Space Sensitive Cache Dumping for Post-silicon Validation

Sandeep Chandran, Smruti R. Sarangi, and Preeti Ranjan Panda
Department of Computer Science and Engineering, Indian Institute of Technology Delhi
Hauz Khas, New Delhi – 110016
{sandeep, srsarangi, panda}@cse.iitd.ac.in

***Abstract*—The internal state of complex modern processors often needs to be dumped out frequently during post-silicon validation. Since the last level cache (considered L2 in this paper) holds most of the state, the volume of data dumped and the transfer time are dominated by the L2 cache. The limited bandwidth to transfer data off-chip coupled with the large size of L2 cache results in stalling the processor for long durations when dumping the cache contents off-chip. To alleviate this, we propose to transfer only those cache lines that were updated since the previous dump. Since maintaining a bit-vector with a separate bit to track the status of individual cache lines is expensive, we propose 2 methods: (i) where a bit tracks multiple cache lines and (ii) an *Interval Table* which stores only the starting and ending addresses of continuous runs of updated cache lines. Both methods require significantly lesser space compared to a bit-vector, and allow the designer to choose the amount of space to allocate for this design-for-debug (DFD) feature. The impact of reducing storage space is that some non-updated cache lines are dumped too. We attempt to minimize such overheads. Further, the Interval Table is independent of the cache size which makes it ideal for large caches. Through experimentation, we also determine the break-even point below which a t-lines/bit bit-vector is beneficial compared to an Interval Table.**

I. Introduction

The increasing complexity of modern processors has resulted in the evolution of sophisticated post-silicon validation features because simulation techniques are inadequate for executing large test case scenarios [14], [15]. During post-silicon validation, sample chips are validated on test platforms where the execution is performed at high speed, but elaborate mechanisms need to be put in place to identify the cause of errors once faults are identified. Recent research efforts in this area have investigated the broad topics of bug localization and diagnosis [16], [17], [24], trace signal selection [18], [19], and adaptation of test compression techniques [20], [21].

Among several interesting design-for-debug (DFD) features used in the industry today, is a dumping mechanism, where the entire state of the processor is transferred off-chip to the debug infrastructure, to be analyzed offline. However, such DFD hardware needs to operate under tight area constraints, and should cause minimal interference in the normal execution of the processor. Thus, the goals of an ideal DFD hardware supporting efficient dumping during post-silicon validation, are: (i) minimal intrusiveness; (ii) minimal space requirements; and (iii) maximum visibility into the chip. These requirements are clearly orthogonal, and balancing the three is a complex task.

978-3-9815370-0-0/DATE13/©2013 EDAA

The entire processor state consists of L1/L2 caches, and registers in various structures such as pipelines, register files, TLBs, and reorder buffers. Capturing this state at regular intervals and analyzing sequences of such snapshots offline gives crucial hints on possible causes of errors. However, since the L2 cache is large, transferring each snapshot off-chip is a time consuming process. We also require that, during the dumping phase, the processor should not update the cache since it may lead to inconsistencies. Therefore, the processor is stalled during the dumping phase. The duration of processor stalls can be reduced by decreasing the amount of data that is required to be transferred off-chip [4], [20].

Another way to reduce the amount of data to be transferred off-chip is by dumping only the cache lines that were updated after the previous snapshot. The straightforward method of storing the information on the lines that were updated in the current dumping cycle is to maintain a bit-vector where each bit represents a cache line and is set to 1 if the line is updated. Its disadvantage is that the amount of space required could be unacceptably large for the large caches of modern processors since the bit-vector size equals the number of cache lines. In this work we attempt to store the information about updated cache lines in less space than that required by a bit-vector representation, and allow the designer to control the area overhead. The reduction in space utilized to capture the information on updated cache lines in our proposed structure, may lead to a small increase in the number of lines transferred off-chip, which in turn, results in slightly longer durations of the dumping phase as compared to that of bit-vector.

