LZW-DC	3.99	12.21	0.33	27
LZW-PLRU	5.15	6.01	1	166

Table 2.2: Area, Critical Path, Processing Speed

2.5.3 Compression Results

Effect of compressing cache fields separately

Table 2.3 shows the compression ratios of all the cache components when they are compressed separately. Column 1 shows the application and Column 2 shows the compression algorithm. The Tag and Control 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, BZIP2 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 2.3 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% over the baseline case 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. However, our design is configurable to dump either row-wise or column-wise. Application characteristics can be taken into consideration in setting this configuration before dumping.

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

Table 2.3: Compression Ratio(%) of Cache Fields

ECC Compression

Our ECC compression strategy is based on the assumption that ECC errors are relatively rare. Table 2.4 shows the variation in the ECC compression ratio with changes in the error rate. The first column gives the application name. The second column shows the compression algorithms used. The 3^{rd}, 4^{th}, 5^{th}, and 6^{th} column show the compression ratio of the ECC fields with error rate being 10%, 1%, 0.1%, and 0.01 % respectively. The improving compression with lower error rates is due to longer strings of zeros, which are generated when ECC errors are fewer.

Parallel Compression Engines

Figure 2.16 shows the average compression ratio of all the benchmarks for LZW-SLU (row-wise). The best compression (by a narrow margin) is observed for 4 parallel compression engines. Beyond that, the overhead of the identification bits leads to worse compression. Similar trends are noticed for other compression algorithms, as observed

Application	Comp. Algo	ECC Error			
		10%	1%	0.1 %	0.01%
CHESS	LZW-DC	36.72	81.44	84.76	96.98
	LZW-SLU	34.96	80.03	83.46	96.69
LBM	LZW-DC	45.62	83.20	95.06	96.96
	LZW-SLU	45.83	81.95	94.57	96.67
MCF	LZW-DC	21.87	78.50	94.70	96.96
	LZW-SLU	18.34	76.97	94.17	96.67
MILC	LZW-DC	21.30	78.74	94.74	96.96
	LZW-SLU	17.00	77.07	94.22	96.67
HMMER	LZW-DC	28.10	79.18	94.35	96.57
	LZW-SLU	25.85	77.90	93.84	96.27
BZIP2	LZW-DC	18.65	77.82	84.25	96.94
	LZW-SLU	14.48	76.25	82.89	96.65
ENC_MPEG	LZW-DC	21.51	78.64	94.76	96.96
	LZW-SLU	17.50	77.02	94.24	96.67
LAME_ENC	LZW-DC	19.18	78.49	94.76	96.93
	LZW-SLU	14.56	76.75	94.24	96.63
MPEG_LAME_ENC	LZW-DC	18.81	78.35	94.69	96.94
	LZW-SLU	14.36	76.71	94.16	96.65

Table 2.4: Variation in ECC Compression Ratio with Error Rate

later in this section. Although parallel dictionaries do not improve compression significantly, they have an important effect in reducing dump time (Section 2.5.5).

Compression due to Different Techniques

Figures 2.17 and 2.18 show the compression results for the benchmarks with different number of parallel compression engines, using the row-wise and column-wise variants respectively of the LZW-DC algorithm. A marginal increase in compression is observed in some cases (LBM, MILC, MCF) for relatively smaller number of engines. As observed earlier, the reduction in compression ratio for larger number of compression engines is due to the need for additional identification bits that get appended to the output stream.

The compression ratio observations due to LZW-SLU algorithm applied row-wise and column-wise, shown in Figures 2.19 and 2.20, are almost the same as LZW-DC, with the absolute values of the compression ratios being slightly lower than LZW-DC.

Figures 2.21 and 2.22 show the compression results for X-Match algorithm, with the data field considered row-wise and column-wise respectively. While most bench-

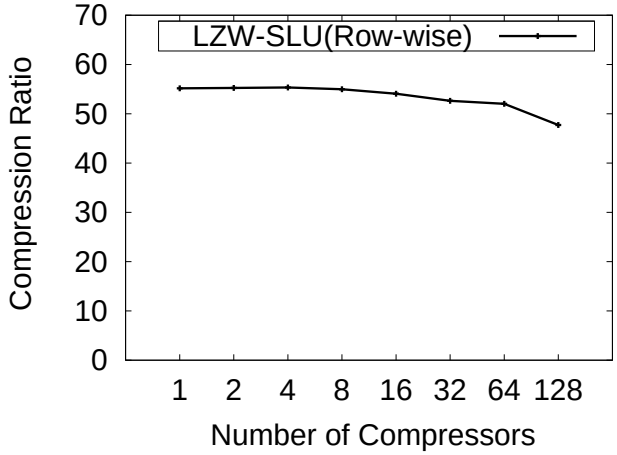


Figure 2.16: Average Compression Ratio for Parallel Compression

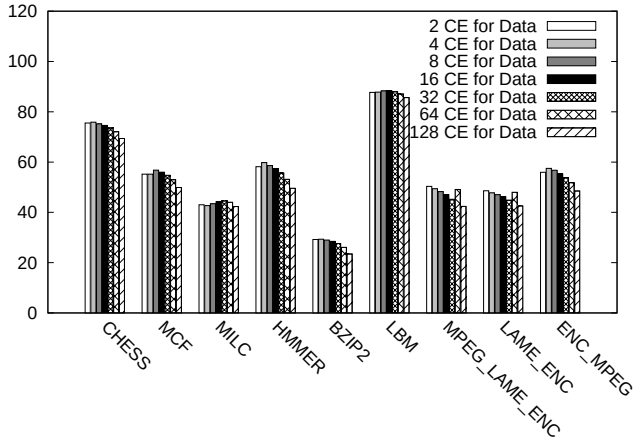


Figure 2.17: Compression Ratio for Benchmarks (LZW-DC Row-Wise)

marks exhibit the same compression behaviour as LZW-DC and LZW-SLU. For some of them (CHESS, MILC, LBM), the compression improved with increasing number of compression engines. Overall, X-Match results in lower compression values than the LZW variants.

Finally, Figures 2.23 and 2.24 show the compression ratios using a variant of LZW with the popular Pseudo-LRU approximation [42] for replacement. LZW-PLRU follows approximately the same trend as LZW-DC and LZW-SLU, with slightly lower overall compression.

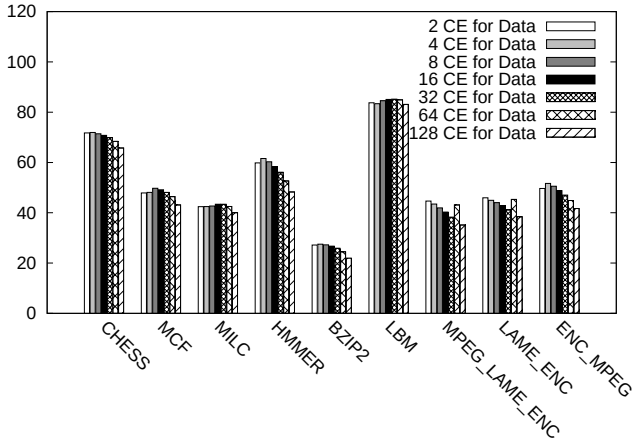


Figure 2.18: Compression Ratio of Benchmarks (LZW-DC Col-Wise)

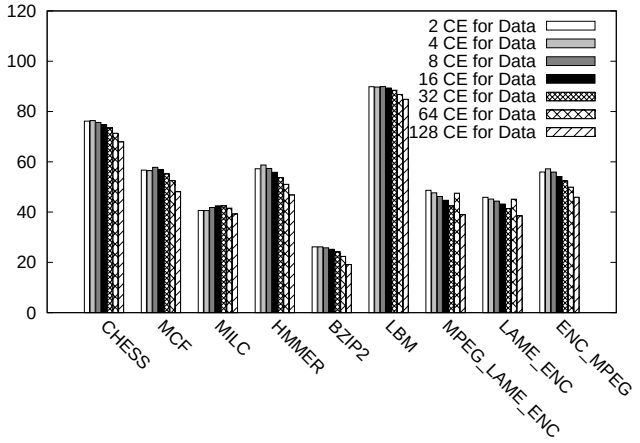


Figure 2.19: Compression Ratio of Benchmarks (LZW-SLU Row-Wise)

Comparison Summary: Average Compression

Figure 2.25 shows a comparison of the average compression ratios of different compression algorithms, computed with the number of parallel compression units varying in powers of 2, from 2 to 128. We notice that, except for Bzip2, X-Match gives worse compression results than other implementations of the LZW algorithms. LZW-SLU gives on an average 9.2% more compression than X-Match and 1.2% more than PLRU. We also observe that LZW-DC gives slightly better (0.5%) compression than LZW-SLU.

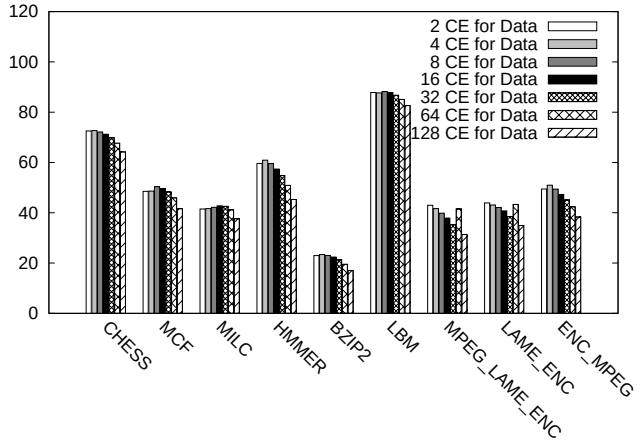


Figure 2.20: Compression Ratio Benchmarks (LZW-SLU Col-Wise)

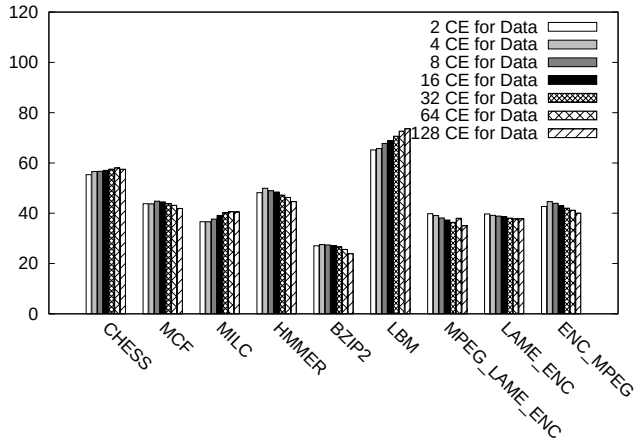


Figure 2.21: Compression Ratio of Benchmarks (X-Match Row-Wise)

2.5.4 Implementation and Synthesis Results

Hardware Breakup

We have designed our hardware implementation to be roughly compatible with Intel Pentium 4 class of processors – with 2 GHz clock, 32 bit bus, 533 FSB, implemented on 180nm technology. Our implementation consists of VHDL models synthesized into a 180nm ASIC library with Synopsys Design Compiler. CACTI [43] was used to estimate area and delay of memory components.

Table 2.5 shows a comparison between the area, critical path delay, the number of CPU cycles needed to accommodate the critical path delay, and the processing speed

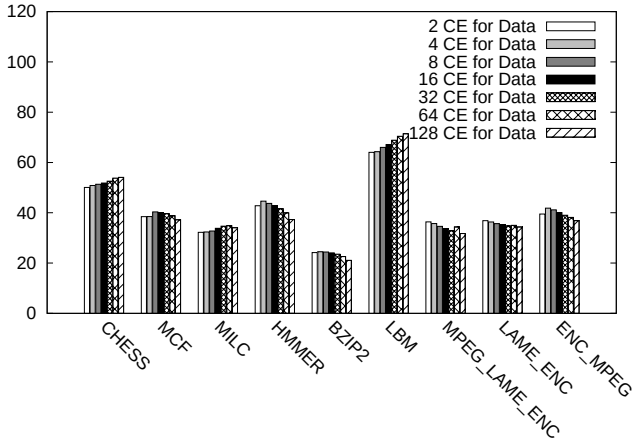


Figure 2.22: Compression Ratio of Benchmarks (X-Match Col-Wise)

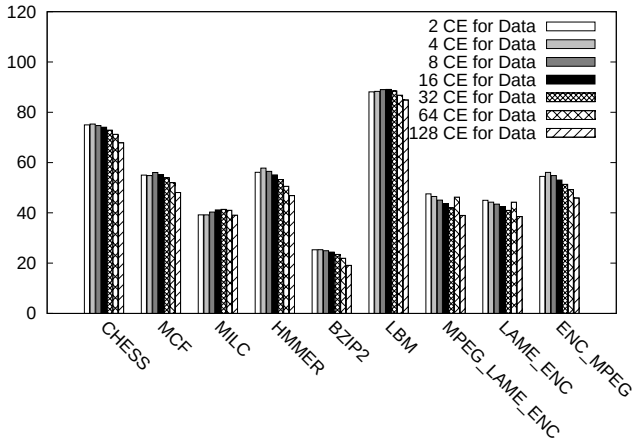


Figure 2.23: Compression Ratio of Benchmarks (LZW-PLRU Row-Wise)

of compression algorithms. The comparison shows X-Match to be superior in area and delay. However, as seen in Figure 2.25, its compression ratio is worse. The computation of total dump time is a function of different parameters. This is discussed in Section 2.5.5. 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.

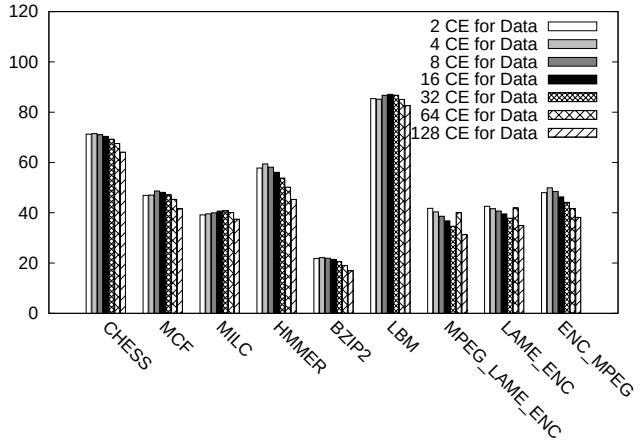


Figure 2.24: Compression Ratio of Benchmarks (LZW-PLRU Col-Wise)

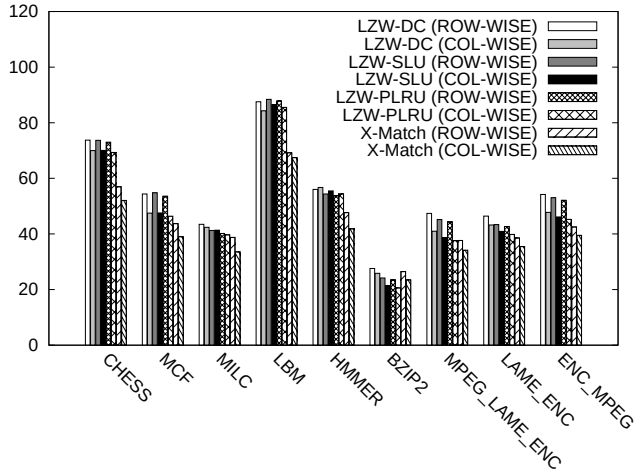


Figure 2.25: Average Compression Ratio

2.5.5 Transfer Time

Transfer time is an important metric for comparison of the hardware compression strategies. The scenario is shown in Figure 2.26.

The time taken to transfer the cache contents with no compression is:

$$T_{dump} = \frac{\text{num\_bytes}}{\text{bus\_speed}} \tag{2.1}$$

For a cache with 512 KB, 4-way, 16 byte line, we have:

$$T_{dump} = 6.12 \times 10^5 \text{CPU Cycles} \tag{2.2}$$

Compression Algorithm	Area (mm^2)	Critical Path(ns)	# CPU Cycles	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 2.5: Area, Critical Path, and Processing Speed

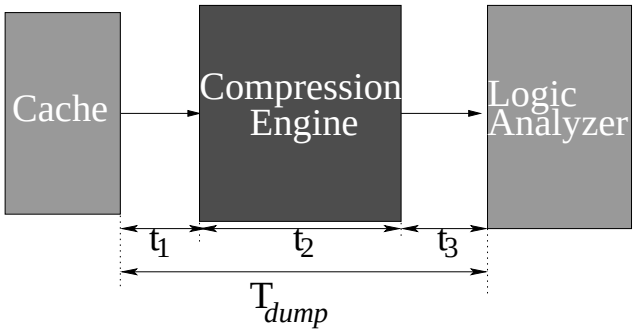


Figure 2.26: Dump Time T_{dump},t_1, t_2, t_3

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

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

- t_3: Time to transfer compressed data

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

Table 2.6 shows the effect on transfer time due to increase in number of parallel com-

Algorithm	*# 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 2.6: Comparison for Parallel Engines

pression 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 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 the Column 4 and the data to be compressed. Compression time decreases with number of compression units. The transfer time (Column 7) 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.

