# Reusing Trace Buffers as Victim Caches

Neetu Jindal, Preeti Ranjan Panda, Smruti R. Sarangi

Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India
E-mail: {neetu, panda, srsarangi}@cse.iitd.ac.in

***Abstract*—With the increasing complexity of modern Systems-on-Chip, the possibility of functional errors escaping design verification is growing. Post-silicon validation targets the discovery of these errors in early hardware prototypes. Due to limited visibility and observability, dedicated design-for-debug (DFD) hardware such as trace buffers are inserted to aid post-silicon validation. In spite of its benefit, such hardware incurs area overheads, which impose size limitations. However, the effective overhead could be reduced if the area dedicated to DFD could be reused in-field. In this work, we present a novel method for reusing an existing trace buffer as a victim cache of a processor to enhance performance. The trace buffer storage space is reused for the victim cache, with a small additional controller logic. Simultaneous multi-threading allows further fine-grained control of the victim cache, which can be shared between the threads based on the requirements of the applications. We also propose and evaluate different approaches to partition the victim cache between threads. Experimental results on several benchmark applications and trace buffer configurations show that the proposed approach can enhance the average performance by up to 8.3% with minimal area overhead.**

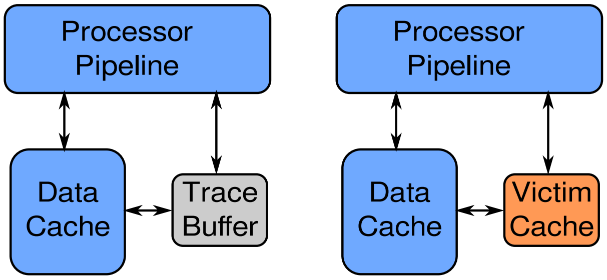
***Index Terms*—Design-for-Debug, Post-silicon Validation, Victim Cache.**

## I. Introduction

Decreasing feature sizes have caused ever increasing levels of on-chip component integration. Simulation or emulation used in pre-silicon validation can take a prohibitive amount of time to check for functional errors. During post-silicon validation, applications are executed on the chip prototype at native speeds and are analyzed using dedicated design-for-debug (DFD) hardware, enabling the discovery of functional bugs that may have slipped past pre-silicon validation [1], [2]. The DFD hardware is used to record the state history of important signals in the chip that could be critical in debugging the chip.

The design of the DFD structure represents a trade off between maximizing visibility and stringent area constraints, since it becomes vestigial once the chip is in production. The *trace buffer* is a typical DFD structure which consists of memory elements and records the values of signals deemed critical by the designer [3], [4]. These recorded signals can be extracted outside the chip and analyzed to determine the root cause of errors.

With increasing complexity and higher levels of integration of modules on a single chip, the area consumed by DFD structures increases significantly. This is a challenge for chip manufacturers because they have to strike a balance between competing goals; increasing visibility enables faster validation, but increases the area overhead of the DFD hardware, which becomes unusable when the chip goes into production (i.e., normal in-field operation).



(a) Debug Phase (b) In-Field

Fig. 1: High level illustration of the proposed approach. (a) Trace buffer used to store debug data. (b) Trace buffer reused as Victim Cache

We address this challenge by repurposing the DFD hardware through reusing the trace buffer as a victim cache of the processor. Reusing the DFD hardware allows the designers to provide more space to validation structures as it no longer constitutes wasted space. Victim cache is a backup buffer that is designed to handle the conflict misses of the data cache [5]. It stores the recently evicted lines of the data cache and is looked up when a miss in the data cache is encountered. A miss in the data cache that hits in the victim cache is addressed by swapping the contents of the data cache line and the matching victim cache line. Mapping decisions of the victim cache over the trace buffer are influenced by architectural parameters such as width of the trace buffer. The standard victim cache is a small fully-associative structure but we use a set-associative structure which is easier to adapt from the trace buffer. We propose to add a new victim cache controller which enables us to use the trace buffer as a victim cache to enhance performance.