II. Related Work

The challenge of limited visibility posed during post-silicon validation is generally handled by two broad approaches: tracing signals and event triggers. Most of the issues that arise when tracing signals are due to limited storage space available on-chip to store the signal values. This is improved by getting rid of redundant signals and intelligently choosing the signals for tracing. This improves the usage of trace buffers, as they can now store values of many more signals than previously possible. Novel methods have been proposed to solve issues related to tracing signals for debugging purposes [1], [2], [3], [5]. On the other hand, event triggers have a different set of issues, including defining the triggers correctly, storing the definitions of triggers, and routing signals to and from the on-chip control units that decide whether or not an event has occurred. Some novel solutions to these problems have been proposed recently by researchers [6], [7], [8], [9]. Recently, transaction based debugging methods such as [10] and [11] have been

proposed. Other methods have been proposed for inferring the subset of signals responsible for the faulty behaviour by observing the manifestations of these signals on other easily observable parameters [12], [13], [22], [23]. Some research works also target compression [25], [26] and analysis [27] of traces collected from executions, which are then used for simulations or bug localization. Our method is orthogonal to, and complements the above mentioned approaches in that it is targeted at capturing a relatively complete snapshot of the chip state. The other approaches are still applicable in this context, during the processor execution cycle between successive state dumps.

The field of data clustering, where n data points are grouped into k-clusters while minimizing a distance metric, is conceptually close to our area of work. We explored this method by adapting a well-established clustering algorithm BIRCH [28] to suit our scenario. In spite of the conceptual similarities, a direct application of this method to hardware will be inefficient because (i) operations such as *split* and *rebuild* (which are central to BIRCH) are very time consuming (ii) the complex algorithm would increase the area overhead, defeating the main objective of our work. Our preliminary simulations showed that BIRCH gave inferior results compared to the proposals in this paper. Hence we did not synthesize this algorithm.

Our work is similar in scope to [4] and [29], where the cache contents are compressed using well-established compression algorithms in order to reduce the amount of data transferred off-chip. Our proposal focuses on two important differences from the above works: (i) using very little space, track and dump only the updated cache lines; and (ii) allow the designer to parameterize this DFD feature based on an area constraint. The compression techniques proposed above can be applied on the cache lines that are marked as updated by our technique to further reduce the size the data transferred off-chip; this further strengthens the state dump-driven debugging philosophy that these approaches follow.

III. Methodology

A. Definitions

Definition 1: A ***bit-vector*** corresponds to a sequence of 0s and 1s, where 1 indicates that the corresponding cache line was modified after the previous cache dump and 0 indicates otherwise.

Definition 2: We define the ***Overhead*** as the number of non-updated cache lines that are transferred off-chip, expressed as a percentage of the total number of cache lines.

The high spatial locality of reference in caches is the primary motivation to use lesser space than a bit-vector to track updated cache lines. Our proposals exploit this property to save storage space.

B. ***Method 1*** : *t-lines/bit bit-vector*

This method maps each bit to t (> 1) adjacent cache lines. A bit is set to 1 if any of the t cache lines to which it corresponds is updated after the previous cache dump. This method reduces the amount of storage required by $(\frac{1}{t})$. A designer has the flexibility to choose any value of t that satisfies the area budget. A large value of t increases the *overhead* due to an increase in the number of non-updated cache lines in the proximity of an updated cache line. In Figure 1, columns (ii), (iv) and (vi) illustrate the t-lines/bit bit-vector for $t = 2$ corresponding to the bit-vectors shown in columns (i), (iii) and (v) respectively. The respective overheads incurred are shown in the bottom row.

C. Interval Table

Definition 3 : A pair of starting address and ending address (start_addr, end_addr) defines an ***Interval***. A set of k intervals is stored in an ***Interval Table I[k]***.

Unlike the previous method, this method does not partition the cache lines into fixed size sets. Instead, a continuous run of 1s in the bit-vector is stored as an Interval. The amount of storage required to store an Interval is constant irrespective of the length of the run of 1s. The shaded portion in columns (i), (iii) and (v) of Figure 1 shows the set of cache lines marked as updated by an Interval Table with $k = 1$.

	(i)	(ii)	(iii)	(iv)	(v)	(vi)
Updated :	11	11	9	11	8	8
Dumped :	11	12	12	12	15	16
Overhead :	0.0	6.25	18.8	6.25	43.8	50

Fig. 1. Example of a Bit-vector with corresponding 2-lines/bit bit-vector. The shaded portion of bit-vector shows the dumped lines using Interval Table with $k = 1$

(a)

St. Ad	Length
2	1
4	2
9	1
11	4

(b)

St. Ad	Length
11	4
4	2
2	1
9	1

(c) - Bit vector (ii)

St. Ad	End Ad
0	10
15	15

(d) - Bit Vector (iii)