2.6 Conclusion

Transferring internal state of a processor off-chip, is a key function during post-silicon debug. 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.

Chapter 3

Online Cache Dumping

3.1 Introduction

During the process of dumping the cache content offline, the processor's execution is halted so that the state can be faithfully reproduced offline. Freezing the processor execution for the entire cache dump duration leads to loss of precious time in the debug infrastructure, particularly since the action is repeated a large number of times [8]. In Chapter 2 we attempted reducing the dump time by transferring the cache content through a compression engine. In this chapter, we utilise the fact that the L2 cache is less frequently accessed and updated by the processor. To reduce the debug cycle time we attempt *Online Cache Dumping* – the L2 cache dump proceeds simultaneously with processor execution – while maintaining valid data in the dump. The obvious challenge here is to ensure that the L2 cache state being dumped out is not corrupted by the executing processor.

In this chapter, we present two strategies for online cache dumping, with different cost-performance trade-offs. When the signal for starting the cache dump is received, we start the dumping, maintaining a logical division between the *dumped* and *non-dumped* areas of the cache. In *Retransmit Non-dumped Line (RNL)*, when the processor attempts to update a cache line that is not yet dumped, we first dump the existing line so that it can be faithfully reproduced offline. In the *Dump History Table (DHT)* mechanism, we ensure that dumping occurs only once per line in the non-dumped area by maintaining

a table that keeps track of whether each line has been already dumped. The dumped cache content is then taken through a compression unit in order to reduce the transfer time and expensive logic analyzer space (Figure 3.1).

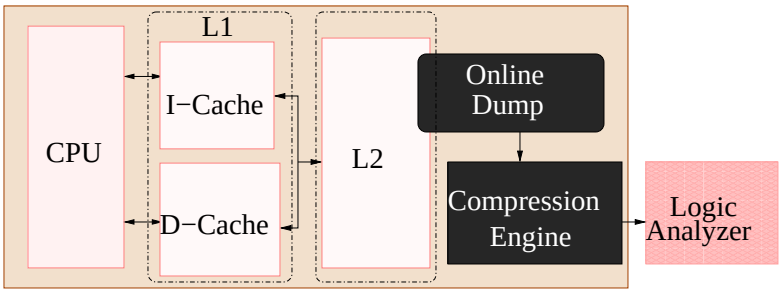


Figure 3.1: Online Dumping with Cache Data Compression

3.2 Online Processing

The objective of online cache dumping is to reduce the total debug cycle time by overlapping the dumping of L2 cache state with processor execution. The key requirement here is that, if we need the state at time $t = T_1$ dumped, the state should not be corrupted by the processor when it resumes execution. Stalling is necessary to dump out the contents of the register file, pipeline registers, L1 caches, etc. However, the bulk of the processor state consists of L2 cache which is not so frequently accessed, and it is possible to overlap the dumping of L2 cache with the processor's execution when it resumes after T_1, with minor impact on execution time.

The central idea in achieving overlapping execution with dumping is as follows. We carry out the cache dump one line at a time in increasing order of the line number. A counter is maintained to keep track of what part of the cache has already been dumped; this partitions the cache logically into two areas: *dumped area* and *non-dumped area*, as shown in Figure 3.2. If, during dumping, the processor requests an update to line B_1 in the dumped area, then execution proceeds normally since the state for B_1 has already been captured. If an update is requested to line B_2 in the non-dumped area, then line B_2 (consisting of data, tag, control bits, and the identifying line number) is first dumped out before updation. Processor execution and dumping proceed simultaneously

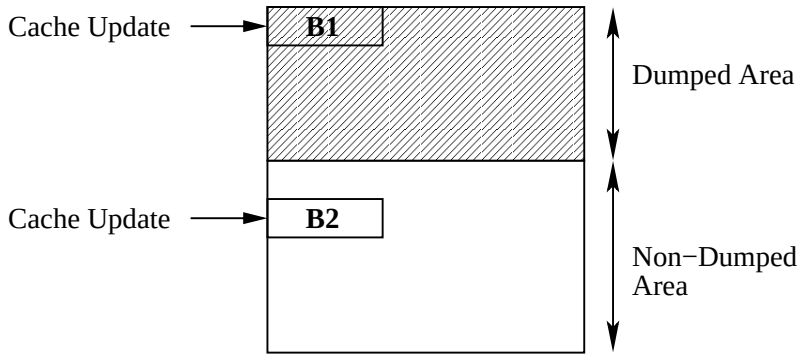


Figure 3.2: Cache Logical Partition

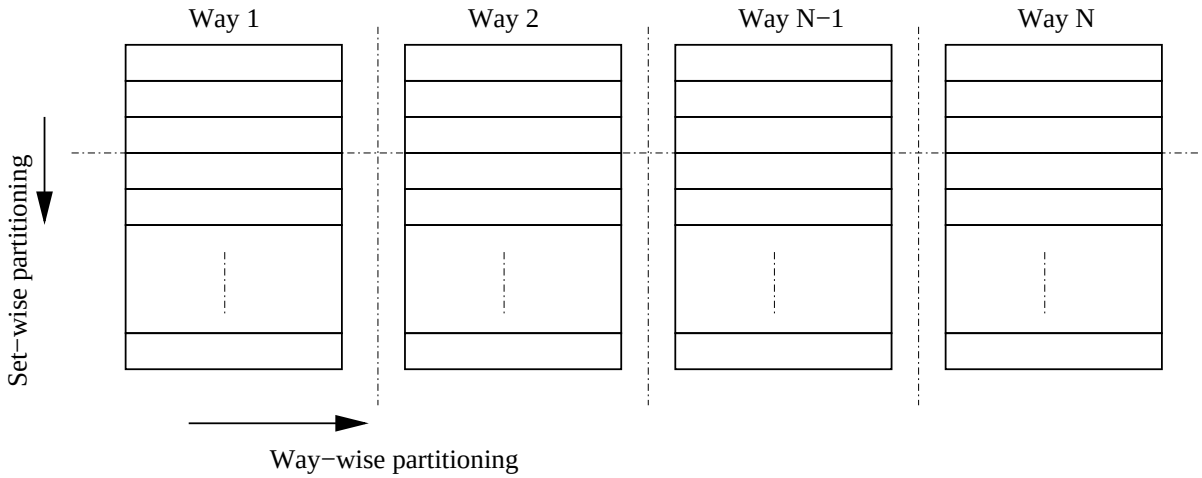


Figure 3.3: Axis for Cache Partition

in normal sequence after this. The line number and data is used later by the offline state reconstruction program to retain the right state for B_2 and ignore the (corrupted) state dumped out later when the dump sequence reaches B_2.

3.2.1 Cache Partioning

Caches can be naturally partitioned at different levels of granularity in different ways. Way-wise partitioning suggested in [44], shown in Figure 3.3, could possibly be adapted for our purpose, where we can dump the cache one way at a time, with the corresponding cache way/bank becoming unavailable for processor execution during this time. However, this interferes with the cache behavior of the executing program and may obstruct the debugging process. Other problems such as data duplication may occur in the different

cache ways. We use a set-wise partitioning (Figure 3.3) approach using a simple *counter* to partition the cache. The counter keeps track of the current index being dumped, and is incremented after dumping the lines in the set corresponding to each index. Determining whether an update request to an address occurs in the dumped or non-dumped area is done by comparing the index with the counter.

3.2.2 Handling Cache Update

A key issue in overlapping execution with cache dump is the proper handling of cache updates while the dumping is in progress. For achieving this we describe two mechanisms that represent a trade-off between the volume of additional dumped data and area overhead.

Retransmit Non-dumped Line (RNL)

In Retransmit Non-dumped Line (RNL), on a cache update during processor execution, we first check whether the update being requested is to the dumped or non-dumped area. No additional action is necessary if the line has already been dumped. If it belongs to the non-dumped area, we first dump the specific line before permitting the cache to be updated. Along with the cache line data, we also dump the cache line and way number to help identify the line/way during offline cache state reconstruction.

The RNL procedure is explained with a simple example shown in Figure 3.4 with a 2-way set associative cache. At time t, the processor requests an update in cache line 2 of way 1. Since at time t the cache set 2 is in the dumped area, the RNL procedure does not dump the cache line and allows the cache to proceed with the update. At time $(t + t')$, the processor again requests an update this time to cache set 5 at way 1, which falls in Non-dumped area. Therefore, RNL dumps the cache line 5 of way 1 before allowing the cache to update the cache line. Along with cache line data, RNL also dumps the line number '5' and way number '1', which is required for offline re-construction of cache.

The important relevant steps in the cache controller algorithm is explained in Algorithm 5. The algorithm takes as inputs the memory address, write request (w_req), dump request (dump_r) and counter and returns the cache line data. When a cache

Algorithm 5 RNL Procedure

Input: Address, w_req, dump_r, Counter
Output: Cache line data

```
1: if dump_s == TRUE then
2:    for i = 0 to num_ways do
3:       dump-cacheline1(Counter,i)
4:    end for
5:    Counter ⇐ Counter + 1
6: else
7:    TAG ⇐ Address.TAG
8:    index ⇐ Address.index
9:    miss ⇐ TRUE
10:   for  i = 0 to num_ways do
11:      if TAG == cache.TAG(index,i) then
12:         miss ⇐ FALSE
13:         way ⇐ i
14:      end if
15:   end for
16:   if miss == TRUE then
17:      // if miss, the way number is determined through replacement policy
18:      way ⇐ replacement-way(index)
19:   end if
20:   if index > Counter then
21:      if miss == TRUE OR w_req == TRUE then
22:         dump-cachline2(index,way)
23:      else
24:         dump-control-bits (index,way)
25:      end if
26:   end if
27: end if
```

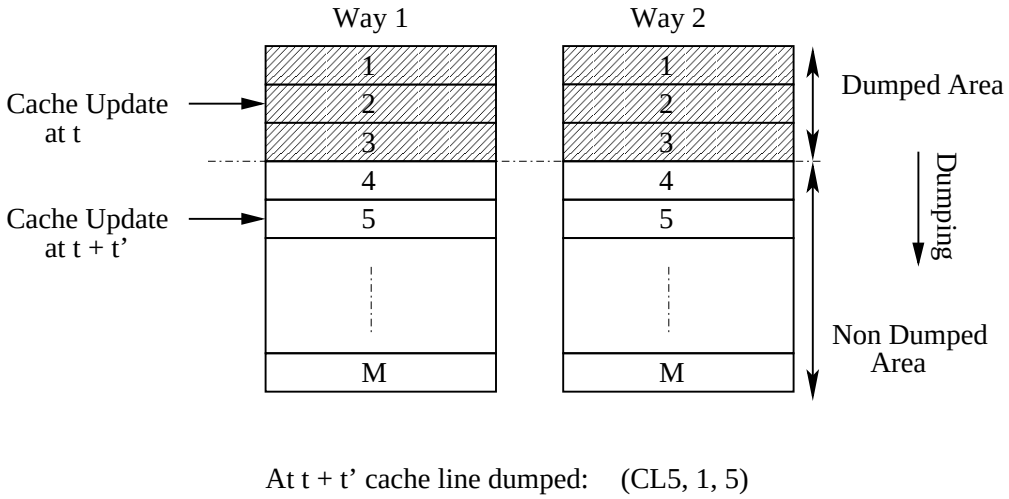


Figure 3.4: RNL Example

set dump is requested, by setting *dump_s* to TRUE, we extract the cache set indicated by the counter (lines 1 to 5 of Algorithm 1). Function *dump-cachline1()* function, used for dumping the cache line, takes the index and way number as input and dumps the corresponding cache line data, consisting of data, tag, and control bits (including any state bits maintained by the replacement policy algorithm, such as LRU bits) to the compression engine. After dumping the cache set the counter is incremented.

If *dump_s* is FALSE (i.e., the processor has requested the cache update), we search the TAG in all ways of the cache set addressed by the index bits. If TAG is found, then the way number is noted. If not, then the cache way to be replaced is decided by the replacement policy of the cache (function *replacement-way()*).

Line 20 of Algorithm 5 checks whether the requested cache update will affect the non-dumped area, by comparing the index with counter. If the index is greater, then we are in the non-dumped area and the existing state needs to be dumped before being updated by the cache write or miss action. We use function *dump-cacheline2()* for this purpose, which is similar to dump-cachline1(), except that, along with the line data, it also dumps out the index and way number, which are required to reconstruct the cache state offline. If we don't have a cache update (e.g, due to a cache READ hit operation) then the cache access can still affect some control bits (e.g., state bits maintained by the LRU algorithm). Hence, to accurately reproduce the cache contents, we capture the

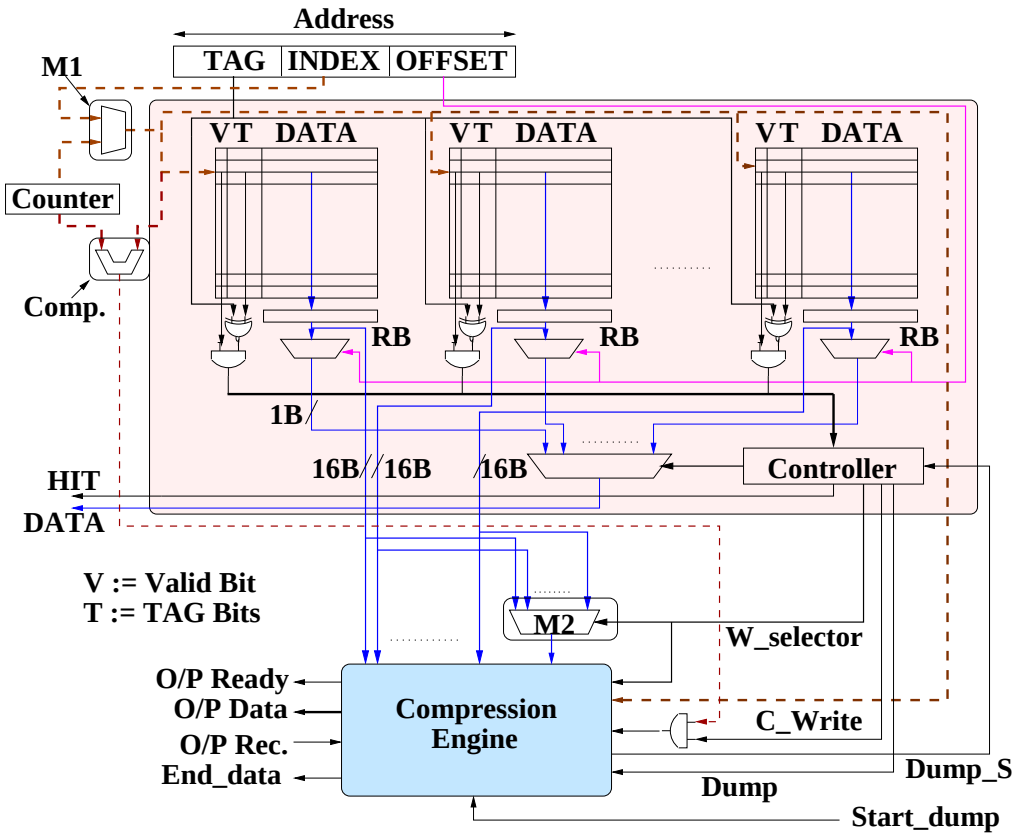
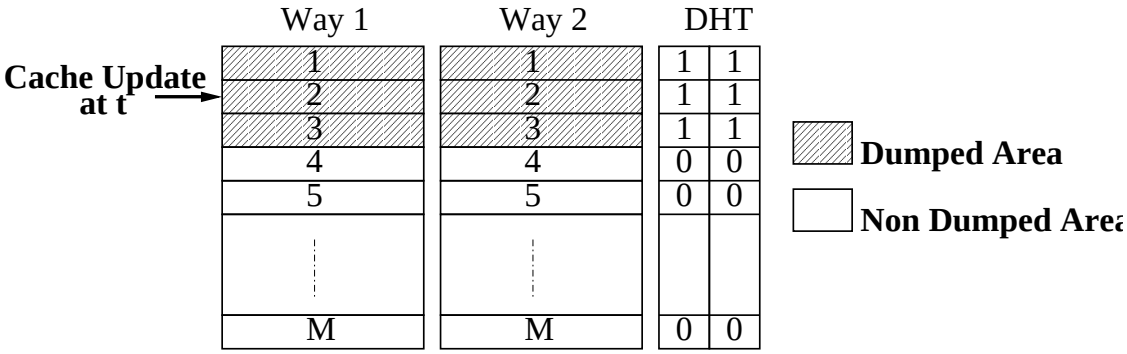


Figure 3.5: Retransmit Non-dumped Line (RNL) Architecture

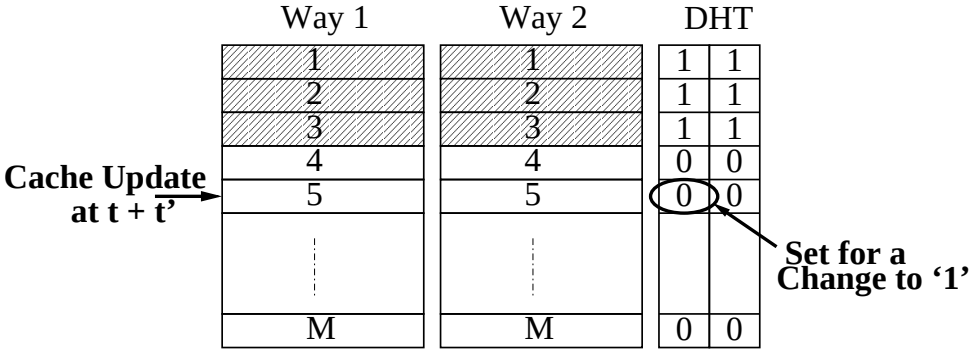
control bits with *dump-control-bits()* function.

