Area-Aware Cache Update Trackers for Postsilicon Validation

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. The proposed architecture of this paper analysis the logic size, area and power consumption using Xilinx 14.2.

Enhancement of the project:

Existing System:

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;
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 look aside 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. 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.

Disadvantages:

- The area overhead accrues

Proposed System:

METHODOLOGY—SHARED CACHE

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.
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 non-updated 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.

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 3: 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.

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 densest 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

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.

METHODOLOGY—DISTRIBUTED CACHES

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.

**Horizontal Sharing**

Fig. 2 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. 2(b) hold the update information of both the caches (A and B) shown in Fig. 2(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.

**Vertical Sharing**

Fig. 3 shows an example of vertical sharing of Update Trackers (bit-vector and Interval Table) between two caches. The bits shaded in blue in Fig. 3(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 cache lines, we exploit the spatial locality of the references to a cache to limit the area overhead.

![Diagram](image)

**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 $\text{minLocalGap}$ and $\text{minGlobalGap}$ to be computed in a single pass.

**Memory Configuration**

An appropriate memory architecture needs to be selected to support the efficient computation of $\text{minLocalGap}$. 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).

Logic Design

Fig. 4 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.

Fig. 4. Hardware design of Greedy algorithm.

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.

Advantages:
The Master of IEEE Projects

- reduce the area overhead

Software implementation:

- Modelsim
- Xilinx ISE