St. Ad	End Ad
0	3
6	10
15	15

Fig. 2. Illustration of the optimal offline algorithm (a) Initial Gap Table (b) Sorted Gap Table (c) Optimal Interval Table for $k = 2$ (d) Optimal Interval Table for $k = 3$

There are two adversarial situations to the Interval-based approach in practice: (i) in a bit-vector, the actual number of runs of 1s can be very large, and (ii) the length of some runs can be smaller than the number of bits required to store its starting and ending addresses. To overcome the former issue, we merge some adjacent intervals to reduce the number of stored intervals to k. The effect of the latter is sought to be overcome by the space saved when there are long runs of 1s. Since $log_2 N$ bits are needed to address N cache lines, storing k intervals requires $2klog_2 N$ bits, and it is essential that $2klog_2 N < N$ (or $k < \frac{N}{2log_2 N}$) for the Interval Table to save space over the bit-vector. Therefore, the main challenge now is to capture the information (starting and ending addresses) of $\boldsymbol{n}$ runs of 1s in just k intervals where $n \gg k$.

Merging adjacent intervals due to the upper bound on k, will capture some non-updated cache lines into the intervals, which leads to overheads. Thus, the intervals with the smallest runs of non-updated cache lines between them should be merged. However, determining the nearest intervals is non-trivial because of the *online* nature of the problem. Interval Table $I[k]$ needs to be maintained as and when the cache lines

are being updated. Intervals that were far apart at some point in history can become the nearest if subsequent updates by the processor occur to the cache lines between the two intervals. Such problems are common in practical online algorithms, such as page replacement.

D. Optimal Offline Algorithm

Definition 4: A set of k intervals ***O[k]*** is said to be ***Optimal*** if no other set of k intervals exists with a lower overhead.

The optimal algorithm takes the number of intervals k that can be stored and the set of all updated cache lines as inputs and returns the optimal set of k intervals for these inputs. It is an *offline* algorithm, which requires the entire set of updated lines as input. The solution returned by the optimal algorithm helps determine the theoretical lower limit of the overhead when using the Interval Table method to maintain the information on updated cache lines and can be used as a benchmark to compare the performance of the other online algorithms.

Definition 5: A continuous run of non-updated cache lines is called a ***Gap*** and a ***Gap Table*** holds the starting address and the length of the current set of *Gaps*.

Algorithm 1 Optimal offline algorithm

Input: Bit vector $B < 0..n-1 >$, Number of Intervals k
Output: Optimal set of k intervals $O[k]$

1: Store (start_addr, length) of all gaps in B, in Gap Table Gap[]
2: Sort the Gap Table in descending order of the gap lengths
3: Select topmost $(k-1)$ entries and sort them in ascending order of start_addr
4: $O[0]$.start_addr $\leftarrow$ Address with first 1 in B
5: **for** $i = 0$ **to** $k-1$ **do**
6: $\quad O[i]$.end_addr $\leftarrow$ Gap[i].start_addr -1
7: $\quad O[i+1]$.start_addr $\leftarrow$ Gap[i].start_addr + Gap[i].length
8: **end for**
9: $O[i+1]$.end_addr $\leftarrow$ Address with last 1 in B

Algorithm 1 summarizes the offline strategy. Line 1 constructs a Gap Table by iterating over the bit-vector B and storing the starting address and length of the continuous runs of 0s. This is illustrated in Figure 2(a) for the initial bit-vector marked (i). After constructing the Gap Table, Line 2 sorts it in the descending order of the gap lengths (Figure 2(b)). The top $k-1$ entries are relevant to us. We first sort them in increasing order of the start addresses (Line 3). Lines 5-8 in the algorithm construct the final set of k intervals ($O[k]$) by simply eliminating the cache lines constituting the topmost $k-1$ gaps in the sorted Gap Table. Figure 2(c) and 2(d) indicate the optimal set of intervals ($O[k]$) for $k = 2$ and $k = 3$ respectively. The corresponding bit-vectors are shown in Figure 2(ii) and 2(iii) with the shaded regions indicating the stored intervals. Some special cases are omitted in the algorithm for the sake of simplicity of presentation.