Dump History Table (DHT)

Typical application behaviour may cause the processor to update the same cache line multiple times due to locality of reference. When such updates happen in the non-dumped area, the existing cache line is dumped each time the line is updated. Clearly, only the first such dump is useful for our purpose. We propose to maintain a dump history table (DHT) in which we maintain one bit corresponding to each cache line. We set this bit the first time the corresponding line is dumped due to an update in the non-dumped area. Future updates refer to this bit, and on finding this set, avoid the dumping. This leads to reduction in the total dump size at the expense of the area overhead of maintaining the table.



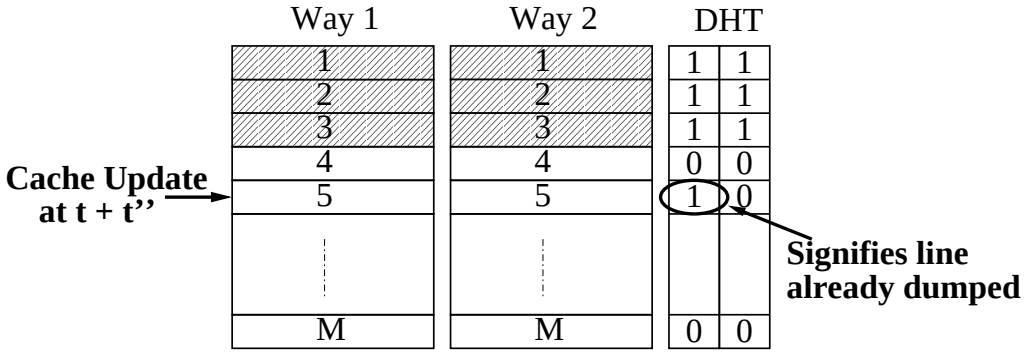
(a) Update in Dumped Area



At t + t' cache line dumped: (CL5, 1, 5)

(b) Update in Non−Dumped Area

t'' > t'



(c) Update in Non−Dumped area at previously updated cached line

Figure 3.6: DHT Example

The DHT cache updating procedure is explained with a simple example shown in

Figure 3.6. In this example we consider a 2 way set associative cache where the contents are dumped set-wise. We also maintain the DHT table which stores the dump information. At time t the processor requests a cache update. Since the update is in the dumped area (cache line 2 in way 1) DHT does not take any action and allows the cache to update (Figure 3.6(a)). After time t', i.e., at time $(t + t')$, the processor makes an update request to line 5 of way 1, which is present in the non-dumped cache area. The DHT procedure checks the corresponding table entry. As the processor makes the update request for the first time to the cache line 5 of way 1, the DHT procedure finds the entry '0' and DHT calls the dump routine to dump the cache line, as shown in Figure 3.6(b). After dumping the cache line, the DHT procedure also sets the corresponding table entry to '1' and allows the cache to update the line. The processor then makes an update request to cache line 5 of way 1 at time $(t + t'')$. Set 5 of the cache is still in the non-dumped area, therefore, the DHT procedure is called (Figure 3.6(c)). Before calling the cache line dump procedure, DHT checks the corresponding entry. Since the cache line was dumped by a previous cache update request, DHT finds the entry '1' and allows the cache to update the cache line.

DHT can be handled with a simple modification in Algorithm 1. The history table entries are initialised to FALSE. The condition (line 20) for calling function dump-cacheline2() is extended by also including a check for whether the corresponding DHT bit is FALSE. If yes, then dump-cacheline2() is called and the DHT bit is set to TRUE.

3.3 Architecture

We discuss here the architecture and implementation issues involved in realizing the above logical alterations in caches.

3.3.1 Modification in Cache Architecture

We attempt to reuse the hardware in the read path of the cache to the extent possible. A counter and comparator shown in Figure 3.5 are used to distinguish between dumped and non-dumped areas. The comparator result is AND'ed with a *C_Write* signal generated by

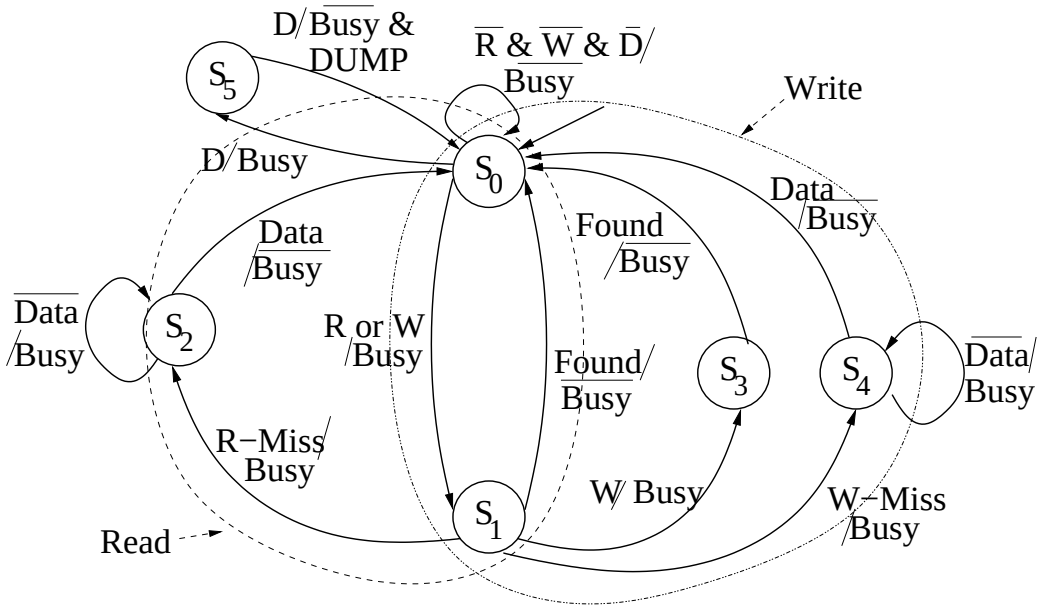


Figure 3.7: Modified Cache Controller FSM

the cache controller (representing a cache update requested by the processor) to enable the compression engine to accept a new cache line, due to cache update. A selector MUX is used to route the read address either from the counter or the index bits.

Figure 3.7 shows a simplified version of the modified cache controller FSM with S_0 being the initial/idle state. On Read or Write request, we proceed to and return from S_1 if there is a hit. Read and Write misses are serviced from states S_2 and S_4. The FSM jumps to a new dump state S_5 when it receives the *Dump_S* (D) HIGH from the compression engine. In S_5, cache lines of the set pointed to by the counter are read into their respective buffers (Figure 3.5) for transfer to the compression engine. After the transfer, the counter is incremented.

The controller generates the following additional signals: (i) *C_Write*, informing the compression engine about updates to the cache that would result in data transfer from the non-dumped area; and (ii) *W_selector*, for selecting the way from where data needs to be dumped.

3.3.2 RNL

Figure 3.5 shows the RNL hardware architecture. On a cache update, the controller sets *C_Write* signal HIGH. Simultaneously, the comparator checks if the index field is greater than counter. If yes, the compression engine fetches the cache line of the addressed way indicated by the *W_selector*. The *W_selector* and index values are also dumped.

3.3.3 DHT

The DHT hardware is similar to the RNL hardware, except for the History Table, as shown Figure 3.8. This history table is checked before a line scheduled for updating is dumped to the compression engine. If the corresponding bit is set, then the dump is suppressed. For an A-way associative cache with N sets, the size of the history table is $N \times A$ bits.

3.3.4 Compressor

The compression engine used in our design was described in Chapter 2.

3.4 Experimental Validation

We extend the framework defined in Section 2.5.1. Along with the cache snapshots, the framework also dumps the cache update trace between consecutive cache snapshots. The update trace contains the trace of cache update instance, cache update address and cache update data.

We use the benchmarks mentioned in Section 2.5 for the experimental evaluation.

3.4.1 Additional Cache Lines Dumped

Additional cache lines are dumped in our mechanism that overlaps dumping with processor execution. The number of additional lines dumped is an important metric for comparison of the cache update handling strategies: RNL and DHT. Figure 3.9 shows

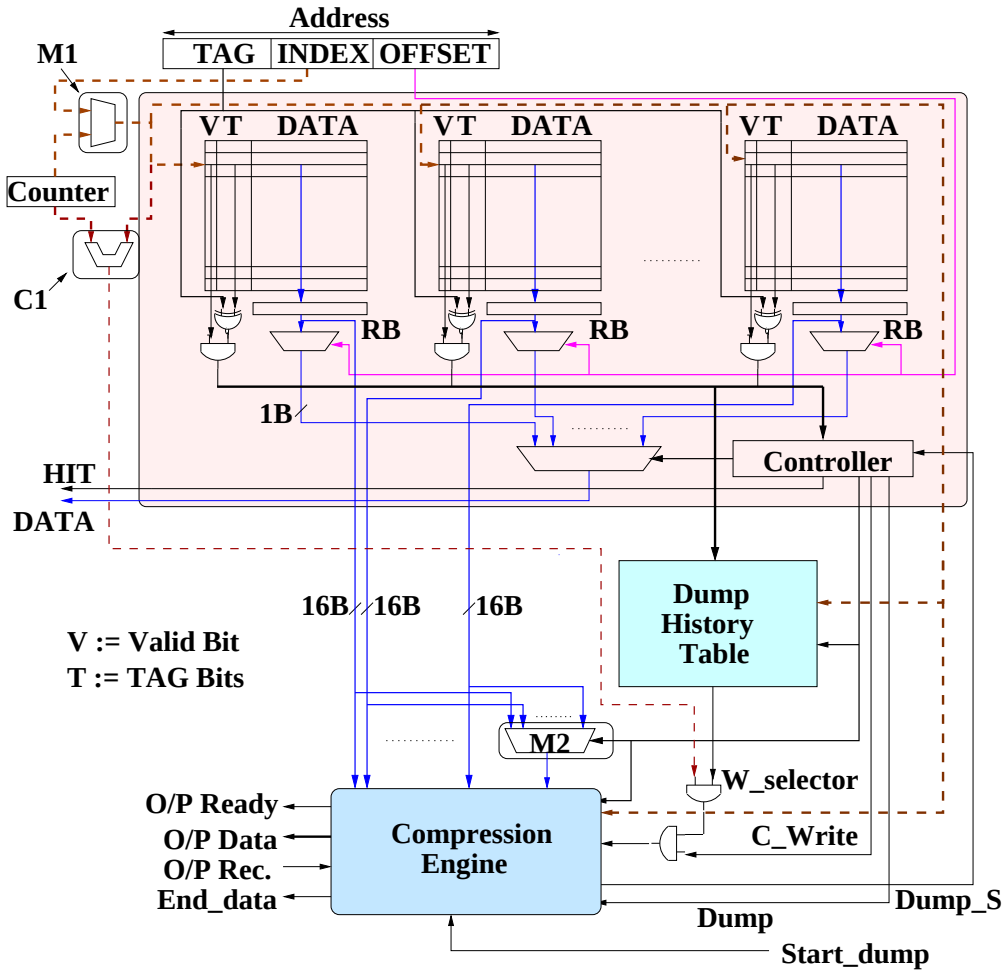


Figure 3.8: Dump History Table (DHT) Architecture

the number of actual cache updates (per 1000 instructions) and the additional cache lines dumped due to updates in the non-dumped area for both strategies RNL and DHT.

On an average, RNL dumps 31% fewer additional cache lines than the actual number of updates, and DHT dumps a further 96% fewer lines. The *BZIP2* and *CHESS* examples exhibit the largest differences between RNL and DHT (66% and 86% respectively). For the remaining benchmarks, DHT always dumps significantly fewer additional cache lines compared to RNL.

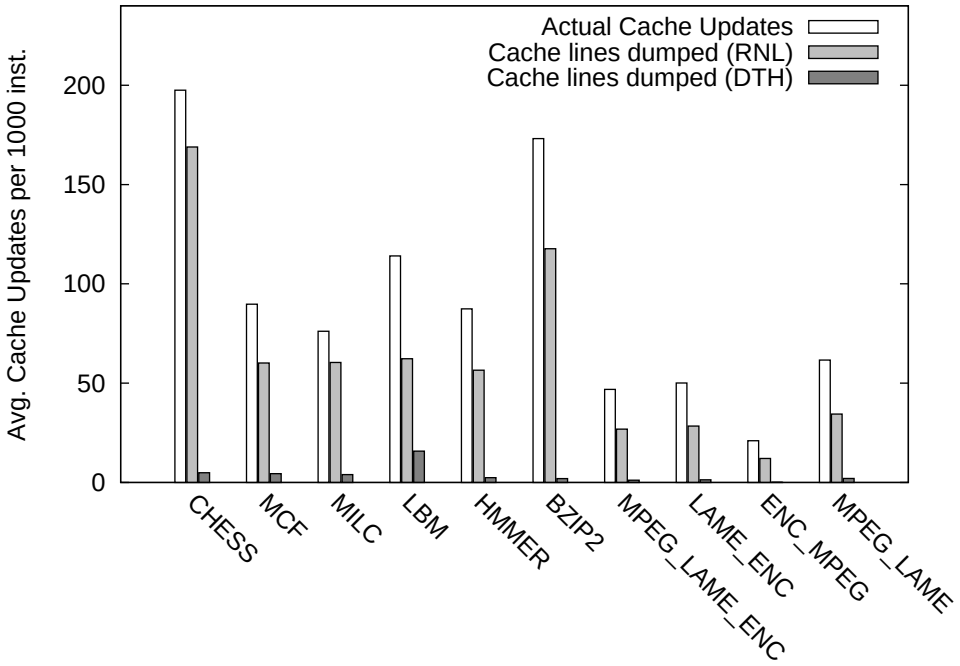


Figure 3.9: Average # Cache line Dumps (due to Cache Update)

3.4.2 Saved Debug Time

Since L2 cache misses, which lead to traffic over the memory bus, are relatively rare (average of 0.6% for our benchmarks), the overlap with the cache dump leads to a relatively small number of wasted cycles. We call this the effective dump time, since this is the overhead incurred by the program execution due to dumping. Column 3 in Table 3.1 shows the effective dump time for the benchmarks. Column 4 shows this information as a percentage of the cache dump time if the processor was stalled for dumping; the cache configuration used results in a dump time of 6.12×10^5 cycles. We note that the effective dump time is very small – 0.01% to 3.5%, with the average being 0.67% – compared to stalling the processor for the entire cache dump duration.

