This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Area-Aware Cache Update Trackers for Postsilicon Validation

Sandeep Chandran, Smruti R. Sarangi, and Preeti Ranjan Panda, *Senior Member, IEEE*

Abstract*— The internal state of the complex modern processors often needs to be dumped out frequently during postsilicon validation. Since the caches hold most of the state, the volume of data dumped and the transfer time are dominated by the large caches present in the architecture. The limited bandwidth to transfer data present in these large caches off-chip 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 two methods: 1) where a bit tracks multiple cache lines and 2) 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 with a bit-vector, and allow the designer to choose the amount of space to allocate for this design-for-debug feature. The impact of reducing storage space is that some nonupdated cache lines are dumped too. We attempt to minimize such overheads. We propose a scheme to share such cache update tracking hardware (or Update Trackers) across multiple caches in case of physically distributed caches so that they are replicated fewer times, thereby limiting the area overhead. We show that the proposed Update Trackers occupy less than 1% of cache area for both the shared and distributed caches.

***Index Terms*— Cache compression, parameterized design-for-debug (DFD) architecture, postsilicon validation, processor debug, state dump-driven debugging.**

I. Introduction

THE increasing complexity of modern processors has resulted in the evolution of sophisticated postsilicon validation features because the simulation techniques are inadequate for executing large test case scenarios [1], [2]. During postsilicon 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 [3]–[5], trace signal selection [6], [7], and adaptation of test compression techniques [8], [9].

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 postsilicon validation are: 1) minimal intrusiveness; 2) minimal space requirements; and 3) maximum visibility into the chip. These requirements are clearly orthogonal, and balancing the three is a complex task.

The entire processor state consists of all caches and registers in various structures such as pipelines, register files, translation lookaside buffers (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, due to the large sizes of the last-level caches, 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 [8], [10]. Such reduction in off-chip transfers also limits the perturbations experienced by the test workloads due to the debug infrastructure. This significantly improves reproducibility of intermittent bugs.

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. This approach is also very efficient, as the amount of processing required is minimal. However, 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 paper, 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.

In case of distributed caches, updates to all the caches need to be tracked in order to obtain a consistent snapshot of the state of the processor. The naïve way to achieve this is to duplicate the Update Tracker for every cache. However, this

Manuscript received April 24, 2015; revised August 6, 2015; accepted August 27, 2015.

The authors are with the Department of Computer Science and Engineering, IIT Delhi, New Delhi 110016, India (e-mail: sandeep@cse.iitd.ac.in; srsarangi@cse.iitd.ac.in; panda@cse.iitd.ac.in).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2015.2480378

1063-8210 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

may result in significantly high area overheads. We aim to limit this area overhead by allowing multiple caches to share the Update Tracker. We propose a scheme under which the number of non-updated cache lines that are dumped off-chip remains under acceptable limits, even when the Update Tracker is shared across multiple caches.

The rest of this paper is structured as follows. Section II outlines the prior research. Section III discusses in detail the motivation and the methodology used for shared caches. Section IV discusses schemes to share the Update Tracker in physically distributed caches. Section V describes the hardware implementation of our proposed method. Section VI explains the experiments and results. Finally, the conclusions are given in Section VII.

II. Related Work

The challenge of limited visibility posed during postsilicon validation is generally handled by two broad approaches: 1) tracing signals and 2) event triggers. Most of the issues that arise when tracing signals are due to the 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 the issues related to tracing signals for debugging purposes [11]–[15]. Event triggers are associated with a different set of research problems, 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 [16]–[20]. Recently, transaction-based debugging methods such as [21] and [22] have been proposed. Other methods have been proposed for inferring the subset of signals responsible for the faulty behavior by observing the manifestations of these signals on other easily observable parameters [23]–[27]. Some research works also target compression [28], [29] and analysis [30]–[32] 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, particularly caches. The other approaches are still applicable in this context, during the processor execution cycle between successive state dumps.

Another related research area is the field of compressing bit-vectors [33]–[35]. This paper aims to track only the cache lines updated after the previous dumping phase, using space lesser than that used by a bit-vector. Therefore, our focus is to eliminate the bit-vector itself.

The field of data clustering is conceptually close to our area of work where n data points are to be grouped into k clusters while minimizing a distance metric. We explored this option by adapting a well-established clustering algorithm BIRCH [36] to suit our application scenario. Through experimentation we found that, in spite of the conceptual similarities, a direct application of this method is relatively inefficient.

	(a)	(b)	(c)	(d)	(e)	(f)
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. (a), (c), and (e) Examples of bit vectors that capture the information on updated cache lines. (b), (d), and (f) The corresponding 2-lines/bit bit vector. The shaded portion of bit vector shows the dumped lines using the interval table with $k = 1$.

This paper is similar in scope to [10] and [37]. Here, we compress the cache contents 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: 1) we attempt to track and dump only the updated cache lines using very little space and 2) we allow the designer to parameterize this Update Tracker based on an area constraint. The compression techniques proposed in the above works can be applied on the cache lines that are marked as updated by our technique to further reduce the size of the data transferred off-chip; this further strengthens the state dump-driven debugging approaches.

III. Methodology—Shared Cache

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. Unless otherwise mentioned, the bit-vector maintains one bit per cache line.

Definition 2: We define the overhead as the number of nonupdated cache lines that are transferred off-chip, expressed as a percentage of the total number of cache lines. It is quantified as $(y - x/N) \times 100\%$, where x is the number of updated lines, y is the number of dumped lines, and N is the total number of lines in the cache.

In this paper, we aim to use lesser space than a bit-vector to track the cache lines that were modified since the previous dump. The motivation to use lesser space is illustrated in the example shown in Fig. 1(a). In this case, of the total 16 cache lines only 11 cache lines have been updated after the previous snapshot (indicated by 1 in the corresponding bit-vector). Moreover, all the updated lines are contiguous. In this scenario, instead of a bit-vector, it is efficient to store only the start and end addresses of the continuous run of 1s. This requires only 8 bits (4 bits for each address), whereas the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

bit-vector requires 16 bits. This example results in a saving of 50% of the number of bits used to store the information on updated lines. Such runs of 1s is not uncommon due to the high spatial locality of reference in caches. 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 a factor of 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 nonupdated cache lines in the proximity of an updated cache line. In Fig. 1, columns (b), (d), and (f) illustrate the t-lines/bit bit-vector for $t = 2$ corresponding to the bit-vectors shown in columns (a), (c), and (e), respectively. The respective overheads incurred are shown in the bottom row.

C. Interval Table

Definition 3: A pair of starting address and ending address ($\langle$start_addr, end_addr$\rangle$) 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 1's 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 1's. The shaded portion in columns (a), (c), and (e) of Fig. 1 shows the set of cache lines marked as updated by an Interval Table with $k = 1$.

There are two adversarial situations to the interval-based approach in practice: 1) in a bit-vector, the actual number of runs of 1's can be very large and 2) the length of some runs can be smaller than the number of bits required to store their starting and ending addresses. To overcome the former issue, we merge some adjacent intervals to reduce the number of stored intervals to a given constraint k. The effect of the latter is sought to be overcome by the space saved when there are long runs of 1's. Since $\log_2 N$ bits are needed to address N cache lines, storing k intervals requires $2k\log_2 N$ bits, and it is essential that $2k\log_2 N < N$ (or $k < (N/2\log_2 N)$) 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 n runs of 1's in just k intervals where $n \gg k$.

Merging adjacent intervals due to the upper bound on k, will capture some nonupdated cache lines into the intervals, which leads to overheads. Thus, the intervals with the smallest runs of nonupdated cache lines between them should be merged. However, determining the nearest intervals is nontrivial 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. Fig. 2 illustrates this issue with $k = 2$.