E. *Method 2* : Greedy algorithm

We propose a greedy online algorithm to capture the information on updated cache lines into Interval Table $I[k]$. When the number of runs of 1s exceeds k, we merge adjacent

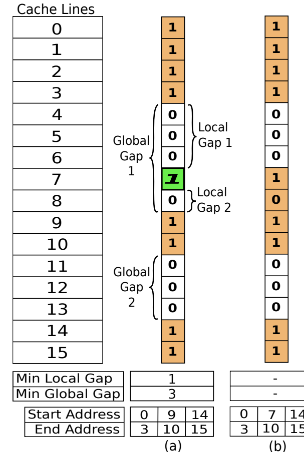

Fig. 3. (a) Illustration of local gaps and global gaps (b) Final bit-vector on accommodating new incoming request

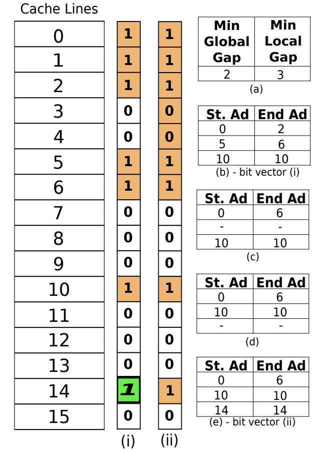

Fig. 4. Illustration of greedy algorithm when *minGlobalGap* is smaller than *minLocalGap*

intervals to form a larger interval. Since this action causes us to include some 0s in the interval, we select for merging two adjacent intervals with minimum Gap between them. Of course, some of the included 0s may get updated to 1 in the future due to spatial locality of accesses.

Algorithm 2 Greedy algorithm (Online)

Input: cache line *lineNum*, Number of Intervals k
Output: $I[k]$ at time t

1: minLocalGap $\leftarrow N$
2: minGlobalGap $\leftarrow N$
3: **for** $i = 0$ **to** $k-1$ **do**
4: $\quad$ **if** lineNum is within interval $I[i]$ **then**
5: $\quad\quad$ **return**
6: $\quad$ **else**
7: $\quad\quad$ minLocalGap $\leftarrow$ min((lineNum$-I[i]$.end_addr), ($I[i+1]$.start_addr $-$ lineNum), (minLocalGap))
8: $\quad\quad$ minGlobalGap $\leftarrow$ min((minGlobalGap), ($I[i+1]$.start_addr $-I[i]$.end_addr))
9: $\quad$ **end if**
10: **end for**
11: merge intervals around min(minGlobalGap, minLocalGap)
12: place the lineNum in suitable interval (singleton if required)

Definition 6: ***Local Gap*** is the distance of a newly updated cache line to a boundary of its nearest runs of 1s. ***minLocalGap*** is the minimum among the two local gaps (one to each boundary). ***Global Gap*** refers to the distance between adjacent intervals already stored in the Interval Table $I[k]$. Similarly, ***minGlobalGap*** is the minimum among the $k-1$ global gaps.

The greedy strategy is outlined in Algorithm 2. The **for**-loop iterates over all the stored intervals, and checks for membership of the newly updated cache line *lineNum* in the corresponding interval. If *lineNum* is already part of a stored interval, the algorithm returns without modifying anything. Otherwise, *lineNum* lies in a gap between two intervals. *minLocalGap* captures the smallest distance of *lineNum* to its nearest interval. Similarly, *minGlobalGap* stores the minimum distance between any two previously stored adjacent

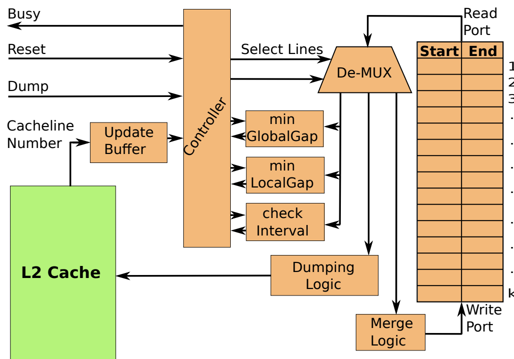

Fig. 5. Hardware design of Greedy algorithm

intervals. On exiting the **for**-loop, the algorithm would have determined the least possible distance when accommodating *lineNum*. In case *minLocalGap* is smaller, the new cache line *lineNum* will be merged to its nearest adjacent interval. This requires modification only to the nearest stored interval. In case *minGlobalGap* is smaller, then the intervals around the *minGlobalGap* will be merged to create space to store *lineNum* in an interval of its own.