3.4.3 Compression Results

Columns 5 and 6 of Table 3.1 shows the compression ratios (computed as $(1 - O/I) \times 100\%$, where O is the number of output bytes of the compressor and I is the cache dump size) obtained by two hardware implementations of compression algorithms; LZW_SLU

(hardware implementation described in Chapter 2) and X-Match [15], of the cache data dumped by RNL and DHT. We observe that, DHT always outperforms RNL on an average with 8.25% and 10.23% more compression with LZW-SLU and X-Match respectively. The maximum gain of 17% in compression ratio is noticed in *BZIP2*. Also, LZW-SLU gives 15% and 13% more compression than X-Match for RNL and DHT respectively.

Benchmarks	*Comp.* Algo.	*Eff. Dump Time*		*Compression Ratio*	
		N $\times 10^3$ CPU Cycles	Perc. (%)	*RNL*	*DHT*
Chess	LZW-SLU	1.44	0.22	62.29	77.24
	X-Match	2.25	0.37	40.22	56.32
MCF	LZW-SLU	19.02	3.11	47.77	56.49
	X-Match	21.20	3.46	30.31	44.88
MILC	LZW-SLU	2.50	0.41	27.59	35.47
	X-Match	8.75	1.43	21.10	33.67
LBM	LZW-SLU	4.95	0.81	86.36	92.11
	X-MATCH	12.08	1.97	55.70	69.95
HMMER	LZW-SLU	6.66	1.09	36.32	43.02
	X-Match	6.63	1.08	34.75	47.86
BZIP2	LZW-SLU	5.80	0.95	11.76	29.26
	X-Match	5.41	0.88	8.47	10.32
MPEG_LAME	LZW-SLU	0.09	0.01	53.95	59.69
	X-Match	0.12	0.02	32.86	47.15
Average	LZW-SLU	4.07	0.67	**50.90**	**59.15**
	X-Match	5.68	0.93	**35.77**	**46.00**

Table 3.1: Dump Time and Compression Results

3.4.4 Implementation and Synthesis Results

We synthesized VHDL descriptions of the new cache controller and compression engine with Synopsys Design Compiler with a 180nm library, and used estimations from CACTI [43] for cache and SRAM components.

For our targeted cache of 512KB, 16 byte cache line and 4 way associativity in 180 nm, the CACTI estimates are: 2.63 ns for access time and 38.9 mm^2 for area. The area difference for the synthesized cache controller with and without our proposed modifica-

tion is 0.0002 mm^2, which is negligible in comparison to the cache area. The critical path of both synthesised modules are almost identical.

The area impact on the cache with the modification required for the RNL, is negligible in comparison to the cache area, as only a few muxes, a counter, and controller gates are needed, as shown in Figure 3.5. DHT requires an SRAM module for implementing the history table along with the controller modification. The estimated area of the history table is 0.15 mm^2 and access time 1.47 ns. Since the table is accessed in parallel with the cache access and the latter dominates the delays, its access does not cause any delay overhead.

For the LZW-SLU implementation we used 4 parallel compressors for the data field of the cache, 1 compressor for the additional cache lines dumped due to cache update and 1 compressor for TAG and ECC fields. The compression engines have two input buffers with size equal to the cache line data size and two output buffers of 256 bytes.

3.4.5 Limitations

The following limitations still exist with our proposed strategy. The overlapping of dumping and execution places an upper bound on the interval between successive dumps; it cannot be smaller than the dump duration. If a smaller duration is needed, we can either use multiple compression engines, or, in the worst case, revert to the original mode of execution and dumping in sequence.

Finally, dumping the state of other components carrying state information closer to the CPU, such as pipeline registers, register file, L1 cache, TLB, reservation station etc., still requires the CPU to be halted. Nevertheless, these components represent a relatively small fraction ($< 7\%$ in our example) of the total processor state.

3.5 Conclusion

We proposed and evaluated a mechanism for overlapping processor execution with transfer of cache contents off-chip for debug purposes with the objective of reducing the total debug cycle time, which includes both processor execution and internal state transfer

time. In order to prevent corruption of the cache state to be dumped by the executing processor, we proposed two alternatives based on the overall plan of explicitly transferring a cache line not already dumped before a regular cache update is permitted to occur. Experimental evaluation shows that the mechanism reduces the effective L2 cache dump time to between 0.01% and 3.5% of the original duration.

Chapter 4

Incremental Cache Dumping

4.1 Introduction

In the cache dump techniques discussed earlier, we dump the whole cache content, where as a relatively small portion of the cache content may be changed and contain interesting information since the previous dump sequence. The dumping of the complete cache leads to wastage of precious debug time and memory (in the logic analyzer), particularly since this action is repeated a large number of times. In this chapter we attempt to save the dump time and data size with *Incremental Cache Dumping*: we dump only the portion of the cache that was updated since the last dump sequence. For analysis the whole cache structure is reproduced offline with the help of the previous cache dumps which contain the information for the remaining cache state.

The key point of incremental cache state dumping is to keep track of all cache updates (due to write requests and cache misses) between two consecutive dumps. We propose the maintenance of an *Update History Table (UHT)* for cache update information, where each entry represents the update status of a cache line. We study the cost-performance trade-offs involved here, varying the number of cache lines represented by one UHT entry.

Based on processor execution status, we present two methodologies for incremental cache dumping: 1) *Blocking Incremental Cache Dumping (BICD)*, and 2) *Non-blocking Incremental Cache Dumping (NICD)*. In BICD the processor execution is stopped during the cache dump. In NICD the cache dumping is performed simultaneously with the pro-

cessor execution overlap. There are two main challenges with the NICD: 1) Maintenance of the fidelity of the dumped cache data, which could get corrupted by the executing processor; and 2) Maintenance of the UHT table which could get incorrectly updated with cache state dumping by the executing processor. The first problem is solved by dumping of the cache line before the cache attempts to update it. The second problem – maintenance of UHT – is solved by the use of two UHTs, where one UHT contains the cache update information before cache dumping starts and the other UHT tracks the cache updates after start of the cache dumping sequence. The dumped cache content is later taken through the compression unit, described in Chapter 2, to reduce the transfer time and logic analyzer space requirements.

Since the program's locality of reference is likely to result in update of consecutive cache lines in a clustered manner, we studied the possibility of reducing the UHT overhead by considering the UHT entry for maintenance of update information in a block of cache lines.

This chapter is organised as follows: Section 4.2 describes the algorithmic details of Incremental Cache Dumping. Architectural details of Incremental Cache dumping are discussed in Section 4.3. Experimental validation is discussed in Section 4.4. Section 4.5 presents the concluding observations.

4.2 Incremental Cache Dumping

The main objective of incremental dumping is to reduce the total amount of L2 cache data to be transferred off-chip by dumping only the cache lines that are updated since the last cache dump. The key idea here is to track all cache updates between two consecutive dump sequences using an Update History Table (UHT). Each UHT entry represents this update information for a corresponding cache line.

Due to locality of reference present in the program, it is highly probable that write requests update consecutive cache lines in clusters. Therefore, we made the study of cost-performance by varying the number of lines that a UHT entry represents. Other references to the cache may modify some control bits (valid, dirty, ref, etc.), and for

analysis it is very important to dump those control bits also. The same procedure can also be applied to the control bits. The alterations/modifications in cache control bits can be tracked using separate UHTs. Alternatively, we can dump all the control bits first before initiating the dumping of the other cache components. In Section 4.4 we present the analysis considering both strategies for dumping the control bits.

4.2.1 Blocking Incremental Cache Dumping (BICD)

In Blocking Incremental Cache Dumping the processor is halted during the transfer of the cache content. During the cache dump, we dump only those cache lines (including TAG, ECC bits, etc.) whose corresponding UHT entry is set.

The central idea of BICD is explained with an example shown in Figure 4.1, with a cache containing 8 lines. The shaded cache lines have been updated since the last cache dump. Traditionally, we would dump all the cache lines without checking their update status, as shown in Figure 4.1 (a). Such dumping leads to redundancy, as only updated cache lines hold interesting information about the processor execution since the last dump sequence. We maintain the cache line update information in an Update History Table (UHT) and dump the updated cache lines with prefix '1'. For non-updated cache lines we send only '0'. The output from the cache with BICD would be as follows:

0, (1,B), 0, 0, (1,E), (1,F), 0, 0

With the above output we dump only 3 cache lines (B,E,F) out of 8 cache lines. This leads a saving of 61.7% for a typical 16 byte cache line.

In this case 8 bits are used for the UHT. It is possible to use a smaller UHT by having each UHT bit represent a window of more than one cache line. For the above example, with a window size of 2 lines, shown in Figure 4.1(c), we require only 4 bits in the UHT table with the following output.

(1,A,B), 0, (1,E,F), 0

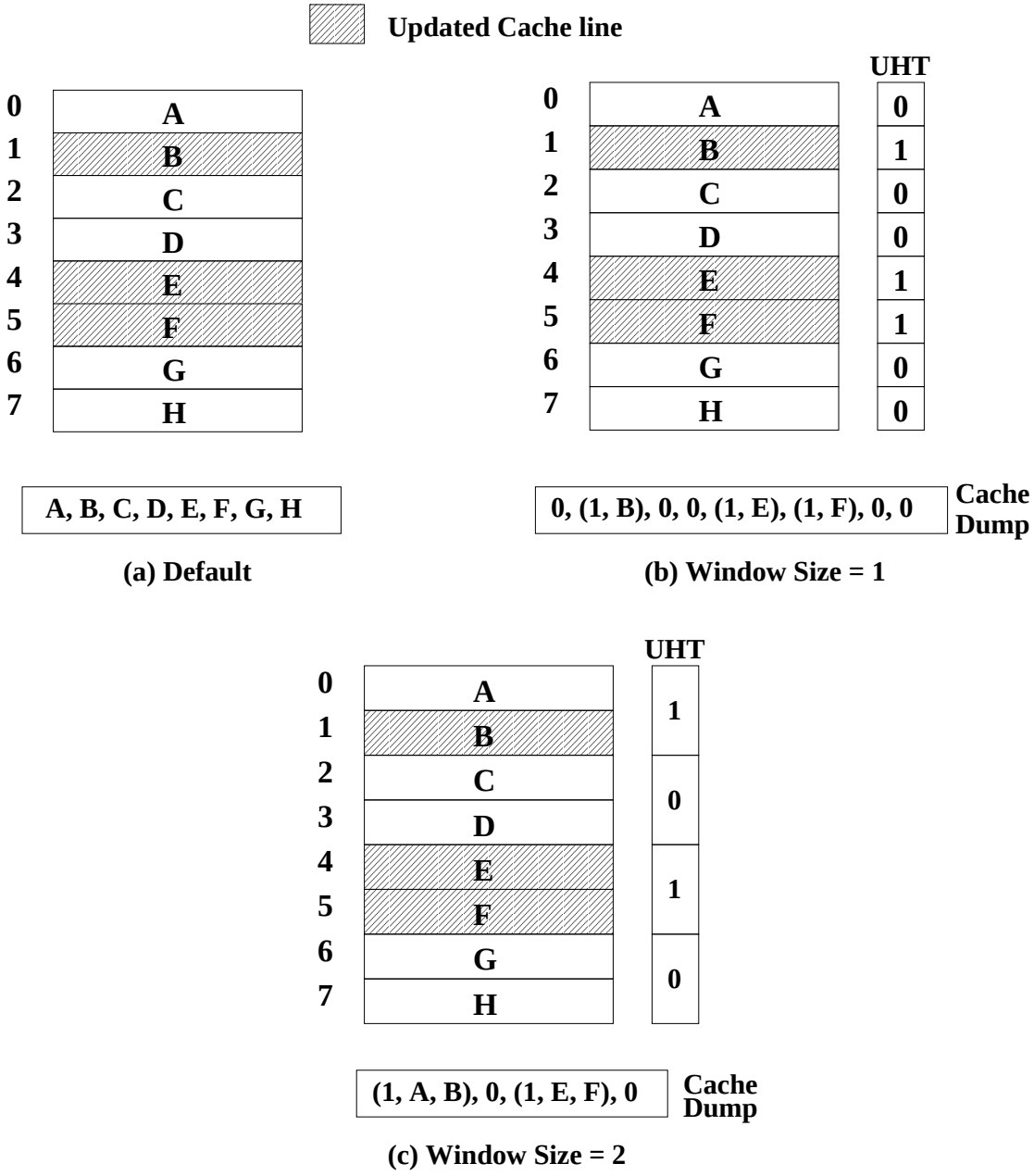


Figure 4.1: Example of Blocking Incremental Cache Dumping (BICD)

Introducing a window of cache lines helps when locality in program results in updates to consecutive cache lines within a dump interval. In Figure 4.1(c), updated consecutive cache lines **E** and **F** are dumped with one bit prefix. On the other hand, windowing may lead to extra cache lines being dumped. Lines **A** and **B** are both dumped, where only one of them, **B**, is updated. The best choice of cache line window size depends on

Algorithm 6 BICD Procedure

Input: dump_r
Output: Cache line data, end_data

```
1: window_number ⇐ 0
2: if dump_r == TRUE then
3:   if window_number ! = Number of sets in a cache / Number of sets in a window then
4:     window_number ⇐ window_number + 1
5:     for way_number = 0 to #ofways do
6:       if UHT[window_number][way_number] == 1 then
7:         for line_counter = 0 to window_size - 1 do
8:           dump_line(window_number,line_counter,way_number)
9:         end for
10:        UHT[window_number][way_number] ⇐ 0
11:      else
12:        dump_UHT_bit()
13:      end if
14:    end for
15:  else
16:    end_data ⇐ 1
17:    window_number ⇐ 0
18:  end if
19: end if
20: return
```

the program running on the processor and also on the phase within a program (more in Section 4.4).

Algorithm 6 describes the BICD procedure. Input to the Algorithm 6 is *dump_r*. When a cache line set is requested, by setting the *dump_r* TRUE, we first check *window_number* value. If we have not reached the end of the table, then we dump the block of cache lines of the way, according to the corresponding UHT entry (lines 2 to 7 of Algorithm 6). If the UHT entry is LOW then we call *dump_UHT_bit* to dump the UHT bit value at the output.

Function *dump_line* is used to dump the cache line. It takes window_number, line_counter and way_number as input and dumps the corresponding cache line data (including data, Tag, ECC bits and other bits) prefixing with the corresponding UHT value.

When all the relevant cache lines are dumped, we reset the window_number and set end_data to '1', signifying the end of the dumping procedure.

4.2.2 Non-blocking Incremental Cache Dumping (NICD)

In Non-blocking Incremental Cache Dumping, we dump the cache content with processor execution overlap. The executing processor can update the cache during the dump procedure, which may corrupt the cache state and UHT. Hence, the key issues to be addressed in NICD are: 1) proper handling of cache updates and 2) maintenance of UHT. The cache updates are handled by dumping the cache lines before a cache update, if the corresponding UHT entry is HIGH. As we dump the cache lines in the order of increasing cache set number, the cache divides logically into two parts: 1) *Dumped area* and 2) *Non-dumped area*. The cache updates in the Non-dumped area only are of interest, with respect to the current cache state. We dump the block of cache lines corresponding to the cache line update in non-dumped area, and after dumping the block we reset the corresponding UHT entry. This ensures that a cache line is dumped only once.

When a processor request updates the cache, the Incremental Cache Dumping procedure leads to a UHT update. This may lead to corruption of the cache state. We solve this problem by using two UHT's. At any instant, one UHT represents the cache updates before dumping starts, while the other maintains the cache update after the current dump sequence starts. At the start of the next dump, both the UHTs swap their roles.

NICD is explained with an example shown in Figure 4.2. We have 8 cache lines with the shaded lines representing those updated since the last dump. For simplicity, we show UHT with window size 1. NICD uses two UHTs – UHT-P and UHT-C. UHT-P contains the information about the cache updates since the last dump. UHT-C maintains the information about the cache updates during the current dump interval.

Figure 4.2(a) shows the cache and UHT state at the start of the cache dump. All UHT-C entries are 0s and UHT-P entries are '0' and '1' according to whether the cache line was updated in the previous intervals.