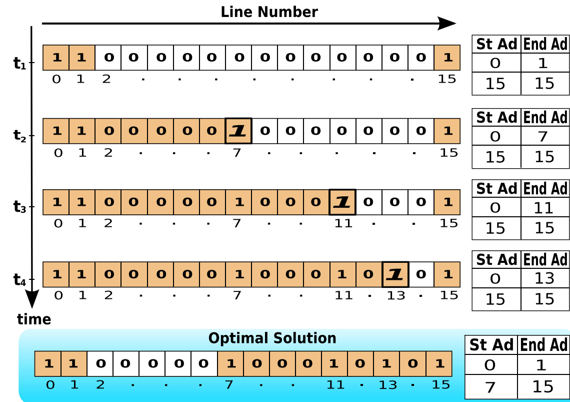

Fig. 2. Interval merging decisions for $k = 2$.

Initially, at time t_1, only the cache lines 0, 1, and 15 are updated; this is represented easily in two intervals $\langle 0, 1\rangle$ and $\langle 15, 15\rangle$. At time t_2, line 7 is updated (highlighted in the figure). Now, we can merge 7 with either $\langle 15, 15\rangle$ or $\langle 0, 1\rangle$. We merge it with $\langle 0, 1\rangle$ as line 7 is closer to it. The resulting interval is $\langle 0, 7\rangle$. Next, say the cache line 11 gets updated at time t_3. The newly updated line 11 is now equidistant from the current intervals ($\langle 0, 7\rangle$ and $\langle 15, 15\rangle$). Let us assume we merge 11 with the interval $\langle 0, 7\rangle$ resulting in the new interval $\langle 0, 11\rangle$. Similarly, the update to Line 13 at t_4 results in the interval $\langle 0, 13\rangle$. We observe that the final two intervals $\langle 0, 13\rangle$ and $\langle 15, 15\rangle$ include nine 0's, corresponding to a 56% overhead. If the merging strategy was different, it could be defeated by a different sequence of future accesses. 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 Offline 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. The solution returned by the Offline 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 nonupdated 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 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 0's. This is shown in Fig. 3(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 [Fig. 3(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. Fig. 3(c) and (d) indicates

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1 Offline Algorithm (Optimal)

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

Cache Lines / Bit Vector

Cache Lines	(i)	(ii)	(iii)
0	1	1	1
1	1	1	1
2	0	0	0
3	1	1	1
4	0	0	0
5	0	0	0
6	1	1	1
7	1	1	1
8	1	1	1
9	0	0	0
10	1	1	1
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	1	1	1

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

(a)

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

(b)

St. Ad	End Ad
0	10
15	15

(c) - Bit vector (ii)

St. Ad	End Ad
0	3
6	10
15	15

(d) - Bit Vector (iii)

Fig. 3. 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 $k = 3$.

the optimal set of intervals ($O[k]$) for $k = 2$ and $k = 3$, respectively. The corresponding bit-vectors are shown in Fig. 3(ii) and (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 1's exceeds k, we merge the adjacent intervals to form a larger interval. Since this action causes us to include some 0's in the interval, we select for merging two adjacent intervals with the minimum Gap between them. Of course, some of the included 0's may get updated to 1 in the future due to spatial locality of accesses.

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

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: **if** minLocalGap $<$ minGlobalGap **then**
12: $\quad$extend the nearest interval to accommodate lineNum
13: **else**
14: $\quad$merge the intervals around minGlobalGap
15: $\quad$place the lineNum as a separate interval
16: **end if**

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 a 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 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.

Fig. 4(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 $\langle 8, 8\rangle$) is smaller than minGlobalGap (for Gap $\langle 11, 13\rangle$), and hence, only the nearest stored interval ($\langle 9, 10\rangle$) is extended to accommodate 7. Fig. 4(b) shows the final state of the Interval Table $I[k]$ and the corresponding bit-vector.

Fig. 5 shows an example where minGlobalGap is smaller than minLocalGap. In this case, the intervals around the minGlobalGap ($\langle 0, 2\rangle$ and $\langle 5, 6\rangle$) are merged [as shown in Fig. 5(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 ($\langle 14, 14\rangle$).

This algorithm requires minimal amount of storage as the information required for deciding the intervals to be merged is derived by iterating once through the Interval Table. We observe in our experiments that the overhead of the Greedy

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

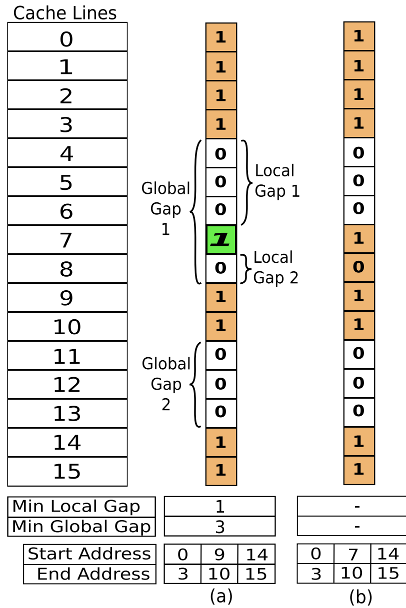

Fig. 4. (a) Illustration of Local Gaps and Global Gaps. (b) Final bit-vector on accommodating new incoming request.

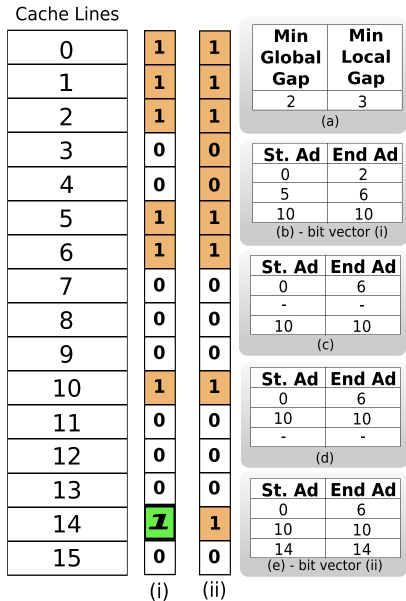

Fig. 5. Illustration of Greedy algorithm when minGlobalGap is smaller than minLocalGap. (a) The minLocalGap and minGlobalGap for the bit vector (i). (b), (c), (d), and (e) The series of steps through which the Greedy algorithm accommodates a newly updated cache line.

algorithm tracks that of the Offline algorithm well for most of the benchmarks.

The advantages of Greedy algorithm are: 1) the storage space required is independent of the number of cache lines unlike bit-vector, and hence, is suitable for large caches and 2) 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 number of cache lines dumped by the Greedy algorithm is at most twice that dumped by the Offline algorithm (2 competitive) is given in [38].

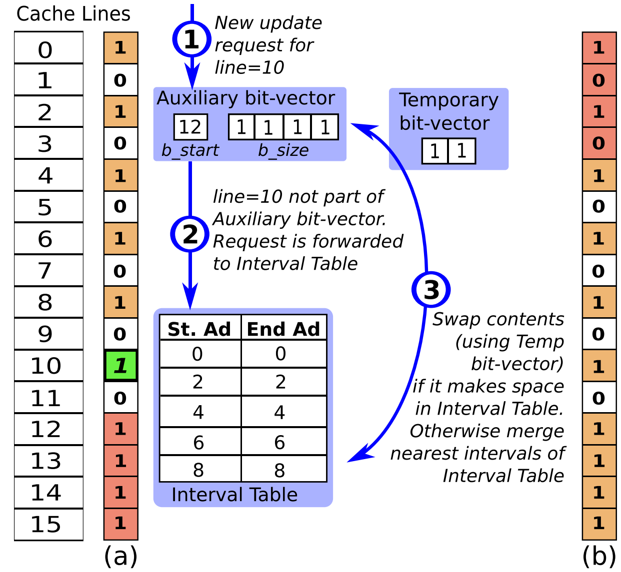

Fig. 6. Hybrid algorithm (cache lines shaded in pink and yellow are captured in auxiliary bit-vector and Interval Table, respectively). (a) State when a new line (line 10) is updated. (b) State after accommodating the update request.

F. Method 3: Hybrid Algorithm

Definition 7: The Density (D) of a given window is defined as the number of intervals in the window of cache lines. A window with high density indicates large number of updated cache lines in the window that are not contiguous.