Figure 3(a) shows the *local gaps* and *global gaps* for a newly updated cache line (*lineNum* = 7) with an Interval Table of size 3. Here, *minLocalGap* (for gap <8,8>) is smaller than *minGlobalGap* (for gap <11,13>) and hence only the nearest stored interval (<9,10>) is extended to accommodate 7. Figure 3(b) shows the final state of the Interval Table $I[k]$ and the corresponding bit-vector.

Figure 4 illustrates an example where *minGlobalGap* is smaller than *minLocalGap*. In this case, the intervals around the *minGlobalGap* (<0,2> and <5,6>) are merged (as shown in Figure 4(c)) so as to create space for *lineNum* (14 in this example). Once space is created, *lineNum* is stored in a new interval by itself (<14,14>).

The advantages of the Greedy algorithm are: (i) the storage space required is independent of the number of cache lines unlike bit-vector (ii) the granularity of addressable units can also be increased to t-lines instead of 1 line – larger t values causes a decrease in the number of bits stored in the Interval Table. The proof that the overhead is within 2x of that reported by the optimal algorithm is outlined in [30].

IV. Hardware Design

In this section, we discuss our design choices when implementing Greedy algorithm in hardware. The hardware implementation of bit-vector is straightforward and is not discussed.

A. Sorted Storage of Intervals

We decided to store the intervals in the Interval Table $I[k]$ in sorted order (based on their starting address). This simplifies the computation of *minLocalGap* and *minGlobalGap* in Algorithm 2. Had we stored intervals in unsorted order, finding the nearest two intervals (*minGlobalGap*) would require, for every interval, iteration over all the other stored intervals ($O(k^2)$ time). Now, in the case when *minGlobalGap* is smaller than *minLocalGap*, we have to shift some stored intervals, through a sequence of swaps to maintain the sorted order of the table. Figure 4(b),(c),(d) and (e) illustrate the sequence of operations when *minGlobalGap* is smaller than *minLocalGap*. The movement operation requires $O(k)$ time.

B. Logic Design

Figure 5 shows the detailed design of the hardware implementation of the Greedy algorithm. The hardware uses a single dual-ported memory to store all the k intervals. After each interval is read out in sequence, the check for membership, computing *minLocalGap* and *minGlobalGap* is performed simultaneously. If the newly updated cache line is determined to be a member of an existing interval, the *Controller* aborts all the in-flight operations and returns to the initial state. Otherwise, the *Local Gaps* and *Global Gap* are computed for the interval and checked against the current *minLocalGap* and *minGlobalGap*. If either the *Local Gap* or *Global Gap* is smaller than *minLocalGap* or *minGlobalGap* respectively, the new values are stored, along with the index of the interval. The *Controller* then decides the intervals that are to be merged based on the values of *minLocalGap* and *minGlobalGap*. In subsequent cycles, the suitable intervals are read out, merged and written back at suitable locations to the Interval Table $I[k]$.

Reading and computing *minLocalGap* and *minGlobalGap* requires k cycles. The merging operation would take 2 cycles. Some intervals may have to be shifted in order to accommodate the newly updated cache line. In the worst case, such movement of intervals require $k-1$ cycles, wherein all the $k-1$ intervals need shifting. Thus, our design requires a maximum of $2k+1$ cycles to accommodate a newly updated cache line into the Interval Table $I[k]$. The check for membership of the newly updated cache line in the Interval Table can be done faster by using binary search through all the intervals of the Interval Table. However, we cannot avoid visiting all the intervals as we have to compute *minLocalGap* and *minGlobalGap* required by the algorithm.

C. Update Buffer

We use an ***Update Buffer*** to temporarily store cache line update requests that are received when the hardware is busy processing the current cache line update. During the $2k+1$ cycles described above, the processor is not allowed to update any other cache line. Such stalls could potentially slow down the execution if they occur frequently. To minimize the impact of such stalls, we include a small buffer to hold the addresses of the cache lines that are updated while the hardware is busy. Clearly, as the size of this Update Buffer increases, the number of processor stalls decreases, but the debug-hardware area increases. The designer can control the size of the Update Buffer. We examine its implications in Section V.