When the dump sequence starts, we dump '0' representing non-updated cache line **A** and we dump the updated line **B** with prefix '1' and reset its corresponding UHT-P entry. While the cache transfers line **B** to the logic analyzer, the processor requests an update to the updated cache line **F** in non-dumped area. Before the cache updates F,

NICD dumps the line with the cache line number and resets the corresponding UHT-P entry and sets the UHT-C entry. The updates are shown in Figure 4.2(b).

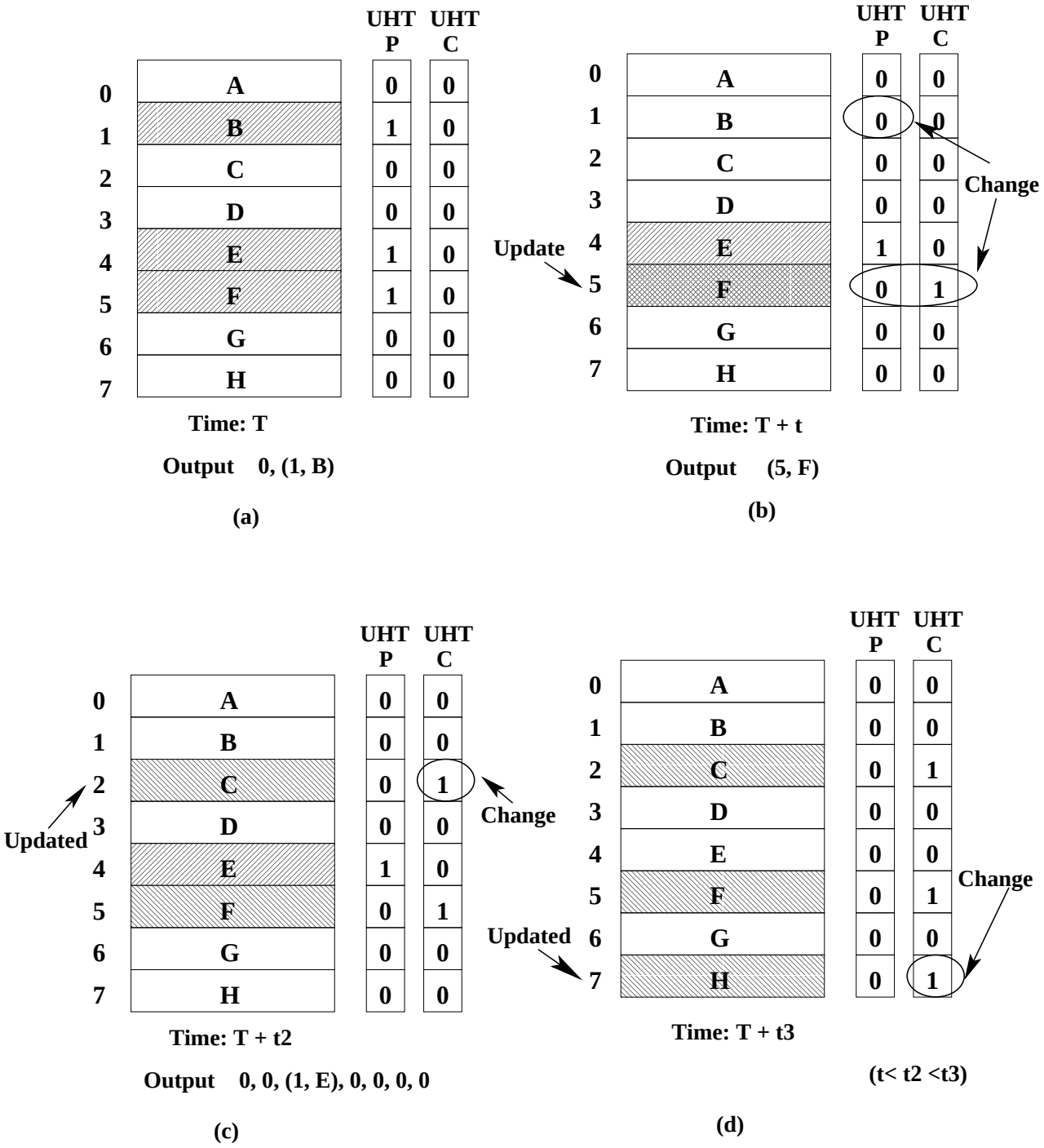


Figure 4.2: Example of Non-blocking Incremental Cache Dumping (NICD)

Figure 4.2(c) shows the cache and UHT state at time $t2$ after dumping starts. The dump is as follows:

0,0,(1,E),0,0,0,0

The key point to notice here is, for initial corrupted cache line **F**, we dump only '0'. This happens as NICD takes care of cache lines already dumped due to requested cache update. Between T to $T + t2$ we have one more change – an update to cache line **C**, shown in Figure 4.2(c). Since this change does not affect the current dumping state of the cache, NICD does not dump the cache line. Figure 4.2(d) shows the state of the cache and UHTs when the cache is ready for the next dump, with one more updated cache line **H**. Now for the next cache dump sequence, the UHT-P will capture the current cache update and UHT-C will provide information for cache dump.

The NICD algorithm is discussed in Algorithm 7. The algorithm takes Address, w_req (write request) and dump_r (dump request) as an input and returns the cache line data. Lines 2 to 18 of Algorithm 7 are similar to Algorithm 6 except line 9 of Algorithm 7, where we reset UHT1 (the current dumping UHT) entry of the dumped cache lines window.

If dump_r is FALSE and a request is made to the cache, we search TAG in the cache. If the Tag is found, then the way of the block to be dumped is determined by the way number. If not, then we have a miss and the way to be replaced is decided by the replacement function *replacement-way*.

Line 31 checks the miss and write request (w_req HIGH). If either (or both) is HIGH, we compute the table_line which is computed as index/window_size. If the UHT1 entry corresponding to the table_line and way is HIGH, we dump the block of cache lines and reset the UHT1 entry using the *dump_line1* function (line 33 to 35). In line 39 we store the cache update information in the UHT2 for the next sequence cache dump.

Function *dump_line1* prefixes the address information of the cache line window with the dump stream. This is necessary to reconstruct the cache, as the function will be called randomly depending upon the cache updates.

Algorithm 7 NICD Procedure

Input: Address, w_req, dump_r
Output: Cache line data, end_data

```
1: //UHT1 provides info. for cache dumping & UHT2 captures current cache updates
2: if dump_r == 1 then
3:    if window_number ! = Number of sets in a cache / Number of sets in a window then
4:       window_number ← window_number + 1
5:       for way_number = 0 to #ofways do
6:          if UHT1[window_number][way_number] == 1 then
7:             for line_counter = 0 to window_size - 1 do
8:                dump_line(window_number,line_counter,way_number)
9:                UHT1[window_number][way_number] ⇐ 0
10:            end for
11:         else
12:            dump_UHT_bit()
13:         end if
14:      end for
15:   else
16:      end_data ⇐ 1
17:      window_number ⇐ 0
18:   end if
19: else
20:   TAG ⇐ Address.TAG
21:   index ⇐ Address.index
22:   miss ⇐ TRUE
23:   for i = 0 to num_ways do
24:      if TAG == cache.TAG(index,i) then
25:         miss ⇐ FALSE
26:         way ⇐ i
27:      end if
28:      if miss == TRUE then
29:         way ⇐ replacement-way(index)
30:      end if
31:      if miss == TRUE OR w_req == TRUE then
32:         table_line ⇐index / window_size
33:         if UHT1[table_line][way] == 1 then
34:            for line_counter = 0 to window_size do
35:               dump_line1(table_line,line_counter,way)
36:            end for
37:            UHT1[table_line][way] ⇐ 0
38:         end if
39:         UHT2[table_line][way] ⇐ 1
40:      end if
41:   end for
42: end if
43: return
```

4.3 Architectural Details

Here we discuss alterations in the cache for realizing the incremental cache dumping hardware.

4.3.1 Modification in Cache Architecture

We attempt to reuse the cache architecture to the extent possible. On a Read operation the cache line data is read into the read buffer. We extend this path to the compression engine, shown in Figure 4.3, to get the cache data. The following additional signals are extracted from the cache controller: 1) *Write* – signifies the updates in cache, and 2) *W_sel* – the way information where the cache needs to update. The signals *Dump_s* (Start dump) and *Dump* (cache line dumped) are already present in the cache controller to facilitate the cache dumping procedure.

4.3.2 BICD Architecture

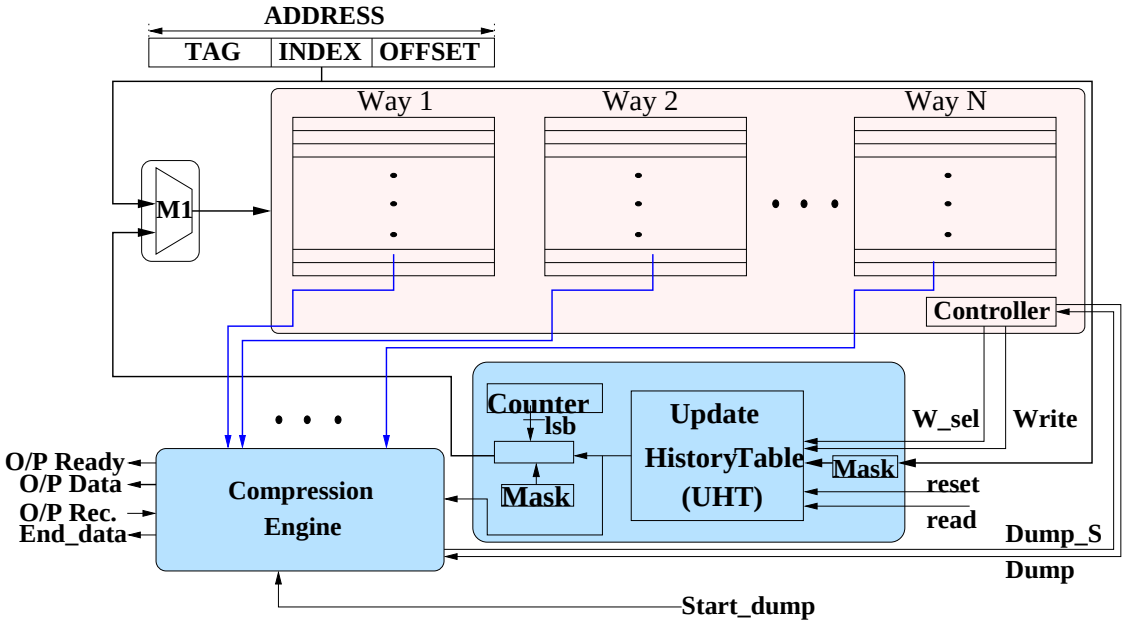


Figure 4.3: Blocking Incremental Cache Dumping (BICD) Architecture

The architecture of BICD, shown in Figure 4.3, consists of UHT (SRAM), counter,

mask registers and concatenator. The counter is used to generate the index lines for the cache line addressing. The mask register is used to mask the index bit, to generate the address of the cache update window. The size of counter and mask registers depend on the window size. If the number of cache lines is N, and the number of cache lines in a window is W, the required size of UHT is N/W bits.

On a cache update the cache controller sets the signals *Write* and *W_sel*. On receiving the *Write* signal, UHT sets the entry HIGH corresponding to the masked index and way represented by *W_sel* signal.

On receiving the *Start_dump* signal, the compression engine sets the *Dump* signal HIGH. After receiving the *Dump* signal the cache controller activates the *Dump_S* signal when data is ready for dumping. We also have signals *I_Way* from UHT to the compression engine. The *I_Way* signals carry the address information of the cache line block. On sensing the valid *I_way* signals and signal *Dump_S* the compression engine downloads the data from the data lines into its internal buffer for compression.

4.3.3 NICD Architecture

Figure 4.4 shows the NICD architecture which is similar to the BICD architecture except that it has one extra UHT and some decision making MUX'es. The configurations of the NICD components are similar to the BICD components.

For explanation, we use term UHT1 for the UHT involved in the current cache dump and UHT2 for the UHT that gathers the current cache update. On cache update, after receiving the *Write* signal UHT2 gets the update and UHT1 checks the entry corresponding to masked index and *W_sel*. If the entry is HIGH, it dumps the block of cache lines and resets the entry. Otherwise, it does nothing.