The Hybrid algorithm extends the Greedy algorithm by maintaining an additional bit-vector called auxiliary bit-vector (of size *b_size*). This bit-vector is used to store the update information of the most dense window of size *b_size*. Since this region can change during the online operation of the cache, we associate a start address *b_start* with this bit-vector. The updates to the cache line at *b_start* and the next (*b_size*-1) lines are tracked by the auxiliary bit-vector. A temporary bit-vector is used for swapping intervals between the Interval Table and the auxiliary bit-vector, as and when the need arises. Since this temporary bit-vector is used only for swapping, we use a t-bit bit-vector with ($t = 2$) as the temporary bit-vector to limit the area overhead.

The Hybrid algorithm extends the Greedy algorithm in two ways: 1) the auxiliary bit-vector filters the update requests before sending them to the Interval Table and 2) update information from the bit-vector is swapped with that of the Interval Table in case it leads to freeing up of intervals in the Interval Table. The check to determine whether or not a window of length *b_size* exists in the Interval Table with higher number of intervals than that in the bit-vector is performed just before merging the intervals around minGlobalGap (Line 14 of Algorithm 2). Fig. 6 details the sequence of steps performed by the Hybrid algorithm to accommodate an update request to a new cache line.

The reduction in the dumping overhead due to Hybrid algorithm increases with the size of the auxiliary bit-vector. However, a large auxiliary bit-vector results in increased area overhead as it also warrants a large temporary bit-vector. Moreover, the Hybrid algorithm requires multiple passes over

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

the Interval Table to find the window that can be swapped with the bit-vector.

IV. Methodology—Distributed Caches

As mentioned in Section I, the straightforward way to track updated cache lines in distributed caches is to maintain the update information of each cache separately. This results in a linear increase of area overhead with the increasing number of caches.

One way to reduce area overhead is to share the Update Trackers across multiple caches. This requires the Update Trackers to be replicated only $N_t = (N/c)$ times (instead of N times) where N is the total number of caches and c ($\geq$1) is the number of caches sharing the structure.

The case of $c = N$ corresponds to a central structure that tracks updated cache lines in all N caches. Since there is no replication here, the area overhead is minimal. However, such a central structure may become a bottleneck in the system due to signal delay considerations. On the other hand, $c = 1$ corresponds to the architecture where each cache has a separate Update Tracker to track its updates. This distributed architecture does not create any bottlenecks, but replicates the hardware, thereby increasing the area overhead. Therefore, the challenge is to determine the configuration where sharing the Update Tracker: 1) occupies limited area; 2) does not create bottlenecks in the system; and 3) dumping overhead (even with sharing) is under acceptable bounds.

Definition 8: We define aggregate overhead in distributed caches (O) to be the total number of nonupdated cache lines across N distributed caches, expressed as a percentage of the total cache lines across all the caches

$$O = \frac{\sum_{i=1}^{N}(\text{number of non-updated cache lines})_i}{\sum_{i=1}^{N}(\text{total number of cache lines})_i} \times 100\%. \tag{1}$$

The above definition is independent of c. In order to keep the area overhead constant when sharing an Update Tracker across c caches, we merge the update information of c cache lines into a single cache line. Such merging of update information can be done in two different ways: 1) horizontal sharing, where the update information of the cache lines at a particular line number spread across c caches is merged or 2) vertical sharing, where the update information of c adjacent cache lines of the same cache is merged (similar to t-lines/bit bit-vector).

A. Horizontal Sharing

Fig. 7 shows an example of two caches A and B sharing a single Update Tracker (bit-vector and Interval Table). The bit-vector and the Interval Table shown in Fig. 7(b) hold the update information of both the caches (A and B) shown in Fig. 7(a). The additional lines that are dumped from each cache due to sharing the bit-vector are marked in blue. The shared bit-vector is the bitwise-OR of the exclusive bit-vectors of A and B.

However, this scheme suffers from increased overheads in case of differential activity in the c caches. For example,

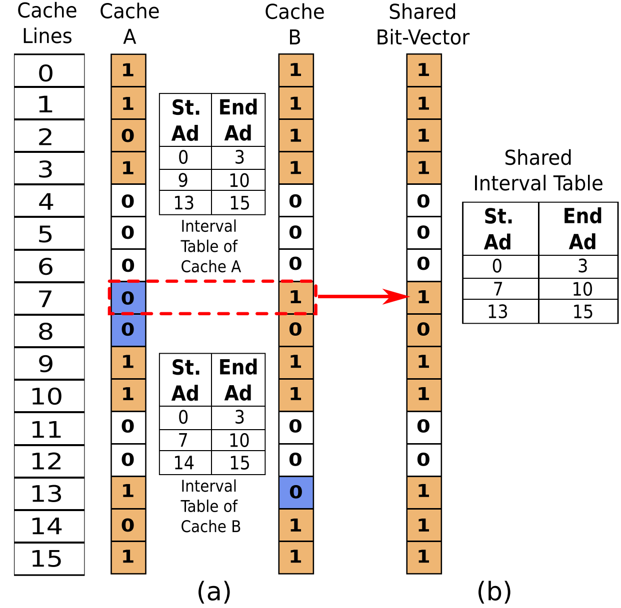

Fig. 7. Horizontal sharing; cache line 7 of the shared bit-vector is the OR of Line 7 of A and B. (a) Caches A and B maintain separate Update Trackers (bit-vector and Interval Table). (b) Caches A and B share the Update Tracker.

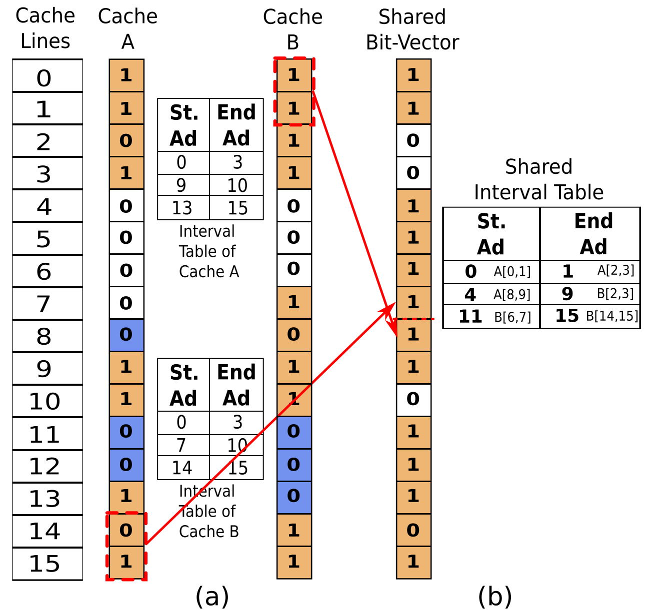

Fig. 8. Vertical sharing; cache line 7 of the shared bit-vector is the OR of Lines 14 and 15 of A (similar to t-bit vector for $t = 2$). (a) Caches A and B maintain independent Interval Tables. (b) Caches A and B shares an Interval Table.

if only one of the c caches is updating its cache lines in a particular epoch, cache lines with that line number in the other $c - 1$ caches will also be dumped. Therefore, the dumping overhead now depends on the number of caches sharing the Update Tracker and the activity in each cache. If the Update Tracker is an Interval Table, the dumping overhead may additionally depend on the sequence of accesses seen by each table.

B. Vertical Sharing

Fig. 8 shows an example of vertical sharing of Update Trackers (bit-vector and Interval Table) between two caches.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

The bits shaded in blue in Fig. 8(a) are the additional cache lines that are dumped due to this sharing scheme. Under this scheme, merging the adjacent c cache lines in each of the caches helps us keep the range of line numbers seen by the Interval Table the same as that of a single cache. By merging the adjacent c cache lines, we exploit the spatial locality of the references to a cache to limit the area overhead.

The advantage of this scheme is that it allows for flexible sharing of the Interval Table between the c caches. If one (out of c) cache has long runs of 1's, and therefore, requires fewer intervals, the other caches will effectively have more intervals available for storing their update information. This also makes the dumping overhead independent of the activity of each cache. An idle cache does not dump anything, which is desirable.