The scenario described above is illustrated in Figure 1. During the post-silicon validation phase (Figure 1a), the DFD hardware is configured as a trace buffer. Following the validation phase, the DFD hardware is configured as a victim cache (Figure 1b). We outline the proposed modifications of the traditional trace buffer structure so that it can operate as a victim cache. In general, the trace buffer sizes could vary significantly, sometimes making it larger than conventional victim caches. The implementation of such non-standard victim cache designs admits the scope of partitioning the victim cache for Simultaneous Multithreading (SMT) cores. In Simultaneous Multithreading, instructions from more than one thread can be executed in parallel at any given stage of a pipeline [6]. Since the data of all the application threads running on the same core would be in the victim cache, the application thread with low data reuse may flush the contents of other threads, resulting in performance degradation. We propose two techniques to partition the victim cache among application threads so that the threads cooperate to make efficient use of victim cache. The first technique, multi-mode victim cache, allows the victim cache to operate in two modes: *shared* and *exclusive*. The second technique partitions the victim cache efficiently using a logistic regression based algorithm [7], [8].

The rest of this paper is structured as follows. Section II surveys related work on both validation hardware and cache partitioning. Section III discusses the methodology of reusing the trace buffer as victim cache. Section IV discusses the need of partitioning and the two techniques that are used for partitioning the victim cache. Section V describes the experiments and results. Finally, the conclusion and future work are described in Section VI.

## II. Related Work

Related research can be classified into two categories on the basis of the hardware validation structures: dedicated DFD hardware for validation, and reuse of existing architectural components for validation.

Significant research efforts have already been invested in the area of dedicated DFD hardware. Storing signal history inside the trace buffer helps in validating the chip as it provides visibility into the chip [3], [4], [9]. Only a selected set of signals can be stored in the on-chip trace buffers due to size limitations. Researchers have proposed techniques to identify the set of such critical signals [9]–[16]. The volume of execution traces stored in the trace buffer can be reduced significantly by determining when to start and stop the tracing by using event triggers and embedded logic analyzers. Novel methods have been proposed by researchers for defining the event triggers [17]–[19]. Chandran et al. [20] have proposed to store trace summaries along with recent execution traces to reduce the stalls caused by the limited size of trace buffers. Techniques have been proposed for reducing error detection latencies (Lin et al. [21]) and compressing the volume of stored data in the trace buffers [22], [23]. DeOrio et al. [24] propose a methodology called Bug Positioning System for locating inconsistent bugs arising out of the interaction of asynchronous clock domains or varying electrical and environmental conditions. The system has two components: a hardware structure to log the activity signals and a post-analysis software algorithm to identify the time and location of the bug.

Alternatively, several proposals aim to reuse existing architectural components to store traces, instead of using dedicated hardware. DeOrio et al. [25] aimed at validating memory consistency and coherence by storing activity logs in L1 and L2 caches to observe memory operations during program execution. Along these lines, Abdel-Khalek and Bertacco [26] have suggested validating the NoC interconnect by periodically taking snapshots of the packets in flight and storing those traces in node-specific L2 caches. Lai et al. [27] used the data cache to store traces together with cache data during validation. They configured some of the cache ways to store trace data which includes bus traces, performance traces, and processor traces, and used the write-back circuitry to dump out the trace contents. Mammo et al. [28] also used L1 data caches to store activity logs for validating memory consistency in multiprocessor systems. These logs are aggregated to main memory and can be further analyzed either on the processor under verification or on another processor/host to detect any memory consistency violations. An interesting use of the instruction cache is proposed by Lai et al. [29]; they use it for compressing program execution traces instead of inserting additional hardware compressors. This line of work interferes in some way with the normal functionality of the memory system [30], as it can potentially hide some performance bugs.

Analogous to reusing architectural components for DFD, our work attempts to reuse a standard DFD structure as an architectural enhancement. The DFD hardware does not interfere with the chip functionality during validation and is used in-field to enhance performance. Such reuse is along the lines of work by Basak et al. [31] who proposed to repurpose the DFD hardware as security wrappers.