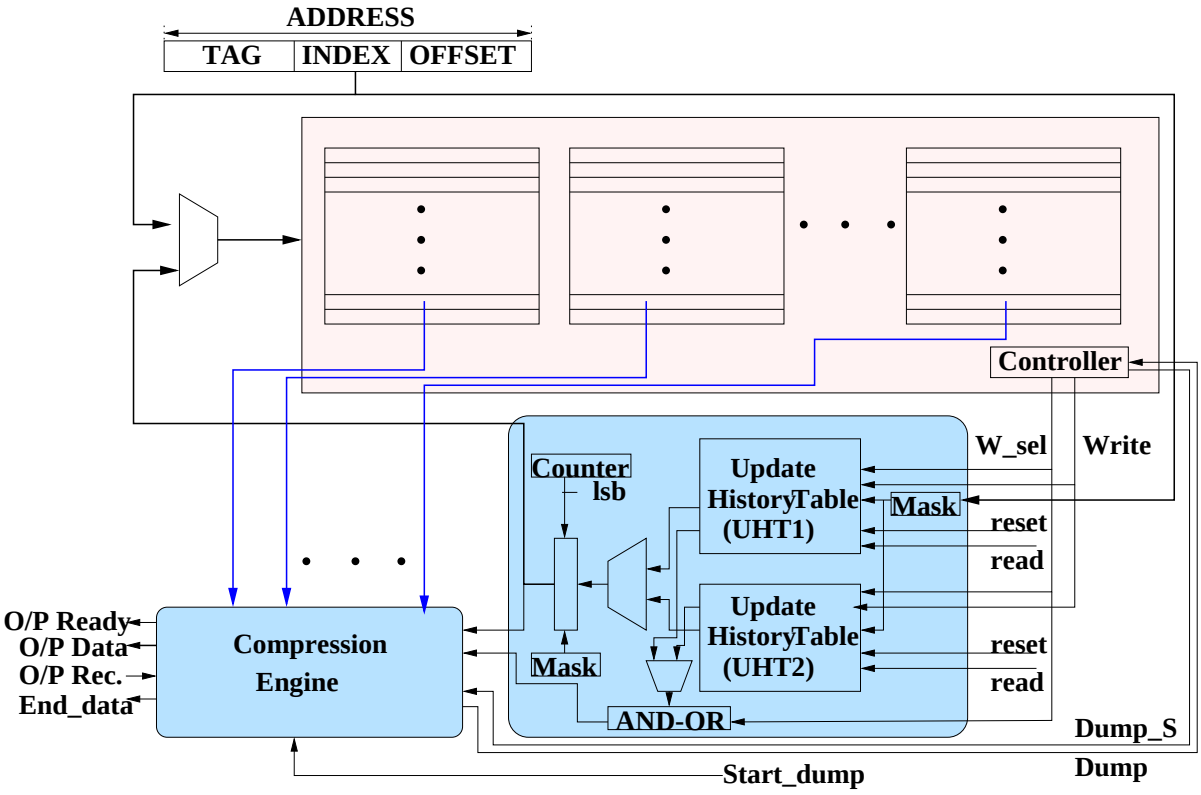


Figure 4.4: Non-blocking Incremental Cache Dumping (NICD) Architecture

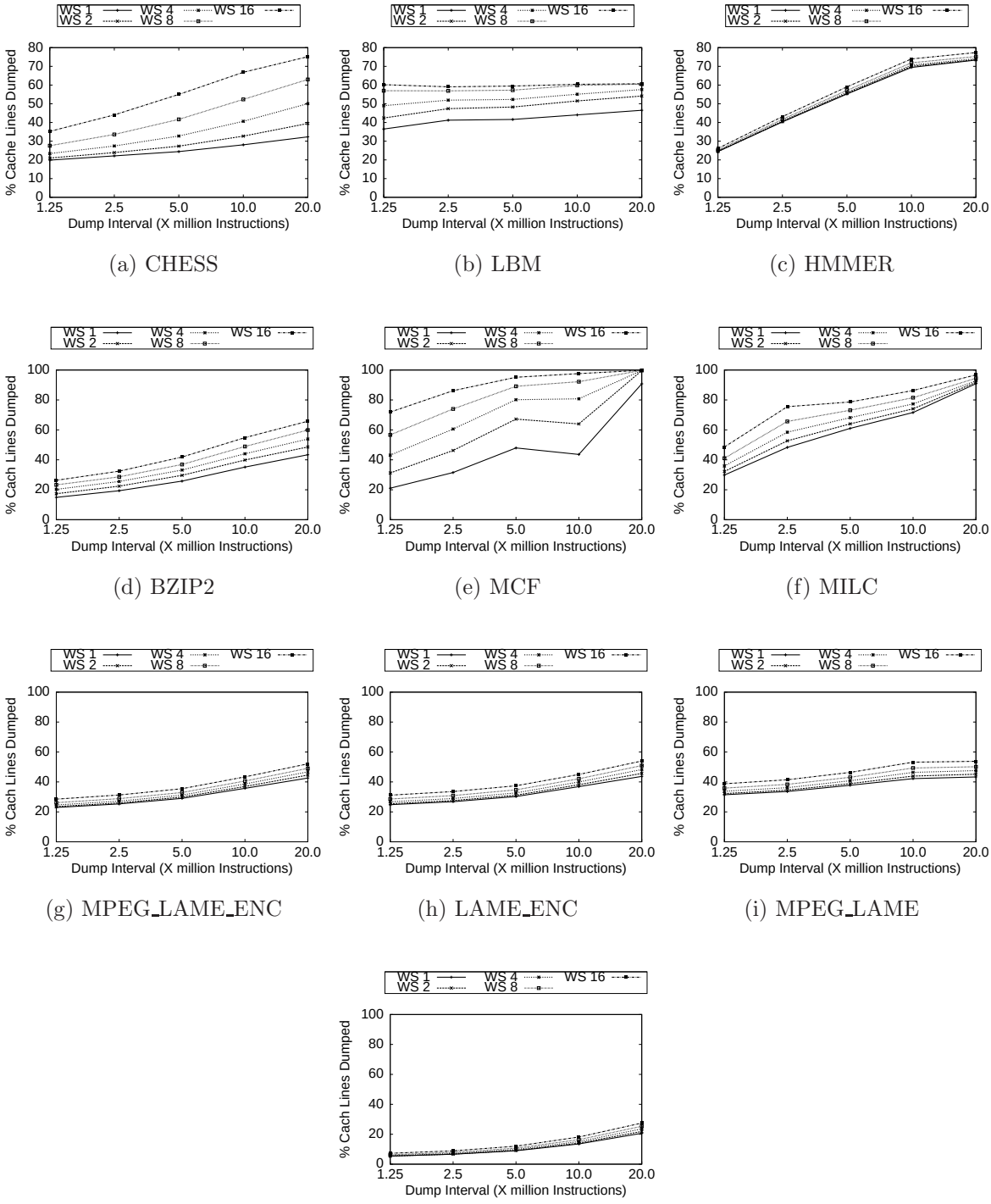
4.4 Experimental Evaluation

4.4.1 Generation of Experimental Data

We extend the framework described in Section 2.5 for our experiments. We also dump the trace of cache update between two cache dumps, along with cache snapshot. The cache update trace contains the time stamp of cache update, cache line address (way number and set number), requested address, updated data and nature of cache update (due to write request or miss).

We use the benchmarks mentioned in Section 2.5 for our experiments.

4.4.2 Cache Lines Dumped



(a) CHESS (b) LBM (c) HMMER

(d) BZIP2 (e) MCF (f) MILC

(g) MPEG_LAME_ENC (h) LAME_ENC (i) MPEG_LAME

(j) MPEG_ENC

Figure 4.5: % of Cache lines Dumped at Various Dump Intervals

We present our observations on the percentage of cache lines dumped with varying dump intervals from 1.25 to 20.0 million instructions in Figure 4.5. These graphs also present a comparison of the number of cache lines dumped when the cache is divided into windows of cache lines varying from 1 to 16 (in power of 2, Window Size = WS). As we increase the dump interval, more cache line windows are updated and the number of cache lines dumped increases in Figure 4.5

The CHESS benchmark finds the next move by exploring a tree of variation nodes created in random order. This access pattern leads to random cache line accesses and we have an increase in the cache lines dumped with window size and dump interval (Figure 4.5(a)). We noticed an average percentage of cache line dumped for window size 1 and 2 are 25.37% and 28.88% respectively.

The LBM benchmark simulates the incompressible fluids in 3D and reads the obstacle file into fixed buffers. We noticed lesser change in the cache lines dumped with respect to the dump interval (Figure 4.5(b)). The buffers are created in memory in such a way that it leads to an increase in the number of cache lines dumped with increase in the window size. For the window sizes 1 and 2, we noticed that an average of 41.98% and 48.74%, respectively, of the total cache lines got dumped. A similar pattern is observed for benchmarks MILC, MCF, and BZIP2. For MCF we observed an average of 46.98% and 61.66% of cache lines dumped for window sizes 1 and 2, respectively (Figure 4.5(e)). Similarly, we noticed that for window sizes 1 and 2, 59.69% and 62.09% of cache lines are dumped for MILC (Figure 4.5(f)) and 27.71% and 31.56% are dumped for BZIP2 (Figure 4.5(d)).

HMMER, a sequential pattern matching benchmark from the Computational Biology domain, maintains a buffer and most of the updates happen in it. We noticed almost the same number of cache lines dumped with change in cache line window size (Figure 4.5(c)). In a small time window, it accesses a small portion of the buffer, which results in relatively fewer cache lines dumped. We notice an average of 52.56% and 52.81% of cache lines dumped for window sizes 1 and 2 respectively.

Media Benchmarks are streaming applications which process the data in blocks. Therefore, we notice for all Media benchmarks a small variation in percentage of number

of lines dumped with respect to window size (Figures 4.5(g), 4.5(h), 4.5(i), 4.5(j)). For MPEG_LAME_ENC, LAME_ENC, MPEG_LAME and MPEG_ENC we notice an average of 32.1%, 33.2%, 33.48% and 11.04% and 35.28%, respectively, of cache lines dumped for window size 1.

We notice across all benchmarks on an average of only 36% (i.e. 64% reduction) of the total cache lines dumped for window size 1. CHESS, LBM and similar benchmarks have the lowest averages for window size 1. For benchmarks like HMMER and Media benchmarks, the difference in cache lines dumped is minimal with respect to window size. To minimize the overhead of the UHT table, we select the window size 16. It is clear that the best choice of the actual cache line window size depends on the program running on the hardware.

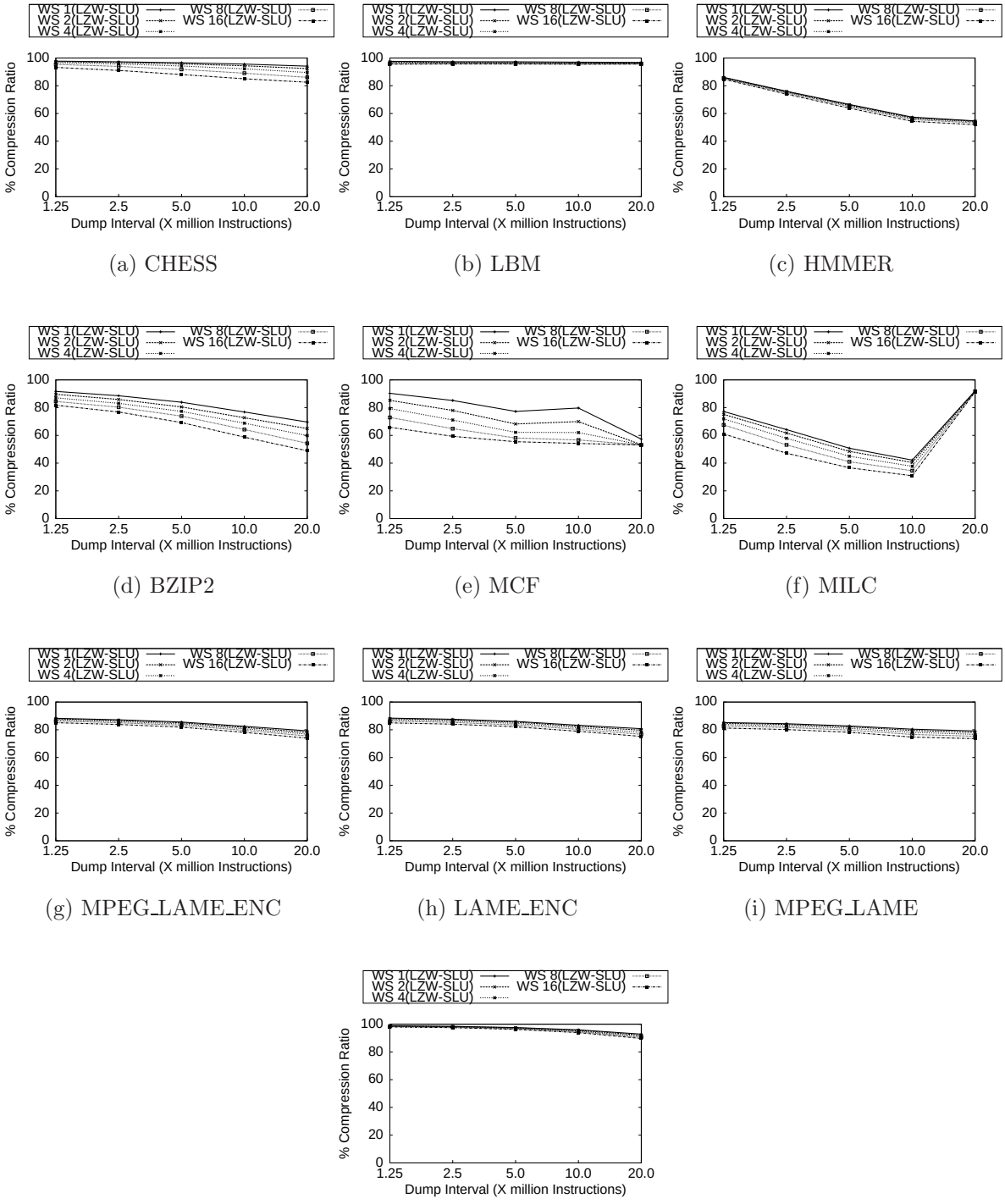
4.4.3 Compression Results

BICD Compression Results

Figure 4.6 shows the compression ratio for the BICD methodology. Since the dump time for BICD depends on the time taken to transfer the compressed data, we used LZW-SLU in the compressor. Here compression ratio directly depends on the lines to compress. Figures in 4.6 follow similar trends with respect to cache lines dumped. An increase in window size leads to more cache lines to compress, and we noticed a decrease in the compression ratio.

Benchmark	Window Size 1	Window Size 2	Figure
CHESS	82.13%	78.12%	4.6(a)
LBM	82.39%	81.68%	4.6(b)
HMMER	77.96%	70.94%	4.6(c)
BZIP2	84.57%	83.90%	4.6(d)
MCF	96.73%	96.43%	4.6(e)
MILC	97.21%	96.71%	4.6(f)
MPEG_LAME_ENC	96.73%	96.44%	4.6(g)
LAME_ENC	85.22%	84.76%	4.6(h)
MPEG_LAME	82.39%	81.68%	4.6(i)
MPEG_ENC	96.73%	96.43%	4.6(j)

Table 4.1: Average Compression ratio (%) for BICD



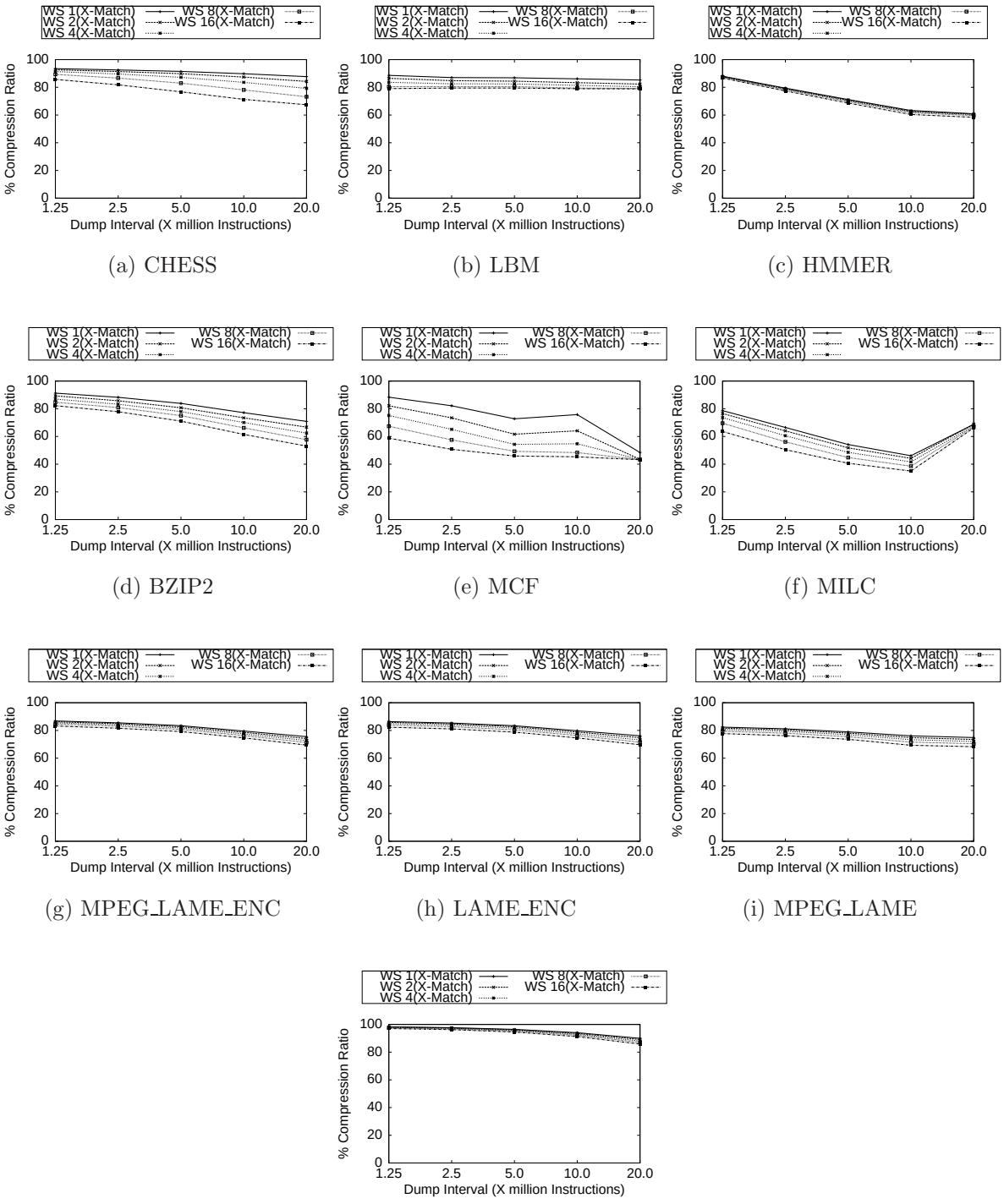
(a) CHESS (b) LBM (c) HMMER

(d) BZIP2 (e) MCF (f) MILC

(g) MPEG_LAME_ENC (h) LAME_ENC (i) MPEG_LAME

(j) MPEG_ENC

Figure 4.6: Compression Ratio for BICD (LZW-SLU)



(a) CHESS (b) LBM (c) HMMER

(d) BZIP2 (e) MCF (f) MILC

(g) MPEG_LAME_ENC (h) LAME_ENC (i) MPEG_LAME

(j) MPEG_ENC

Figure 4.7: Compression Ratio for NICD (X-Match)

The average compression ratio observations for Figure 4.6 is shown in Table 4.1 for

window sizes 1 and 2 for all benchmarks. The cache data with BICD processing are compressed from 70 % to 98 %, with the maximum compression ratio being 97.21% for MILC for window size 1 and minimum 70.94% for HMMER for window size 2. The average compression for all benchmarks was 84%.