V. Hardware Design

In this section, we discuss the design choices for implementing the Update Trackers. 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) because it allows minLocalGap and minGlobalGap in Algorithm 2 to be computed in a single pass. We notice that the sorted order of intervals is disturbed by a newly updated cache line only in the case when minGlobalGap is smaller than minLocalGap. We restore the sorted order of the table through a sequence of swaps that shift some stored intervals. Fig. 5(b)–(e) shows the sequence of operations when minGlobalGap is smaller than minLocalGap. The movement operation requires $O(k)$ time.

B. Memory Configuration

An appropriate memory architecture needs to be selected to support the efficient computation of minLocalGap. Fast parallel mechanisms for computing the minimum of n values require $\log_2(n)$ comparisons (using a tree of comparators). In an aggressive design using b separate dual-ported banks, we can read out $2b$ intervals simultaneously, and find the minimum distance among these $2b$ values in $\log_2(2b)$ cycles (assuming one cycle per comparison). The same operation requires b cycles if the table is stored in a single-bank dual-port memory, with sequential accesses of a pair of intervals at a time. The difference in time is pronounced only for large values of b. The aggressive parallel implementation requires a relatively large number of subtractors (to calculate the distances) and comparators. Due to the above considerations, we decided to use a single-bank dual-ported memory for storing the Interval Table. Since the value of k is expected to be small, the sequential processing does not impose significant delays, as observed in our experiments.

C. Logic Design

Fig. 9 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, and computation of minLocalGap and minGlobalGap, are 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 $I[k]$.

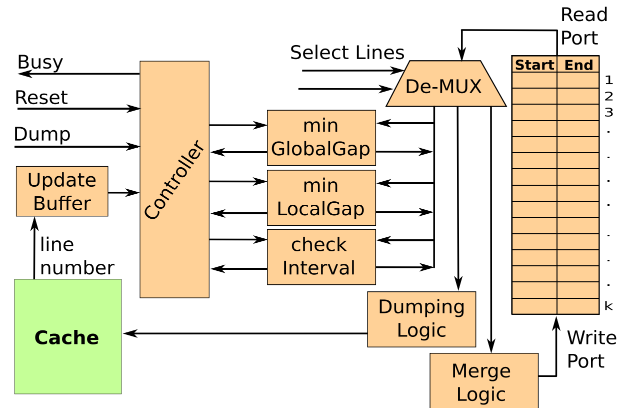

Fig. 9. Hardware design of Greedy algorithm.

Reading and computing minLocalGap and minGlobalGap require k cycles. The merging operation would take two 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 requires $k - 1$ 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 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.

D. 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 VI.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

E. Extensions for Hybrid Algorithm

The controller of the Hybrid algorithm extends the controller of the Greedy algorithm in two ways: 1) it filters the incoming update requests through the auxiliary bit-vector before sending them to the Interval Table and 2) it swaps the update information contained in the Interval Table with that stored in the auxiliary bit-vector. The first extension is simple, and is not discussed further. The second extension involves: 1) scanning the Interval Table to identify the most dense window of size *b_size* and 2) swapping the update information between the Interval Table and the auxiliary bit-vector (using the temporary bit-vector).

We maintain two pointers: 1) start of window and 2) end of window to identify the most dense window in the Interval Table that is to be swapped. Initially, the start pointer points to the starting address of the first interval of the Interval Table and the end pointer points to the ending address of the next interval. In case the number of lines captured between the two pointers is smaller than the size of the auxiliary bit-vector (*b_size*), the end pointer is incremented to point to the ending address of the next interval; otherwise, the start pointer is incremented to the next interval, and the scan for a window starts over. This repeats until the start pointer points to the penultimate interval. In the worst case, this operation takes up to $((k-1)k/2)$ cycles, but in practice, we have observed that this step takes a maximum of $3.5k$ cycles (for $k = 32$). We avoid scanning the bit-vector to determine the number of intervals in it by maintaining the count during online operation. The number of intervals is incremented by 1 if both the bits adjacent to the bit updated in the bit-vector are 0. Similarly, the count is decremented by 1 if both the adjacent bits are 1.

Swapping of the update information between the auxiliary bit-vector and the intervals identified above is done using the temporary bit-vector. During the online operation, it is possible that the update information is stored in both the Interval Table and the auxiliary bit-vector. This happens when the minGlobalGap is less than minLocalGap, the number of intervals in bit-vector is more than the number of intervals in a window in the Interval Table, and the bit-vector lies in the minGlobalGap. This occurs very rarely and is handled by filtering the cache lines again through the bit-vector at the time of dumping. The hardware that filters the incoming requests is reused here.

VI. Experiments

A. Setup

We used a simulation infrastructure to evaluate the impact of our Update Trackers, and synthesized our proposed designs with a 90-nm ASIC library. Our simulation setup consists of three components as follows.

1) A pintool [39] to generate a trace of all the memory addresses accessed by the processor during the execution of a benchmark.
2) An in-house, functional, cache simulator which instruments the cache using the traces generated in the previous step.
3) 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 a trace-driven functional cache simulator for faster simulations. To determine the size of the Update Buffer in the case of a single shared cache, we used Simplescalar as this required cycle-accurate simulations. However, since Simplescalar does not support multiple threads, we used an in-house, cycle-accurate simulator [40] to evaluate the size of the Update Buffer in multicore systems with the distributed caches. We also maintained a bit-vector to track the updated cache lines during 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 Offline algorithm (Algorithm 1).

All simulations discussed in Sections VI-B–VI-F used L1 caches with 32-kB size, two-way associativity, 32 B/line; shared L2 caches with 4-MB size, eight-way associativity, 128 B/line. For experiments with distributed caches, we used 16 distributed L2 caches, each of 256 kB size, four-way associative, and 64 B/line (total 4 MB). We used an auxiliary bit-vector of size 256 bits for evaluating the hybrid scheme.

We discarded the first 100 million memory references when collecting the memory traces using pintool. This helped us capture a region of execution where the threads of some applications under consideration updated cache lines of a select few caches, while in other applications, the cache lines of all the caches were updated. This gave us a mix of applications that is representative of possible load conditions under which the Update Tracker is expected to perform. We simulated five million memory accesses between consecutive cache dumps. For cycle-accurate simulations, we allowed the caches to warm up for a maximum of two million cycles.

We used CACTI 6.5 to estimate the cache areas. The designs were implemented in VHSIC Hardware Description Language (VHDL) and synthesized using Cadence Encounter RTL Compiler.

B. Overhead of Various Schemes—Shared Cache

We varied the size of the Interval Table from 8 to 32 for eight different benchmarks (6 and 2 from Mediabench-I & II, respectively). The actual value of k need not be limited to powers of two. In general, smaller values of k are the more important values to explore, because the smaller Interval Tables lead to area-efficient designs compared with a bit-vector. For our selected L2 cache configuration, the upper limit on k is given by $(32\,768/2\log_2(32768)) \approx 1092$ intervals, where 32 768 is the total number of cache lines.