The victim cache idea was first introduced by Jouppi [5] as an auxiliary structure that is looked up when a data cache miss is encountered. Evicted lines of the data cache are placed in the victim cache. Bahar et al. [32] suggested parallel look-up in the victim and data cache for improved performance. In our proposed design, the victim cache is accessed in parallel to the data cache. Selective victim caching [33] uses a prediction technique that selectively places incoming blocks either in the direct-mapped L1 cache or the victim cache depending on the past usage history of these blocks.

Cache partitioning is another research area related to our proposal. Novel solutions have been proposed by researchers for exclusively partitioning the shared last level cache among different processor cores for performance and power improvements [34]–[38]. This is conceptually related to our proposal of partitioning the victim cache, but these techniques cannot be used directly at the victim cache because the latter is a much smaller structure that needs a lightweight solution. In this work, we propose to partition the victim cache for a multi-threaded SMT core and also present a comparison with the application of standard cache partitioning.

## III. Reusing the Trace Buffer

Our architectural proposal is to reuse the trace buffer when the processor-based system is in field, so that the area dedicated towards the DFD structure is reclaimed and reused to enhance the functionality.

### A. Trace Buffer as Victim cache

We outline here the architectural changes that enable the reuse of the trace buffer as a victim cache (VCache). These include the addition of a victim cache controller logic to improve performance. The DFD hardware can be configured to be used as either a trace buffer during validation, or as a victim cache during normal operation.

*1) Baseline architecture:* We use a LEON3 SPARC-based CPU [39] as the baseline architecture. The standard design includes a debug infrastructure in the form of a distributed set of trace buffers organized as queues. The trace buffer in a processor core is used to store the pipeline trace data. The trace buffer controller (TB Ctlr) is used to control the data stored in the trace buffer and receive control instructions as well as timestamp information from a central Debug Support Unit (DSU).

*2) Proposed architecture:* The memory space dedicated to the trace buffer is proposed to be reused as victim cache storage. The changes made in the modified processor architecture are highlighted in blue in Figure 2. A new component *victim cache controller* is added to the architecture to support the new functionality. Unlike the standard victim cache [5] which is architected as a small fully-associative structure, we use a set-associative structure, which is easier to adapt from the trace buffer with minimal changes. However, we impose no size limit on the victim cache size, which can be derived from the trace buffer size. To configure the DFD hardware as either a trace buffer or a victim cache, a new control signal *vc_en*, sent by the central DSU is added, as shown in Figure 3. The victim cache controller is connected to the data cache (DCache) and the trace buffer controller, but not to the main pipeline.