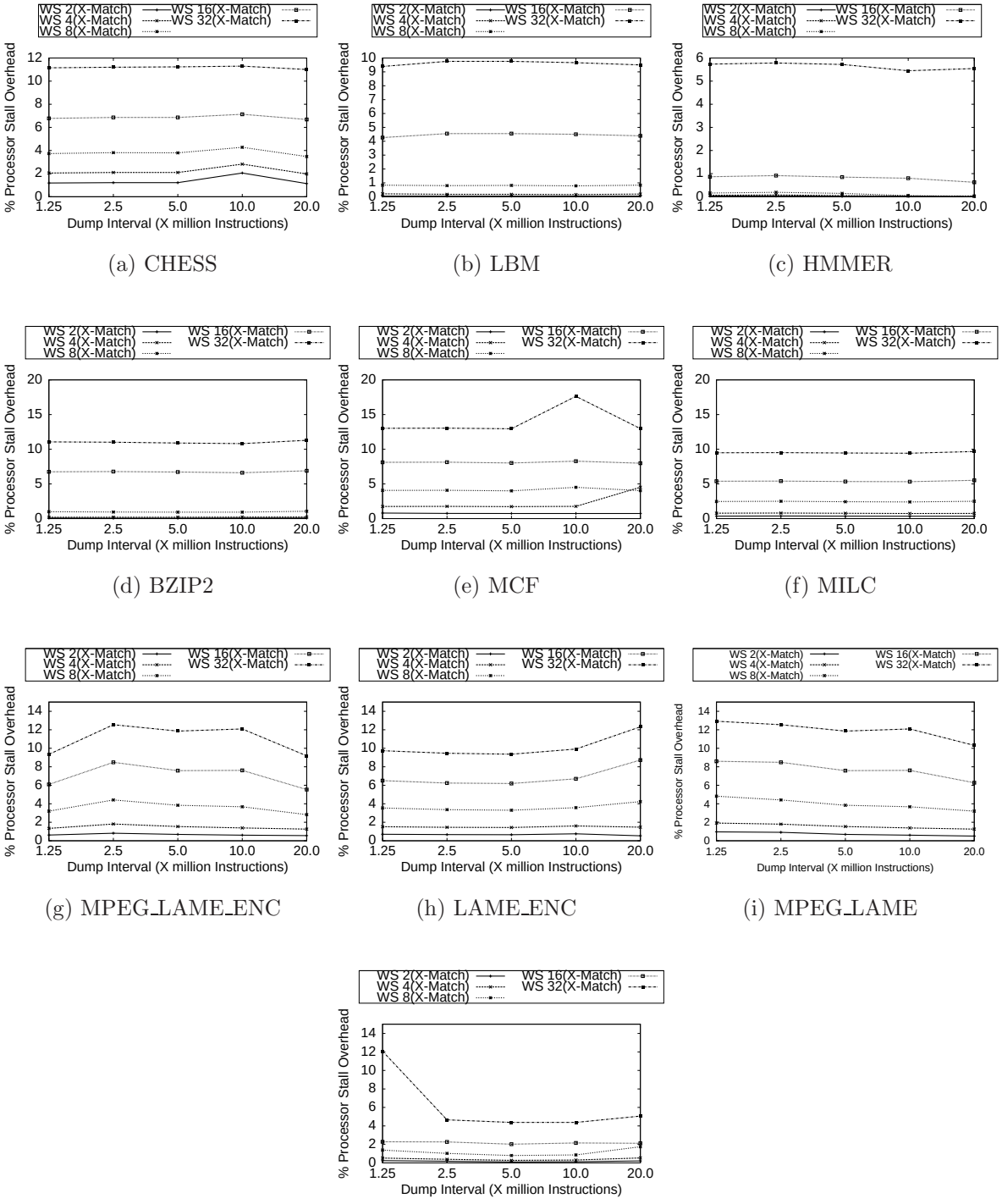
NICD Compression Results

For NICD methodology we have selected X-Match compressor, as the dump time depends on faster compression. Figure 4.7 shows similar trends to BICD compression but with lower compression ratio (except HMMER). The average compression ratio for window sizes 1 and 2 for all benchmarks is shown in Table 4.2. The data is compressed from 64% to 96% with maximum compression ratio 95.28% for MILC for window size 1 and minimum 64.96% for BZIP2 for window size 2. The average compression for all benchmarks was 82.5%.

We also carried out the experiments to capture the activity in the control bits and dumped the control bits according to the respective update information. We noticed a 0.23% average improvement in compression ratio.

Benchmark	Window Size 1	Window Size 2	Figure
CHESS	82.29%	79.20%	4.7(a)
LBM	78.71%	77.70%	4.7(b)
HMMER	91.02%	89.12%	4.7(c)
BZIP2	73.49%	64.96%	4.7(d)
MCF	95.28%	94.89%	4.7(e)
MILC	86.80%	84.29%	4.7(f)
MPEG_LAME_ENC	82.18%	81.34%	4.7(g)
LAME_ENC	82.21%	81.38%	4.7(h)
MPEG_LAME	78.71%	77.70%	4.7(i)
MPEG_ENC	95.28%	94.89%	4.7(j)

Table 4.2: Average Compression ratio (%) for NICD



(a) CHESS (b) LBM (c) HMMER

(d) BZIP2 (e) MCF (f) MILC

(g) MPEG_LAME_ENC (h) LAME_ENC (i) MPEG_LAME

(j) MPEG_ENC

Figure 4.8: Processor Stalls with NICD

4.4.4 Effect on Processor Execution (For NICD)

Dumping a window of cache lines in NICD affects processor execution. The processor is stalled if a cache update is requested during the dumping of a window of cache lines. Hence, the stall count is an important metric for the evaluation of cache line window size, as it also contributes towards the debug time for NICD.

Figure 4.8 shows the processor stalls, expressed as a percentage of actual dump time (6.12×10^5 CPU cycle), with respect to the various dump intervals as well as variation of cache line window sizes from 2 to 32 (in powers of 2). Processor stalls increase with increasing window size.

Cache update requests to the non-dumped lines initiate the process to dump the window of cache lines. This results in processor stalls for subsequent cache updates. Since X-Match compressor compresses the data at a faster rate, it sees fewer processor stalls. Since LBM updates the cache in a random manner, it results in fewer processor stalls (Figure 4.8(b)), with minimum average 0.05% for window size 2. Similar trends are observed for CHESS benchmark (Figure 4.8(a)) with minimum average processor stalls being 1.2% of actual dump time for window size 2.

In HMMER the memory requests are spread over time with infrequent updates. Hence during dumping it results in fewer processor stalls with a minimum average 0.01% for window size 2 (Figure 4.8(c)). HMMER also performs well for window size 4 with average 0.1%. BZIP2 and MILC exhibits similar trends for processor stalls with minimum average of 0.07% and 0.36%, respectively, for window size 2 (Figure 4.8(d) & 4.8(f)). MCF shows similar trends as BZIP2 except for window size 32 at dump interval of 10 million instructions and window size 4 at dump interval of 20 million instructions with minimum average 0.73% for window size 2 (Figure 4.8(e)).

Media Benchmarks are stream applications that update the cache mostly due to cache misses only. For window size 1, average processor stalls for MPEG_LAME_ENC is 0.81%, for LAME_ENC is 0.66%, for MPEG_LAME is 0.73% and for MPEG_ENC is 0.17%. Hence, it is clear that each benchmark has it own optimal window size.

4.4.5 Dump Time

For our targeted cache, without using any of our compression strategies it takes 6.12 × 10^5 CPU cycles to dump the contents.

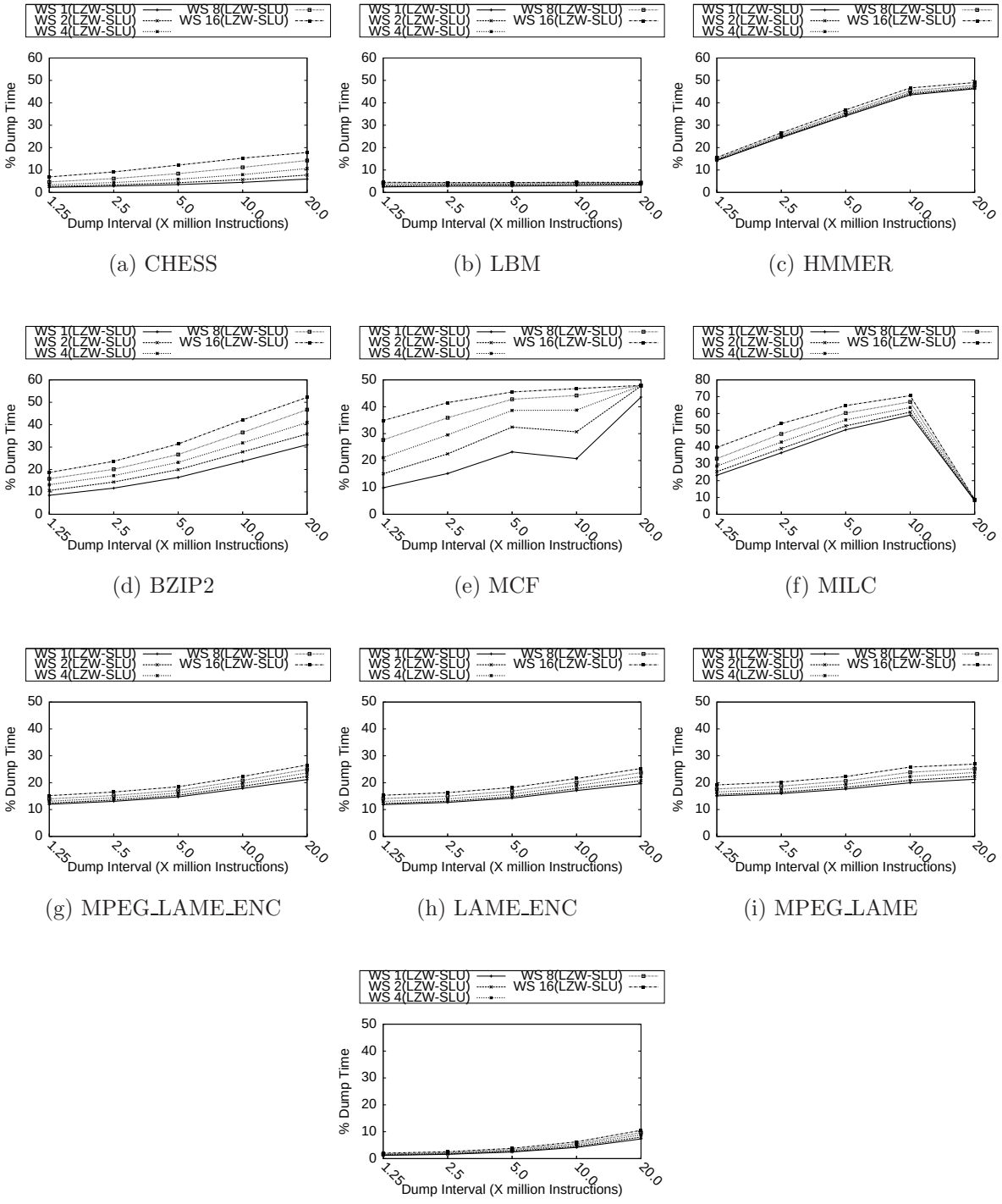
BICD Dump Time

The dump time from compression engine to logic analyzer for BICD depends on the maximum of the compression time and data transfer time. In Chapter 2 we show that parallel compressors can reduce the overall effect of compression and transfer time. For our experiments we use 16 parallel compressors. Figure 4.9 shows the transfer time for BICD for different cache dump intervals with various window sizes.

We observed for window size 1 and 2 of CHESS benchmark an average of 3.8% and 4.7%, respectively, of actual dump time across all dump intervals (Figure 4.8(a)). For LBM and HMMER, the minimum average dump times are 2.8% and 31.9% respectively (Figures 4.8(b) and 4.8(c)). For benchmarks BZIP2, MILC and MCF the minimum average dump times are 18.25%, 35.4% and 22.5% respectively for windows size 1 (Figures 4.8(d), 4.8(f) and 4.8(e)). For Media benchmarks MPEG_LAME_ENC, LAME_ENC, MPEG_LAME and MPEG_ENC the minimum average dump times are 15.76%, 15.09%, 17.98% and 3.34 respectively. The average transfer time for window sizes 1 and 2 across all benchmarks are 102.6 × 10^3 CPU cycles and 112.7 × 10^3 CPU cycles respectively which amounts to 16.8 % and 18.4 % respectively of actual dump time.

NICD Dump Time

The NICD mechanism has a dumping overhead, as it uses the system bus to transfer the data to the logic analyzer. The processor sees the system bus busy if it requests its use during data transfer between the compression hardware and the logic analyzer. The total dump time overhead for NICD is the sum of the overhead caused by the processor stalls and dumping overhead. Figure 4.10 shows the variation of the percentage of dump time for different dump intervals for different window sizes (WS, from 1 to 16 in power of 2).



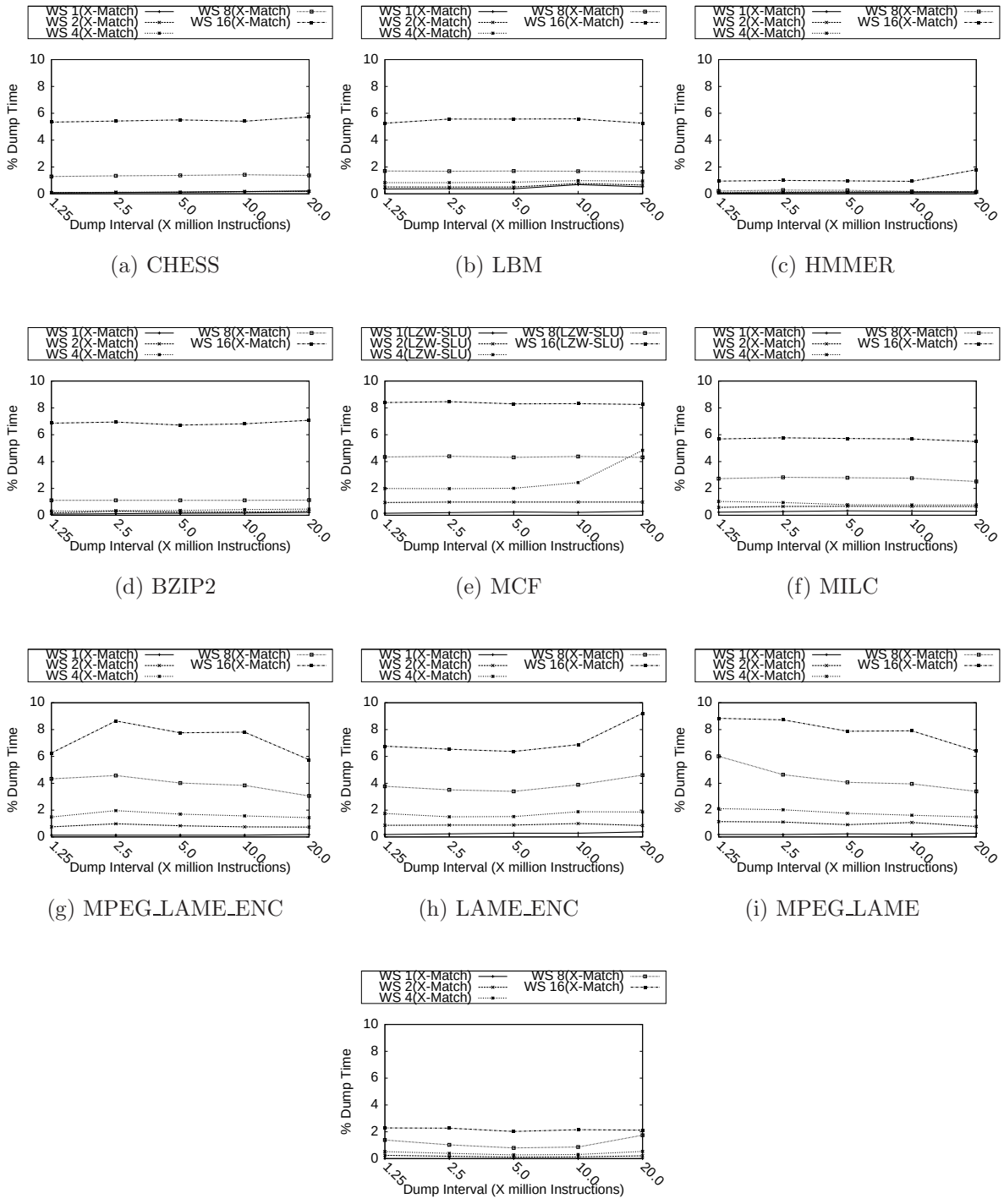
(a) CHESS (b) LBM (c) HMMER

(d) BZIP2 (e) MCF (f) MILC

(g) MPEG_LAME_ENC (h) LAME_ENC (i) MPEG_LAME

(j) MPEG_ENC

Figure 4.9: % of Dump Time Overhead using Parallel Compressors for BICD



(a) CHESS (b) LBM (c) HMMER

(d) BZIP2 (e) MCF (f) MILC

(g) MPEG_LAME_ENC (h) LAME_ENC (i) MPEG_LAME

(j) MPEG_ENC

Figure 4.10: % of Dump Time Overhead using Parallel Compressors for NICD

Since the dumping overhead is small in comparison to the overhead caused by the processor stalls, we noticed that for all window sizes, the overall dump time follows the trends of processor stalls. We observed 0.11% dump time which was almost identical for all dump intervals for the window size 1 of CHESS benchmark (Figure 4.10(a)). For LBM and HMMER (Figures 4.10(b) and 4.10(c)), the minimum dump time is 0.0022 % and 0.009% respectively. For MCF, MILC, BZIP2 benchmarks the average dump times for window size 1 are 0.016%, 0.032% and 0.0014% respectively (Figures 4.10(e), 4.10(f) and 4.10(d)). For Media benchmarks MPEG_LAME_ENC, LAME_ENC, MPEG_LAME and MPEG_ENC average dump times are 0.0014%, 0.0026%, 0.0012% and (approximate) 0% respectively for window size 1. The average transfer time for window sizes 1 and 2 across all benchmarks are 1.2×10^2 CPU cycles and 4.4×10^2 CPU cycles respectively which amounts to 0.0019% and 0.0071% respectively of actual dump time.