Fig. 10 shows the dumping overhead for various sizes of Interval Table. We observe that, as the size increases from 8 to 32, the respective overheads of all schemes (Offline, Greedy, and Hybrid algorithms) decreases significantly for all benchmarks. This is expected because using higher number of intervals reduces the number of times the adjacent intervals are merged. Therefore, fewer nonupdated cache lines are captured in the Interval Table.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

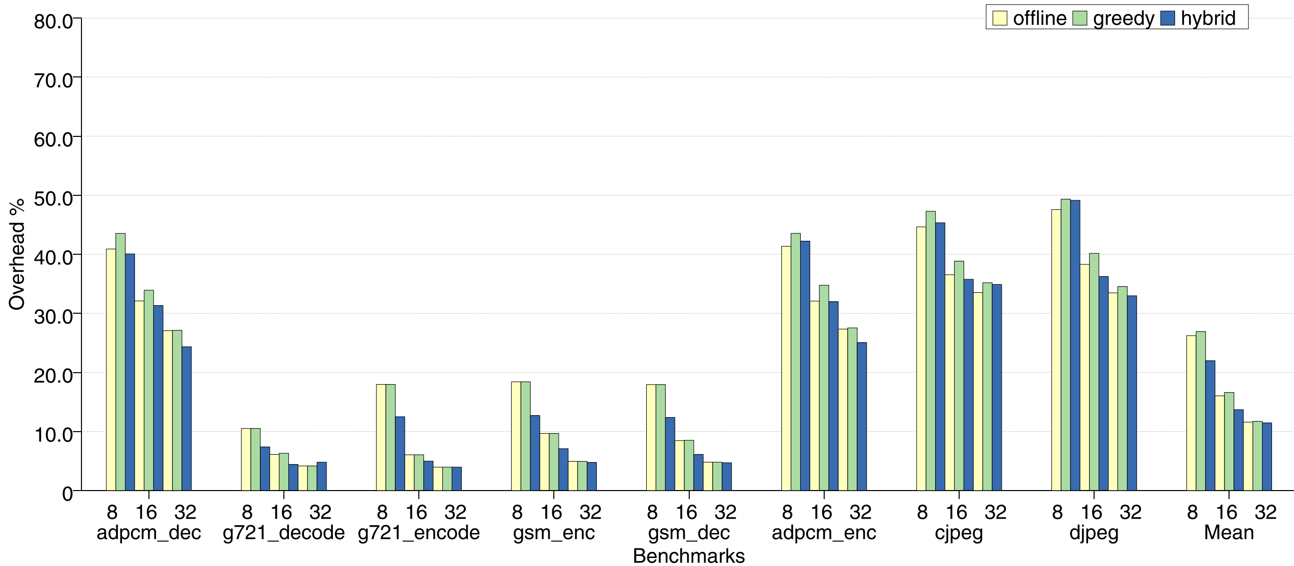

Fig. 10. Overhead of Offline, Greedy, and Hybrid algorithms for different values of k (shared cache).

We also observe that the overhead of the Greedy algorithm closely follows that of the Offline algorithm across all benchmarks for all the values of k. The maximum difference in the overheads of the Offline and Greedy algorithms is 2.65% in the case of *adpcm_dec* for $k = 8$. For $k = 32$, the maximum difference is just 1.7%, and the average is 0.38%.

The Hybrid algorithm offers the best results in case of shared caches for $k = 8$ and $k = 16$. In case of $k = 32$, the performance of the Greedy and Hybrid algorithms are similar. We observe that the dumping overhead of the Hybrid algorithm is significantly smaller than that of the Offline (and Greedy) algorithm for $k = 8$ and $k = 16$ for some applications like *g721_decode*, *gsm_enc*, and *gsm_dec*. The overhead of the Hybrid algorithm is less than that of the Offline algorithm because the Hybrid algorithm merges adjacent intervals less frequently. The average overhead of the Hybrid algorithm is 22% and 13.7% for $k = 8$ and $k = 16$, which is relatively high. Hence, we fixed the amount of storage required by each method to the amount of storage required by an Interval Table of size $k = 32$ for the remaining experiments and for our hardware implementation. Table I shows the number of cache lines transferred off-chip for all benchmarks under each scheme.

Since we need 15 bits to address 32 768 cache lines, each interval requires $2 \times 15 = 30$ bits. Therefore, the total storage space used for evaluating the rest of the experiments is $30 \times 32 = 960$ bits, unless explicitly mentioned.

Fig. 11 shows the overhead for $t = 2, 4, 8, 16, 32$, and 64 (refer Section III-B). We observe that, as t increases from 2 to 16, the overhead increases too. This is expected, as for larger values of t, a larger number of cache lines are tracked together.

C. Overhead of Various Schemes—Distributed Caches

We simulated a multiprogrammed environment where the 16 cores of a system are shared by two multithreaded

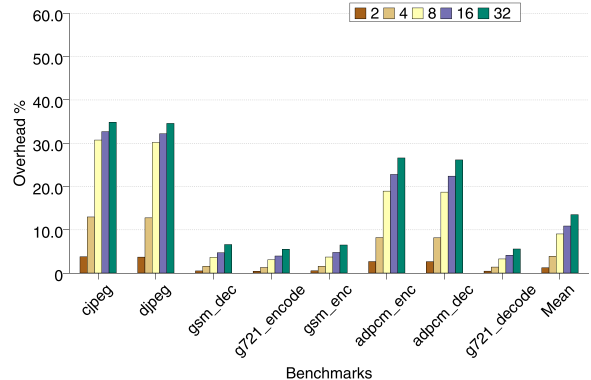

Fig. 11. Overhead of t-lines/bit bit-vector—shared cache.

applications. We evaluated the dumping overhead of various algorithms for horizontal and vertical sharing schemes for $c = 1, 2, 4, 8$, and 16. We randomly scheduled the application threads onto the cores of the system. We repeated the simulation ten times, and have reported the average, maximum, and minimum values observed for the application pair here. We used five different pairs of benchmarks, each chosen from Parsec and Splash suites, to evaluate our proposal. We have not followed any specific policy to pair the applications, but just ensured that the application pairs captured varied activity between them. Therefore, we now have application pairs (*dedup* + *vips* and *barnes* + *ocean*) where all threads of both the applications update large sections of their respective caches (high-activity by all threads), an application pair (*streamcluster* + *x264*) where very few threads in both the applications update small regions of the cache (low-activity by all threads), an application pair (*radix* + *lu*) where some threads update large regions (high-activity by moderate number of threads), and an application pair (*facesim* + *fmm*) where some threads are updating small regions of cache (low-activity by moderate number of threads). This set of application pairs serves as a useful benchmark

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE I
NUMBER OF CACHE LINES TRANSFERRED OFF-CHIP FOR SHARED AND DISTRIBUTED CACHES, RESPECTIVELY

Benchmark	bit-vector	Offline	Greedy	Hybrid
adpcm_dec	996	9870	9886	8975
g721_decode	157	1520	1520	1730
g721_encode	144	1442	1442	1442
gsm_enc	174	1792	1792	1730
gsm_dec	170	1743	1743	1710
adpcm_enc	1009	9968	10034	9218
cjpeg	1897	12884	13428	13333
djpeg	1881	12848	13199	12688

Benchmark	bit-vector	tBits	Offline	Greedy	Hybrid
streamcluster+x264	4809	5046	5994	5996	6018
barnes+ocean	18426	18432	18432	18432	18432
dedup+vips	18313	18386	18416	18432	18432
facesim+fmm	13086	14290	18236	18344	18324
radix+lu	16208	16814	17756	18060	18066

Note: The figures shown above are for k=32 (*b_size*=256 for Hybrid), and c=2 (for distributed caches). The entire cache (32k lines) will be transferred off-chip for all benchmarks in the absence of the proposed schemes.

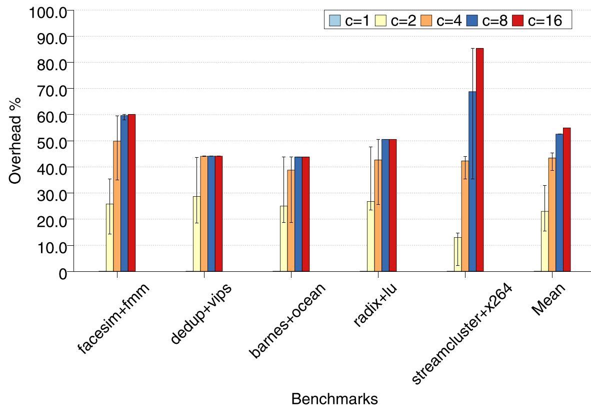

Fig. 12. Overhead of a simple bit-vector during horizontal sharing (distributed cache).

to study the performance of the algorithms under various scenarios that arise during postsilicon validation of the chip.