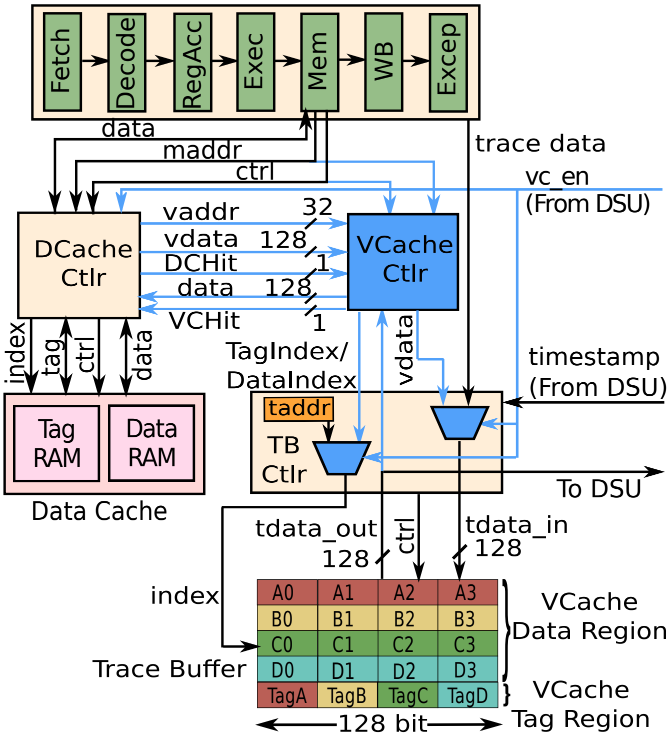


Fig. 2: Modified LEON3 architecture

**Data Storage and Address Mapping**: The default trace buffer is a monolithic single-port memory structure. To reuse it as a victim cache, we divide the address space into *tag* and *data* regions, as in a data cache. In the data region, each line represents a cache line of the data cache and its corresponding tag is present in the tag region. Considering a trace buffer width of 128 bits (i.e., 4 words), one line in the tag region corresponds to 4 lines in the data region. For each request, the victim cache controller reads a tag line and compares all four tags simultaneously, as in a 4-way set associative cache.

When a memory request to address *maddress* arrives, the victim cache controller indexes the request in the tag region and the corresponding data line is fetched from the data region and passed to the data cache controller. The data cache controller updates its memory with this new value and returns the result to the pipeline. For a trace buffer of size $T$ Bytes, the victim cache mappings for the above configuration are determined as:
Size of Data region = $4T/5$ Bytes
Size of Tag region = $T/5$ Bytes
Width of Trace buffer = 16 Bytes
*TagIndex* $= (maddress \gg \log_2 16)\&(nsets - 1)$
*DataIndex* $= si + [TagIndex \times 4 + HitIndex]$
where $nsets = T/5 \times 1/16$, $si$ is the starting index of the data region, and *HitIndex* is one of $\{0, ..., 3\}$ depending on the matching tag within the tag line. The *TagIndex* and *DataIndex* are computed as shown in Figure 3.

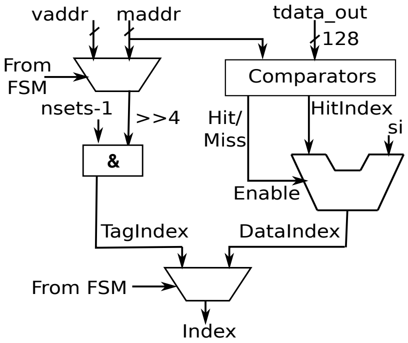


Fig. 3: The index computation logic in the victim cache controller

**Victim Cache Controller**: Figure 4 depicts the finite state machine of the victim cache controller for read and write requests. When a core initiates a memory access request, the VCache controller transitions from the *Idle* state to the *Read Tag* state. In that state, it checks the hit/miss status in the trace buffer and notifies the data cache controller. Simultaneously, the DCache controller also notifies its status to the VCache controller. There are three possible scenarios:

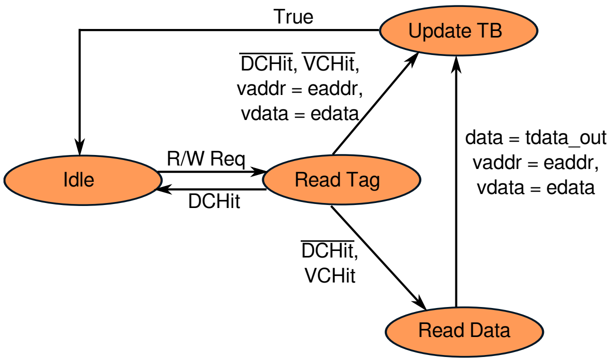


Fig. 4: FSM of the victim cache controller. DC: data cache, VC: victim cache

1) *Data cache hit and victim cache miss (DCHit = 1 and VCHit = 0)*: The data cache controller processes the request and the victim cache controller transitions to *Idle* state without performing any action.
2) *Data cache miss and victim cache hit (DCHit = 0 and VCHit = 1)*: The VCache controller transitions to the *Read Data* state and reads the required line from the data region. Then it transitions to the *Update TB* state, where the two controllers exchange their data and the VCache controller updates its content with the line evicted from the data cache. The data cache controller passes the data to the pipeline for the read request and updates it in memory for the write request. For our experiments, we have used write-through data caches with write-allocate cache miss strategy.
3) *Data cache miss and victim cache miss (DCHit = 0 and VCHit = 0)*: The data cache fetches a line from the next level and evicts one line. The VCache controller updates the trace buffer with this evicted line.