4.4.6 Implementation and Synthesis Results

We synthesized VHDL modules with 180 nm library using Synopsys Design Compiler and estimated the memory component parameters using CACTI tool. For our targeted cache the CACTI estimation in 180 nm are: 2.63 ns access time and 38.9 mm^2 of area.

For BICD mechanism we require a UHT and few MUX'es and registers. CACTI estimation for the UHT is shown in Table 4.3. The first column of the table shows the window size. The second column shows the UHT size requirements for the target cache and the window size. The third column shows the estimation of silicon area requirement for the UHT and the last column shows the access time estimation for the UHT. For all required UHT sizes, the critical path is smaller than the cache access time. Therefore, UHT does not cause any time overhead. The other components have negligible area overhead.

The NICD mechanism requires two UHT tables and an additional MUX. Therefore area requirements for NICD are twice that of BICD, with no extra time overhead, as both UHTs work in parallel. NICD also modifies the cache controller to implement the online dumping mechanism. We synthesised both controllers, with and without online dumping, in 180nm technology and found the area difference to be 0.0002 mm^2 with no

time overhead.

Taking the silicon area and dump time into account from Figures 4.10, 4.9 and Table 4.3, we can conclude that window size 4 results in an acceptable trade-off between dump time and UHT area.

Table 2.5 shows the area, critical path delay and bytes processed per cycle by the compressor. The comparison shows X-Match to be superior in area and delay (its latency is longer, but it processes 4 bytes per cycle). However its compression ratio is worse.

Window Size	UHT Size (Byte)	Silicon Size (mm^2)	Critical Path (ns)
1	4096	0.24	1.5
2	2048	0.13	1.2
4	1024	0.08	1.1
8	512	0.03	1.0
16	256	0.03	1.0

Table 4.3: UHT Size, Area and Critical Path

4.4.7 Limitations

BICD has no known limitations, while NICD has the following limitation. NICD places an lower bound on the dump interval. The dump interval cannot be less than the cache dump duration using NICD. If a smaller duration is needed, we have to BICD strategy for cache dumping.

4.5 Conclusion

Post-silicon debug is an important phase of the processor design cycle. In this phase bugs are identified through analysis of the internal state of the processor. In this work, we have proposed and evaluated incremental cache state dumping for reduction of transfer time and logic analyzer space requirement. To perform incremental cache dumping, we have proposed two methodologies based on processor execution state: Blocking Incremental Cache dumping (BICD) and Non-blocking Incremental Cache Dumping (NICD). Incremental cache dumping reduces the cache lines dumped by 64%. With BICD and

NICD we are able to reduce the processor halt time/dump time to 16.2% and 0.0019%, respectively, of the actual cache dump time. Further compression of the dumped cache line saved 84% of the logic analyzer space.

Chapter 5

Conclusion and Future Work

This thesis addresses an important issue in the context of Post-silicon processor debug. In Post-silicon debug process, we capture the internal state of the processor through a logic analyzer for offline analysis. The internal state consists of all memory elements, the bulk of which is contributed by on chip L2 cache (L2 + L3, if present on chip) and requires lot of expensive logic analyzer memory. Hence, the problem of transferring the processor state is essentially the problem of transferring the cache state to the logic analyzer. In this thesis, we have proposed hardware techniques to reduce the cache content transfer time and amount of cache data.

5.1 Main Contributions

We have proposed the following hardware mechanisms for reducing the cache content transfer time and amount of cache data.

- *Cache Aware Compression*: We have proposed to capture the cache state through a hardware compression engine for reducing the cache state dump time and reducing the amount of data transfer. The hardware compression engine uses our proposed technique of *Cache Aware Compression* which exploits the cache structure efficiently for better compression. We have selected LZW compression algorithm for cache compression and proposed the following two implementations for better results.

  - *LZW-SLU*: Hardware implementation of LZW algorithm requires dictionary pruning. We have proposed a simple two bit LRU approximation **S**ingle bit **L**RU with **U**pdate bit (SLU), for LZW dictionary management in hardware.

  - *LZW-DC*: LZW algorithm works on temporal locality found in the input stream. We have proposed a dual dictionary structure in the LZW algorithm implementation. The proposed approach, LZW with Dictionary Caching (LZW-DC), effectively exploits the temporal locality found in the input stream.

    Our hardware implementation of LZW algorithm compresses data upto 16.5% more than other known hardware compressors.

  Our proposed methodology gives 7-31% more compression than a compression that is unaware of the cache structure. We have also proposed the parallel version of the compression engine which reduces the transfer time to 54%.

- *Online Cache Dumping*: We have proposed *Online Cache Dumping* which minimizes the effective debug time by capturing the cache content with processor execution overlap. To prevent the cache state from getting overwritten by processor execution, we have developed two techniques that dump the cache lines before the regular cache update proceeds. Experimental evaluation shows that this approach reduces the effective L2 cache dump time to between 0.01% and 3.5% of the original duration.

- *Incremental Cache Dumping*: We have proposed *Incremental Cache Dumping* techniques targeting the reduction of the number of cache line dumps. It utilizes the fact that between two consecutive dumps, relatively few L2 cache lines are updated by the processor execution. It dumps only the recently updated cache lines by tracking their update using an Update History Table. Based on the processor execution state, we have proposed two hardware implementation alternatives. Our results show 64% reduction in the number of cache lines dumped.

5.2 Future Directions

In this thesis, we presented methodologies for efficient capture of the cache contents targeting post-silicon processor debug. Future research can develop the following topics further.

- The dump interval, at which the processor snapshot taken, is critical in post-silicon validation. As the post-silicon debug is carried out in a loop, the dump interval determines the debug cycle time. Further analysis is required to arrive at the best dump interval.

- In our work, we have considered a debug platform without asynchronous peripherals and interrupt routines. This work can be extended to capture the cache contents in such an environment.

- Validation of on-chip multiprocessors is a challenge with multiple cores executing simultaneously. This work can be extended to address the validation process of multiprocessors with shared/private caches. For shared caches, this work can also be extended to validate the issues related to the cache coherency protocol.

- The techniques proposed in this thesis could also be used to support data migration in a multiprocessor environment based on dynamic factors such as temperature and data locality.

- In general, design-for-debug features are intended to aid in exposing internal chip contents. The co-design of such features along with security mechanism could be an interesting area of investigation.

Bibliography

[1] Dimitris Gizopoulos. *Advances in Electronic Testing: Challenges and Methodolgies*. Springer, 2006.

[2] http://en.wikipedia/wiki/Moore's_law.

[3] http://www.intel.com/technology/mooreslaw/index.htm.

[4] http://www.intel.com/pressroom/kits/quickrefyr.htm#2008.

[5] B. Roberts. The verities of verification. *Electronic Business*, 2003.

[6] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, Los Alamitos, CA, USA, 2003.

[7] Keith Baker and Jos van Beers. Shmoo Plotting: The Black Art of IC Testing. *IEEE Design & Test of Computers*, 14(3):90–97, 1997.

[8] Bart Vermeulen, Mohammad Z. Urfianto, and Sandeep K. Goel. Automatic Generation of Breakpoint Hardware for Silicon Debug. In *Proceedings of the 41th Design Automation Conference, DAC 2004*, pages 514–517, San Diego, CA, USA, June 7-11, 2004.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture*. Morgan Kaufmann, Third edition, 2003.

[10] Anant Vishnoi, Preeti Ranjan Panda, and M. Balakrishnan. Online Cache State Dumping for Processor Debug. In *Proceedings of the 46th Design Automation Conference, DAC'09*, pages 358–363, San Francisco, CA, USA, July, 26-31, 2009.

[11] Kedarnath J. Balakrishnan, Nur A. Touba, and Srinivas Patil. Compressing Functional Tests for Microprocessors. In *Proceedings of the 14th Asian Test Symposium on Asian Test Symposium, ATS 2005*, pages 428–433, Calcutta, India, December 18-21, 2005.

[12] Ehab Anis and N. Nicolici. Low Cost Debug Architecture using Lossy Compression for Silicon Debug. In *Proceedings of the conference on Design, Automation and Test in Europe Conference and Exposition,DATE 2007*, pages 225–230, Nice, France, April 16-20, 2007.

[13] Sung-Boem Park and Subhasish Mitra. IFRA: instruction footprint recording and analysis for post-silicon bug localization in processors. In *Proceedings of the 45th Design Automation Conference, DAC 2008*, pages 373–378, Anaheim, CA, USA, June 8-13 2008.

[14] Hao Fang, Chenguang Tong, Bo Yao, Xiaodi Song, and Xu Cheng. CacheCompress: a novel approach for test data compression with cache for IP embedded cores. In *Proceedings of the International Conference on Computer-Aided Design,ICCAD 2007*, pages 509–512, San Jose, CA, USA, November 5-8, 2007.

[15] Morten Kjelso, Mark Gooch, and Simon Jones. Design and Performance of a Main Memory Hardware Data Compressor. In *Proceedings of the 22nd EUROMICRO Conference*, pages 423–430, Prague,Czech Republic, September 2-5, 1996.

[16] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research and Development*, 45(2):271–285, 2001.

[17] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[18] Terry A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17:8–19, June 1984.

[19] Ming-Bo Lin. A Hardware Architecture for the LZW Compression and Decompression Algorithms Based on Parallel Dictionaries. *J. VLSI Signal Process. Syst.*, 26(3):369–381, 2000.

[20] Su. Chauchin, Yen Chenq-Fan, Yo Jang-Chaung. Hardware efficient updating technique for LZW CODEC design. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, pages 2797–2800, Hong Kong, June 1997.

[21] Kun-Jin Lin and Cheng-Wen Wu. A Low-Power CAM Design for LZ Data Compression. *IEEE Trans. Comput.*, 49(10):1139–1145, 2000.

[22] Rupak Samanta and Rabi. N. Mahapatra. An enhanced CAM architecture to accelerate LZW compression algorithm. In *20th International Conference on VLSI Design (VLSI Design 2007) jointly held with Sixth International Conference on Embedded Systems (ICES 2007)*, pages 824–829, Bangalore, India, January 6-10, 2007.

[23] Luca Benini, avide Bruni, Bruno Ricco, Alberto Macii, and Enrico Macii. An adaptive data compression scheme for memory traffic minimization in processor-based systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS-02*, pages 866–869, Arizona, USA, May 26-29, 2002.

[24] Youtao Zhang and Rajiv Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 14–28, Grenoble, France, April 8-10 2002.

[25] Haris Lekatsas and Wayne Wolf. Code Compression for Embedded Systems. In *Proceedings of the 35th Design Automation Conference, DAC 1998*, pages 516–521, San Francisco,CA,USA, June 15-19 1998.

[26] H. Lekatsas and Wayne Wolf. SAMC: A Code Compression Algorithm for Embedded Processors. *IEEE Transactions on CAD*, 18(12):1689–1701, 1999.

[27] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. CRAMES: compressed RAM for embedded systems. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2005*, pages 93–98, Jersey City, NJ, USA, September 19-21, 2005.

[28] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture, ISCA 2004*, pages 212–223, Munich, Germany, June 19-23, 2004.

[29] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *Proceedings of Internationl Conference on Computer Design (ICCD 99)*, pages 184–191, Austin,Texas, October 10-13, 1999.

[30] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. A Decompression Architecture for Low Power Embedded Systems. In *Proceedings of Internationl Conference on Computer Design (ICCD 00)"*, pages 571–574, Austin, Texas, USA, September 17-20, 2000.

[31] Alaa R. Alameldeen and David A. Wood. Frequent pattern compression:A significance-based compression scheme for L2 caches. Techincal report, University of Wisconsin, Madison, April 2004.

[32] David Solomon. *Data Compression: The Complete Reference*. Springer, third edition, 2004.

[33] Dhiraj Jairam Shetty. An implementation of compression hardware to assist processor debug. Mtech. Thesis, Dept. Of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, 2006.

[34] Golomb Solomon W. Run-Length Encodings. *IEEE Transacations on Information Theory*, IT(12):399–401, 1966.

[35] Jon Louis Bentley and Daniel Dominic Sleator and Robert Endre Tarjan and Victor K. Wei. A Locally Adaptive Data Compression Scheme. *Commun. ACM*, 29(4):320–330, 1986.

[36] V.S Miller and W.N. Wegman. Variation On a Theme By Ziv and Lempel. *Combinatorial Algorithms on Words*, F12:131–140, 1985.

[37] W. Fung. Low Power circuits for multiple match resolution and detecting for Ternary CAM. MASc. Thesis, Dept. of Electrical and Computer Engineering, University of Waterloo, ON, Canada, 2004.

[38] D. Burger and T. Austin. The simplescalar tool set, version 2.0 technical report cs-tr-97-1342, June 1997.

[39] J. Edler and M. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/ markhill/DineroIV/.

[40] Canterbury corpus compression benchmarks. http://corpus.canterbury.ac.nz/.

[41] http://compression.ca/act/act-files.html.

[42] Hassan Ghasemzadeh, Sepideh Sepideh Mazrouee, and Mohammad Reza Kakoee. Modified pseudo LRU replacement algorithm. In *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006)*, pages 368 – 376, Potsdam, Germany, March 27-30, 2006.

[43] N.Jouppi. http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[44] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *J. Instruction-Level Parallelism*, 2, 2000.

Technical Biography of Author

Anant Vishnoi did his B.Tech. (Information Technology) from V.B.S Purvanchal University, Jaunpur in June 2002. He, then, registered for Ph.D. at IIT Delhi in August 2003. Currently he is working as Research Associate with ISL, GM, Bengaluru, India. His research interest includes Post-silicon Validation, High-Performance Digital Design and Embedded Systems.

List of Publications

1. Anant Vishnoi, Preeti Ranjan Panda and M. Balakrishnan, "Cache Aware Compression for Processor Debug", IEEE/ACM SIGDA Design Automation and Test in Europe (DATE), April 2009, Nice, France.

2. Anant Vishnoi, Preeti Ranjan Panda and M. Balakrishnan, "Online Compression for Processor Debug", IEEE/ACM SIGDA Design Automation Conference (DAC), July 2009, San Franscisco, California, USA.

3. Preeti Ranjan Panda, M. Balakrishnan and Anant Vishnoi, "Compressing Cache State for Post-Silicon Processor Debug", IEEE Transcations on Computers, 2011.

4. Preeti Ranjan Panda, Anant Vishnoi and M. Balakrishnan, "Enhancing Post-silicon Processor Debug with Incremental Cache Dumping", VLSI-SOC, 2010, September 2010, Madrid, Spain.