1) Horizontal Sharing: Fig. 12 shows the dumping overhead of a simple bit-vector that is shared horizontally across c caches. We observe that as the number of caches sharing the bit-vector increases from $c = 1$ to 16, the dumping overhead increases. This is due to an increase in the number of nonupdated cache lines present across different caches that are dumped due to sharing of the bit-vector. The dumping overhead is bad for low-activity application pairs for higher values of c even in case of a bit-vector. Moreover, for all the application pairs, the dumping overhead under horizontal sharing scheme is very high for $c \geq 4$. We do not show the dumping overhead of Interval Table because it is similar to that of bit-vector (mean difference of less than 5%).

We also observe that the dumping overheads under the horizontal sharing scheme is sensitive to the scheduling of the application threads on the cores. A maximum variation of 50% between two runs is observed in case of *streamcluster* + *x264* for $c = 8$. We do not see any variations across multiple runs for $c = 16$ because the update information of all caches is tracked by a single Update Tracker.

2) Vertical Sharing: Fig. 13 shows the dumping overhead of a bit-vector (conceptually similar to a t-bit vector in case of shared cache with $t = c$), Offline, Greedy, and Hybrid algorithms when shared vertically across c caches. The overhead of the t-bit vector is the minimum overhead incurred due to vertical sharing of the Update Tracker. This overhead is due to merging the adjacent cache lines.

We observe that as c increases from 1 to 16, the dumping overhead of the t-bit vector increases too. This is expected because the number of adjacent lines that are merged also increases from 1 to 16, resulting in higher number of nonupdated lines that are adjacent to an updated cache line being dumped. We also observe that the mean dumping overhead of this scheme is relatively small (<2%), for all the values of c, in contrast to the horizontal sharing scheme. We also observe that the variation in dumping overheads between successive runs is significantly lower compared with that of the horizontal sharing scheme. The maximum variation of 2.43% is seen in case of the Hybrid algorithm in *radix* + *lu*. This shows that the vertical sharing scheme is robust in the presence of scheduling of the application threads.

We also notice that the performance of the Greedy and Hybrid algorithms are comparable with the performance of the Offline algorithm as c increases from 1 to 16. The maximum difference in the average dumping overheads of the Offline and the Greedy algorithm is just 1.44% and is seen in the case of *radix* + *lu* for $c = 4$. Similarly, the maximum difference in average overheads of the Offline and the Hybrid algorithm is 1.28%, again in the case of *radix* + *lu* for $c = 4$. The additional complexity in maintaining the auxiliary bit-vector did not translate to significant savings in the dumping overhead. Table I shows the number of cache lines transferred off-chip for all benchmark-pairs for each algorithm.

Another interesting observation is that the average dumping overhead of the Interval Table-based algorithms (Offline, Greedy, and Hybrid) tends to decrease as we go from $c = 1$ to 8, and then marginally increases when we go from $c = 8$ to 16 in some cases. For example, in case of *radix* + *lu*, the dumping overhead decreases from 5.7% to 2.69%. In case of *streamcluster* + *x264* and *facesim* + *fmm*, the dumping overhead increases from 3.1% to 3.2%, and from 15.4% to 15.7%, respectively, when c increases from 8 to 16. This is because, as c increases, there are two opposing forces that comes into play: 1) the savings in dumping overhead due to increased number of intervals per Interval Table and 2) the increase in overhead due to higher number of cache lines tracked per line number. The final dumping overhead is influenced by the relative weights of the two for a given configuration.

Based on these observations, we conclude that vertical sharing is an appropriate strategy for tracking the updates to cache lines in distributed caches. We prefer higher values

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

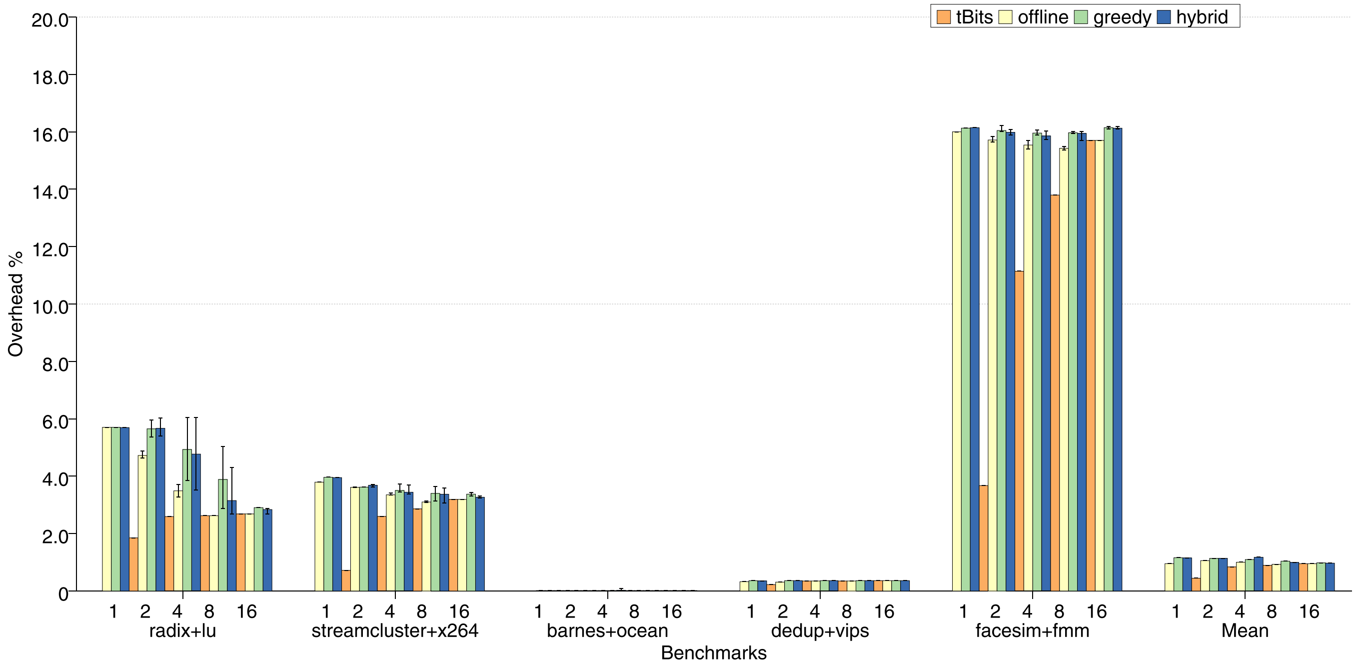

Fig. 13. Overhead of Offline, Greedy, and Hybrid algorithms for different values of k (distributed cache).

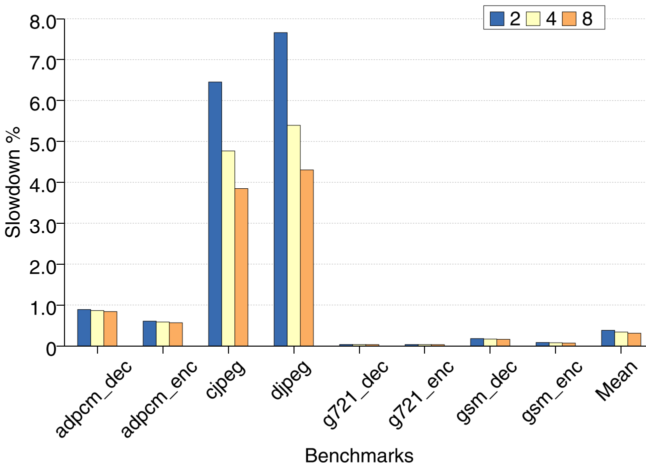

Fig. 14. Slowdown versus buffer size (shared cache).

of c, because the control logic that maintains the Interval Table would then have to be replicated fewer number of times, thereby leading to greater reduction in area overhead.