V. Experiments

A. Setup

We used a simulation infrastructure to evaluate the impact of our DFD hardware, and synthesized our proposed designs with a 90nm ASIC library. Our simulation setup consists of three components: (i) a Pintool [31] to generate a trace of all the memory addresses accessed by the processor during the execution of a benchmark; (ii) an in-house, functional, cache

simulator which instruments the cache using the traces generated in the previous step; and (iii) A *Validation Engine* which uses the proposed algorithms to maintain the information on the recently updated cache lines.

The cache simulator and the Validation Engine were programmed in C++ on Linux platform. We used trace-driven functional cache simulator for faster simulations. However, to determine the size of the Update Buffer, we used Simplescalar as this required cycle accurate simulations. We also maintained a bit-vector to track the updated cache lines during the simulations, which is necessary to compute the overheads of the proposed methods. The final state of the bit-vector is used as input to the optimal offline algorithm (Algorithm 1).

All simulations discussed in subsections V-B to V-E used L1 caches with 32KB size, 2-way associativity, 32B per line; and L2 caches with 2MB size, 8-way associativity, 128B per line. We used CACTI 6.5 to estimate the cache areas. The designs were implemented in VHDL and synthesized using Cadence Encounter RTL compiler.

B. Overhead vs. Size of Interval Table (k)

We varied the size of the Interval Table from 4 to 32 for 8 different benchmarks (6 and 2 from Mediabench-I & II respectively). The actual value of k need not be limited to powers of two. For our selected L2 cache configuration, the upper limit on k is given by $\frac{16384}{2*log_2(16384)} \approx 585$ intervals, where 16384 is the total number of cache lines.

Figure 6 shows the overhead in the number of cache lines dumped for various sizes of Interval Table. We observe that, as the size of the Interval Table increases from 4 to 32, the overhead decreases for both Optimal and Greedy algorithms, as expected. Moreover, we also observe that the actual overhead is less than 5% when $k \geq 8$ for all benchmarks. The average and maximum overheads of Greedy algorithm when $k = 16$, are 2.1% and 3.72% respectively.

We also observe that the overhead of the Greedy algorithm nearly matches that of the Optimal algorithm. The average additional overhead of the greedy algorithm is under 1% for all benchmarks for all values of k. The maximum additional overhead of 1.7% occurs for *adpcm_dec* when $k = 4$. The maximum additional overhead is just 0.16% when $k = 16$ for *gsm_enc*. These low overheads of the online algorithm can be attributed to the locality of references in cache and memory accesses (as discussed in Section III-E). Our hardware implementation of the Greedy algorithm uses 16 intervals. The choice of k represents a trade-off between area overhead and dumping overhead, and is configurable by the designer based on a profiling of the target applications.

C. t-lines/bit bit-vector vs Interval Table

Figure 7 shows the *overhead* for $t = 2, 4, 8, 16, 32$ and 64. We observe that, as t increases from 2 to 16, the *overhead* increases too. This is expected, as for larger values of t, larger number of cache lines are tracked together. The average and maximum overheads of Greedy algorithm with $k = 16$ lie in between that of 4-lines/bit (1.38% and 2.35% respectively) and 8-lines/bit bit-vectors (2.42% and 4.1% respectively).

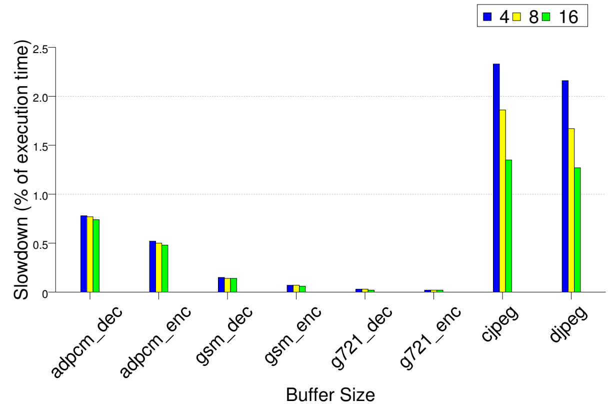

Fig. 8. Slowdown vs. Buffer Size

TABLE I
AREA OVERHEAD

t-lines/bit bit-vector (%)					Greedy algorithm (%)
t=1	t=2	t=4	t=8	t=16	k=16
10.28	4.8	2.21	1.27	0.57	0.64

D. Update Buffer Size vs. Processor stalls

Figure 8 shows the slowdown experienced by the processor as a result of the stalls induced when the hardware (corresponding to Greedy algorithm) is busy when a new cache line update arrives. Since the computational complexity associated with a t-lines/bit bit-vector is limited, the slowdown experienced by processor is negligible and hence is not considered here. We considered Update Buffer sizes of 4, 8, and 16. We observe that for all the benchmarks, as the buffer size increases from 4 to 32, the slowdown decreases. This slowdown is negligible (less than 1%) in 6 out of 8 benchmarks and is maximum (2.3%) in the case of *cjpeg* for buffer size of 4. Based on these observations, we decided to use an Update Buffer of size 4 for the hardware implementation.