D. Update Buffer Size Versus Processor Stalls—Shared Cache

Fig. 14 shows the slowdown experienced by the processor as a result of the stalls induced when the hardware corresponding to the 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 the processor is negligible and, hence, is not considered here. We considered Update Buffer sizes of 2, 4, and 8. As mentioned earlier (in Section V-D), the Update Buffer size is an independent parameter that can be chosen by the designer. We expect this buffer to be small, in keeping with the area-efficiency goal of the design. We observe that for all the benchmarks, as the buffer size increases from 2 to 8, the slowdown decreases. This slowdown is negligible (less than 1%) in 6 out of 8 benchmarks and is maximum (7.66%) in the case of *djpeg* for buffer size of 2. The maximum slowdown is only 5.4% for buffer of size 4. Based on these observations, we decided to use an Update Buffer of size 4 for the hardware implementation. We do not show the slowdown experienced by the processor due to stalls induced by the hardware corresponding to the Hybrid algorithm (for $k = 32$) because it is as high as 13×.

E. Update Buffer Size Versus Processor Stalls—Distributed Caches

Fig. 15 shows the slowdown experienced by the application pairs in case of distributed caches due to the stalls induced when a new update request finds the Interval Table corresponding to the Greedy algorithm busy. We considered Update Buffer sizes of 2, 4, and 8. We modeled a 5×5 NoC, with a hop-delay of three cycles. The caches and the Interval Table are connected to different routers. Therefore, a network delay of 6 and 12 cycles is encountered for $c = 2$ and $c = 4$, respectively. The delays obtained after synthesis of the hardware is used in this experiment.

We notice that the slowdown experienced in the case of $c = 2$ is lesser than that of $c = 4$ for all sizes of update buffers. This is because of two reasons: 1) longer hardware and network delays involved and 2) higher contention due to sharing of the Interval Tables. We also observe that as the Update Buffer size increases from 2 to 8, the mean slowdown experienced decreases from 10.3% to 0.2%, and 35.1% to 8.8% in the case of $c = 2$ and $c = 4$, respectively. Based on these observations, we chose $c = 2$ with an Update

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

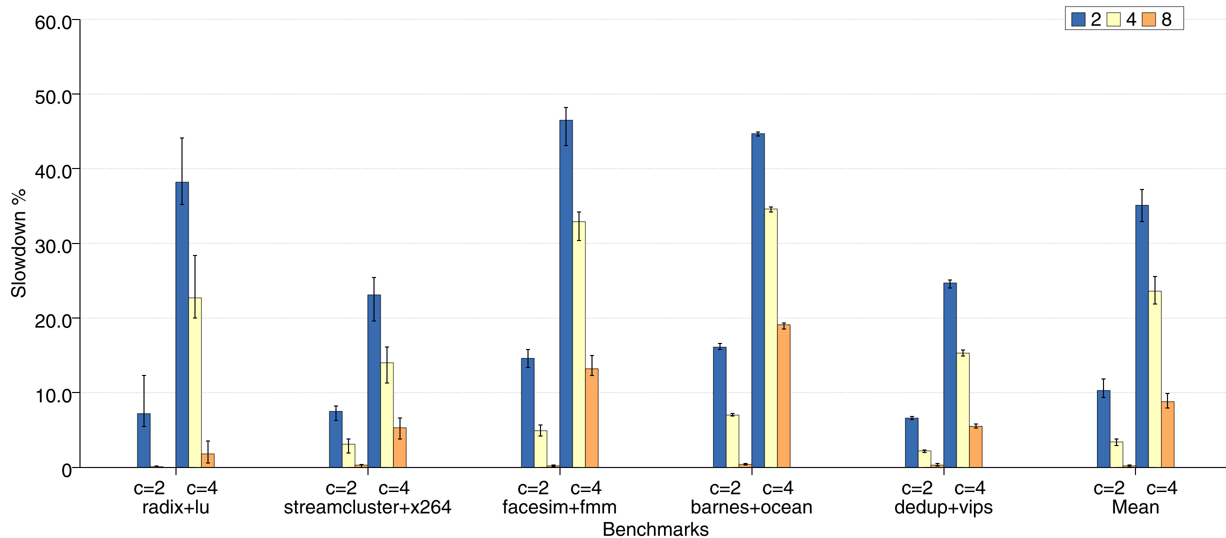

Fig. 15. Slowdown versus buffer size (distributed cache).

TABLE II

AREA OVERHEAD—SHARED CACHE

	t-lines/bit bit-vector					Greedy	Hybrid
	t=1	t=2	t=4	t=8	t=16	k=32	k=32
mm^2	1.293	0.637	0.294	0.145	0.079	0.057	0.0836
%	11.76	5.8	2.69	1.32	0.72	0.525	0.76

Buffer of size 4 as the configuration for implementing the hardware.

The mean slowdown experienced by the processor due to the hardware implementation of the Hybrid algorithm for $c = 2$ and 4 (or $k = 4$ and 8, respectively) is as high as 53.9% and 34.5% in case of distributed caches for Update Buffer of size 4 and 8, respectively.

F. Area Overhead

1) Shared Cache: 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 with $k = 32$ (for the shared cache configuration) as discussed in Section V using 90-nm technology standard cell library. The processor stalls computed earlier used the delays of these designs. Table II shows the results as a percentage of the overall cache area for the shared cache.

The highlight of this experiment is that the Greedy method, in spite of the additional computational complexity, has an area overhead of only 0.525% as compared with 11.76% 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.

2) Distributed Caches: We synthesized a bit-vector of 4096 bits. This bit-vector is shared by two caches in our proposed architecture. The area occupied by a bit-vector in this case is 0.74%. Therefore, the overall area overhead of maintaining eight such bit-vectors is 5.92%.

We synthesized the Greedy algorithm with $k = 4$ (corresponds to $c = 2$), as the above experiments indicated it to be the optimal point. The area occupied by the Greedy algorithm (including an Update Buffer of size 4) is 0.0176 mm^2 which is 0.09% of a distributed cache. For $c = 2$, we have eight such Interval Tables, and hence, the overall area overhead is $0.09 \times 8 = 0.72\%$.

Similarly, we also synthesized the Hybrid algorithm with auxiliary bit-vector of size ($b_size = 256/8 = 32$) bits for the same configuration. The area occupied by the Hybrid algorithm per Interval Table is 0.0253 mm^2 which is 0.13% of the overall cache area. We duplicate this 8 times for $c = 2$, and hence the area overhead is 1.034% of the overall cache area.

G. Power Overhead

The Greedy algorithm consumes 17.18 and 7.94 mW per cycle of operation in the case of shared and distributed caches, respectively, which is 0.5% and 1.95% of the total read dynamic power per read port of the corresponding caches for a toggle rate of 0.02 toggles per ns, and 0.5 probability of a signal being high.

H. Summary

The results indicate that the Greedy algorithm strikes a good balance between the dumping and area overhead well, in case of both shared and distributed caches for $k = 32$. In case of distributed caches, even though the vertical sharing scheme allows a fully centralized scheme that does not require any duplication of the Interval Table, the contention at the Interval Table and the network delays make sharing of the Interval Table difficult beyond two caches ($c = 2$).

The dumping overhead of the Hybrid algorithm is marginally smaller than that of the Greedy algorithm in case of both shared and distributed caches, but the area overhead is

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

higher than that of the Greedy algorithm for Update Buffer of size 4. However, the slowdown experienced by the applications is very high for Update Buffer of size 4 and 8.

Based on these observations, we recommend the Greedy algorithm with $k = 32$, and an Update Buffer of four entries for shared as well as distributed caches, with two caches sharing an Interval Table in case of distributed caches.

VII. Conclusion

In this paper, we proposed two space sensitive techniques, bit-vector and Interval Table, to keep track of the cache lines that are updated after the previous transfer of L2 cache contents off-chip, during postsilicon validation. One major feature of the proposed DFD hardware, called Update Tracker, 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 11.5% as compared with over 10% area overhead of a simple bit-vector for a shared cache. We proposed a scheme to efficiently share the Update Tracker across multiple caches in order to further reduce the area overhead of storing the update information in distributed caches. We restrict the area overhead of the Update Trackers to less than 1% of the cache area in the case of distributed caches too with an average dumping overhead of less than 2%.

Acknowledgment

The authors would like to thank S. Sen, R. Kalayappan, P. Kallurkar, and A. Isaac for their help at different stages of the project. They would also like to thank the anonymous reviewers for their valuable comments on the manuscript.