E. Area overhead

We synthesized the designs corresponding to the conventional bit-vector, t-lines/bit bit-vector for $t = 2, 4, 8$ and 16 and the Greedy algorithm (as discussed in Section IV) using 90nm technology standard cell library. The processor stalls computed earlier used the delays of these designs. Table I shows the results as a percentage of the overall cache area.

The highlight of this experiment is that the Greedy method, in spite of the additional computational complexity, has an area overhead of only 0.64% as compared to 10.28% of a conventional 1-line/bit bit-vector. The savings due to reduction in storage space is more than the increase in combinational logic due to added processing complexity of the Greedy algorithm. As expected, the area of t-lines/bit bit-vector proportionally decreases as we increase the value of t.

We also observe that the area overhead of the Greedy algorithm is lesser than that of 4-lines/bit and 8-lines/bit bit-vector, although the *dump overhead* of Greedy algorithm lies in betweeen the two (as observed in Section V-C).

F. Overhead vs Cache Size

As the cache size (number of lines) decreases, we reach a break-even point where the overhead and area of Greedy

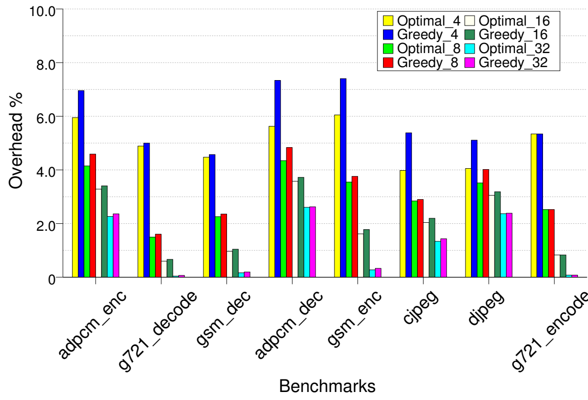

Fig. 6. *Overhead* of Optimal and Greedy algorithms

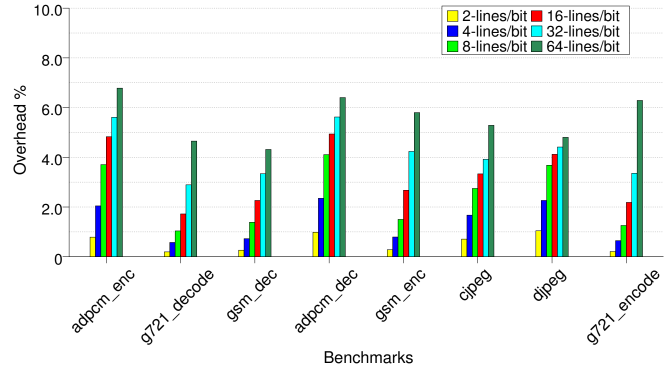

Fig. 7. *Overhead* of t-lines/bit bit-vector

algorithm and a t-lines/bit bit-vector are similar; below this, the t-lines/bit bit-vector is more beneficial than the Greedy approach. Our experiments indicate that the break-even point is 4096 lines (L2 cache of 256KB size, 4-way, 64B per line). For this configuration, the average and maximum overhead of Greedy algorithm with $k = 16$ (2.96% and 4.76%) is closer to that of 4-lines/bit bit-vector (2.56% and 4.35% respectively). The area overheads of Greedy algorithm and 4-lines/bit bit-vector using this configuration was also found to be similar (3.09% and 3.18% respectively). When going from a cache size of 2MB to 256KB, the area of Greedy algorithm showed a marginal downward trend – this is because the table sizes gradually decreased but the other logic was almost constant. This makes the Greedy algorithm ideal for large caches.