References

[1] K.-H. Chang, I. L. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2007, pp. 91–98.

[2] Y.-S. Yang, A. Veneris, N. Nicolici, and M. Fujita, "Automated data analysis techniques for a modern silicon debug environment," in *Proc. 17th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan./Feb. 2012, pp. 298–303.

[3] T. Hong *et al.*, "QED: Quick error detection tests for effective post-silicon validation," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2010, pp. 1–10.

[4] A. DeOrio, D. S. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2011, pp. 755–761.

[5] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-silicon bug localization in processors using bug localization graphs," in *Proc. 47th ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2010, pp. 368–373.

[6] H. F. Ko and N. Nicolici, "Automated trace signals selection using the RTL descriptions," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2010, pp. 1–10.

[7] D. Chatterjee, C. McCarter, and V. Bertacco, "Simulation-based signal selection for state restoration in silicon debug," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2011, pp. 595–601.

[8] E. Anis and N. Nicolici, "On using lossless compression of debug data in embedded logic analysis," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2007, pp. 1–10.

[9] E. Anis Daoud and N. Nicolici, "On using lossy compression for repeatable experiments during silicon debug," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 937–950, Jul. 2011.

[10] P. R. Panda, M. Balakrishnan, and A. Vishnoi, "Compressing cache state for postsilicon processor debug," *IEEE Trans. Comput.*, vol. 60, no. 4, pp. 484–497, Apr. 2011.

[11] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Proc. Design, Autom. Test Eur. (DATE)*, Mar. 2008, pp. 1298–1303.

[12] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Apr. 2009, pp. 1338–1343.

[13] Q. Xu and X. Liu, "On signal tracing in post-silicon validation," in *Proc. 15th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2010, pp. 262–267.

[14] H. Shojaei and A. Davoodi, "Trace signal selection to enhance timing and logic visibility in post-silicon validation," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2010, pp. 168–172.

[15] K. Rahmani and P. Mishra, "Efficient signal selection using fine-grained combination of scan and trace buffers," in *Proc. 26th Int. Conf. VLSI Design 12th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2013, pp. 308–313.

[16] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. 43rd ACM/IEEE Design Autom. Conf.*, Jul. 2006, pp. 7–12.

[17] K. Goossens, B. Vermeulen, and A. B. Nejad, "A high-level debug environment for communication-centric debug," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Apr. 2009, pp. 202–207.

[18] H. F. Ko and N. Nicolici, "On automated trigger event generation in post-silicon validation," in *Proc. Design, Autom. Test Eur. (DATE)*, Mar. 2008, pp. 256–259.

[19] H. Yi, S. Park, and S. Kundu, "A design-for-debug (DfD) for NoC-based SoC debugging via NoC," in *Proc. 17th Asian Test Symp. (ATS)*, Nov. 2008, pp. 289–294.

[20] M. H. Neishaburi and Z. Zilic, "On a new mechanism of trigger generation for post-silicon debugging," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2330–2342, Sep. 2014.

[21] A. M. Gharehbaghi and M. Fujita, "Global transaction ordering in network-on-chips for post-silicon validation," in *Proc. 12th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2011, pp. 1–6.

[22] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2011, pp. 564–568.

[23] A. DeOrio, I. Wagner, and V. Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2009, pp. 405–416.

[24] S.-B. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proc. 45th ACM/IEEE Annu. Design Autom. Conf. (DAC)*, Jun. 2008, pp. 373–378.

[25] E. Anis Daoud and N. Nicolici, "Embedded debug architecture for bypassing blocking bugs during post-silicon validation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 4, pp. 559–570, Apr. 2011.

[26] L. Lee, L.-C. Wang, T. M. Mak, and K.-T. Cheng, "A path-based methodology for post-silicon timing validation," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2004, pp. 713–720.

[27] M. R. Kakoee, V. Bertacco, and L. Benini, "At-speed distributed functional testing to detect logic and delay faults in NoCs," *IEEE Trans. Comput.*, vol. 63, no. 3, pp. 703–717, Mar. 2014.

[28] P. Michaud, "Online compression of cache-filtered address traces," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2009, pp. 185–194.

[29] K. Basu and P. Mishra, "Efficient trace data compression using statically selected dictionary," in *Proc. IEEE 29th VLSI Test Symp. (VTS)*, May 2011, pp. 14–19.

[30] W. Li, A. Forin, and S. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. 47th ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2010, pp. 755–760.

[31] M. Gort *et al.*, "Formal-analysis-based trace computation for post-silicon debug," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 11, pp. 1997–2010, Nov. 2012.

[32] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco, "Machine learning-based anomaly detection for post-silicon bug diagnosis," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2013, pp. 491–496.

[33] A. Colantonio and R. D. Pietro, "Concise: Compressed 'n' composable integer set," *Inf. Process. Lett.*, vol. 110, no. 16, pp. 644–650, Jul. 2010.

[34] F. Delière and T. B. Pedersen, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps," in *Proc. 13th Int. Conf. Extending Database Technol. (EDBT)*, 2010, pp. 228–239.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[35] A. S. Fraenkel, S. T. Klein, Y. Choueka, and E. Segal, "Improved hierarchical bit-vector compression in document retrieval systems," in *Proc. 9th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr. (SIGIR)*, 1986, pp. 88–96.

[36] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," *ACM SIGMOD Rec.*, vol. 25, no. 2, pp. 103–114, Jun. 1996.

[37] A. Vishnoi, P. R. Panda, and M. Balakrishnan, "Online cache state dumping for processor debug," in *Proc. 46th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2009, pp. 358–363.

[38] A. Kumar, P. R. Panda, and S. Sarangi, "Efficient on-line algorithm for maintaining k-cover of sparse bit-strings," in *Proc. IARCS Annu. Conf. Found. Softw. Technol. Theoretical Comput. Sci. (FSTTCS)*, 2012, pp. 249–256.

[39] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A binary instrumentation tool for computer architecture research and education," in *Proc. WCAE*, 2004, p. 22.

[40] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, E. Peter, "Tejas: A Java based versatile micro-architectural simulator," in *PATMOS*, 2015.

Sandeep Chandran received the B.E. degree in computer science and engineering from Visveswaraya Technological University, Bangalore, India, in 2008.

He is currently a Research Scholar with the Department of Computer Science and Engineering, IIT Delhi, New Delhi, India. His current research interests include post-silicon validation methodologies and fault-tolerant systems.

Smruti R. Sarangi received the B.Tech. degree in computer science from IIT Kharagpur, Kharagpur, India, in 2002, and the M.S. and Ph.D. degrees in computer architecture from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2007.

He has spent four years with the industry with IBM India Research Labs, New Delhi, India, and Synopsys Inc., Mountain View, CA, USA. He is currently an Assistant Professor with the Department of Computer Science and Engineering, IIT Delhi, New Delhi. He is involved in the areas of computer architecture, parallel, and distributed systems.

Preeti Ranjan Panda (SM'09) received the B.Tech. degree in computer science and engineering from IIT Madras, Chennai, India, and the M.S. and Ph.D. degrees from the University of California at Irvine, Irvine, CA, USA.

He has been with Texas Instruments, Bangalore, India, and the Advanced Technology Group, Synopsys Inc., Mountain View, CA, USA. He has also been a Visiting Scholar with Stanford University, Stanford, CA, USA. He is currently a Professor with the Department of Computer Science and Engineering, IIT Delhi, New Delhi, India. He has authored two books entitled *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration* and *Power-Efficient System Design*. His current research interests include embedded systems design, CAD/VLSI, post-silicon debug/validation, system specification and synthesis, memory architectures and optimizations, hardware/software co-design, and low power design.

Prof. Panda was a recipient of the IBM Faculty Award in 2007, and the Department of Science and Technology Young Scientist Award in 2003. He has served on the Editorial Boards of the IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *ACM Transactions on Design Automation of Electronic Systems*, and the *International Journal of Parallel Programming*. He has served as the Technical Program Co-Chair of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) and the International Conference on VLSI Design and Embedded Systems.