VI. Conclusions

In this work, we proposed two space sensitive techniques to keep track of the cache lines that are updated after the previous transfer of L2 cache contents off-chip, during post-silicon validation. One major feature of the proposed DFD hardware is that, the designer can tune them to match his area budget. Our proposed methods use a small fraction of the overall cache area with an average dumping *overhead* of less than 2.5% as compared to over 10% area overhead of a simple bit-vector. For a $90nm$ library, the break-even point at which both the methods perform similarly in terms of overhead and area are caches with 4096 lines. For caches with number of lines less than this, t-lines/bit bit-vector is better. The area of our Interval Table based design is almost independent of the cache size, which makes it ideal for large last level caches.

References

[1] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *DATE, 2008*.

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

[3] Q. Xu and X. Liu, "On signal tracing in post-silicon validation," in *ASP-DAC, 2010*.

[4] P. R. Panda, M. Balakrishnan, and A. Vishnoi, "Compressing cache state for postsilicon processor debug," *IEEE TC, 60(4)*, 2011.

[5] H. Shojaei and A. Davoodi, "Trace signal selection to enhance timing and logic visibility in post-silicon validation," in *ICCAD, 2010*.

[6] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *DAC, 2006*.

[7] K. Goossens, B. Vermeulen, and A. Nejad, "A high-level debug environment for communication-centric debug," in *DATE, 2009*.

[8] H. F. Ko and N. Nicolici, "On automated trigger event generation in post-silicon validation," in *DATE, 2008*.

[9] H. Yi, S. Park, and S. Kundu, "A design-for-debug (DfD) for NoC-based SoC debugging via NoC," in *ATS, 2008*.

[10] A. Gharehbaghi and M. Fujita, "Global transaction ordering in network-on-chips for post-silicon validation," in *(ISQED), 2011*.

[11] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *DAC, 2011*.

[12] A. DeOrio, I. Wagner, and V. Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," in *HPCA 2009*.

[13] S.-B. Park and S. Mitra, "IFRA: instruction footprint recording and analysis for post-silicon bug localization in processors," in *DAC, 2008*.

[14] K.-H. Chang, I. L. Markov, V. Bertacco, " Automating post-silicon debugging and repair." *ICCAD 2007*.

[15] Y.-S. Yang, A. G. Veneris, N. Nicolici, M. Fujita, "Automated data analysis techniques for a modern silicon debug environment." *ASP-DAC 2012*.

[16] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, S. Mitra, " QED: Quick Error Detection tests for effective post-silicon validation." *ITC 2010*.

[17] A. DeOrio, D. S. Khudia, V. Bertacco: Post-silicon bug diagnosis with inconsistent executions. *ICCAD 2011*.

[18] H. F. Ko, N. Nicolici, " Automated trace signals selection using the RTL descriptions." *ITC 2010*.

[19] D. Chatterjee, C. McCarter, V. Bertacco, " Simulation-based signal selection for state restoration in silicon debug." *ICCAD 2011*.

[20] E. Anis, N. Nicolici, " On using lossless compression of debug data in embedded logic analysis." *ITC 2007*

[21] E. A. Daoud, N. Nicolici, " On Using Lossy Compression for Repeatable Experiments during Silicon Debug." IEEE TC, 60(7), 2011.

[22] E. Daoud and N. Nicolici, "Embedded debug architecture for bypassing blocking bugs during post-silicon validation," *IEEE TVLSI, 15(2), 2011*.

[23] L. Lee, L.-C. Wang, T. Mak, and K.-T. Cheng, "A path-based methodology for post-silicon timing validation," in *ICCAD, 2004*.

[24] S.-B. Park, A. Bracy, H. Wang, S. Mitra, " BLoG: post-silicon bug localization in processors using bug localization graphs." *DAC 2010*.

[25] P. Michaud, "Online compression of cache-filtered address traces," *ISPASS, 2009*.

[26] K. Basu and P. Mishra, "Efficient trace data compression using statically selected dictionary," in *VLSI Test Symposium (VTS), 2011*.

[27] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *DAC 2010*

[28] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: an efficient data clustering method for very large databases," *SIGMOD Rec., 1996*.

[29] A. Vishnoi, P. R. Panda, and M. Balakrishnan, "Online cache state dumping for processor debug," in *DAC, 2009*.

[30] A. Kumar, P. R. Panda, and S. Sarangi, "Efficient on-line algorithm for maintaining k-cover of sparse bit-strings," in *FSTTCS 2012*., in press.

[31] V. R. A. Settle, D. Connors, and R. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *WCAE*, 2004.