We cannot have a hit in both caches simultaneously as the two are mutually exclusive. One limitation of this methodology is we cannot debug the victim cache together with that architecture simultaneously. However, if necessary, the victim cache contents can be extracted by switching to the validation phase.

## IV. Dynamic Victim Cache Partitioning

Modern processor architectures have multiple hardware threads running on the same core. The victim cache design proposed in Section III-A2 uses the least recently used (LRU) replacement policy to store the data in the shared victim cache based on the application requirements. However, since the victim cache would contain data for multiple application threads, there is the risk of an application with low data reuse flushing out the data of other applications, which may significantly degrade the performance of the overall system. We propose to partition the victim cache between the hardware threads, so that the threads cooperate to make efficient use of the victim cache.

### A. Motivation

We use the *bzip2* and *gamess* benchmarks to demonstrate an opportunity to improve performance by partitioning the victim cache. The results from Section V-B indicate that the performance of *bzip2* improved by less than 1% when the trace buffer was used as a victim cache. On the other hand, the performance improvement of *gamess* was close to 14%. Thus, *gamess* benefited from the victim cache much more than *bzip2*.

If *bzip2* and *gamess* ran concurrently, they would compete for victim cache space. Figure 5 shows the number of hits in the victim cache with respect to the number of insertions performed by each benchmark during the execution of 500 million instructions. Each point in the graph represents the number of hits and insertions during a block of 1 million dynamic instructions. We observe that *bzip2* inserts a large number of cache lines into the victim cache, but the number of hits is relatively small because most of the lines are not reused. On the other hand, the number of hits of *gamess* is higher and could be further improved by reducing the victim cache pollution caused by *bzip2*.

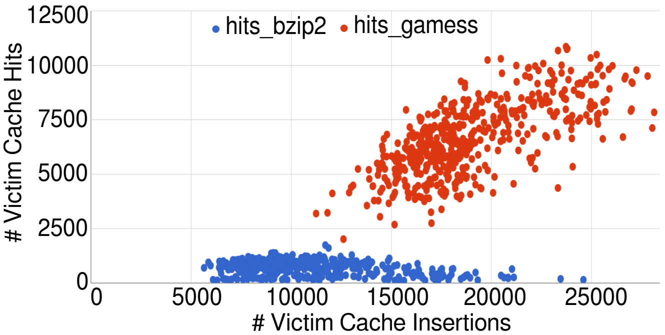


Fig. 5: Victim cache performance when *bzip2* and *gamess* are run concurrently

### B. High-Level Architecture

Our initial target architecture includes two hardware threads per core, which is commonly found in many existing architectures such as the AMD Zen microarchitecture and Intel Itanium Tukwila. We generalize the technique for multiple threads per core in Section IV-D4. Figure 6 depicts the high level architecture where the partitioning controller is implemented at the Debug Support Unit (DSU). The DSU uses statistics maintained by the data cache controller and victim cache controller to adaptively partition the victim cache. The DSU contains a trace buffer that is usually used during the validation phase. However, we reuse it to temporarily store some statistics needed to guide the partitioning. The components highlighted in blue are the new components added to enable the partitioning and their implementation depends on the policies discussed in the following sections. The partitioning unit passes $w$ values, each log(n) bits wide, where $w$ represents the number of ways in the victim cache and n represents the number of hardware threads. These values specify which ways should be allocated for which thread.

When a line is evicted from the data cache, it is stored in the victim cache. The core (not shown Figure 6) accesses the data cache and the victim cache concurrently. If a cache line is found in the victim cache, it is moved to the data cache.

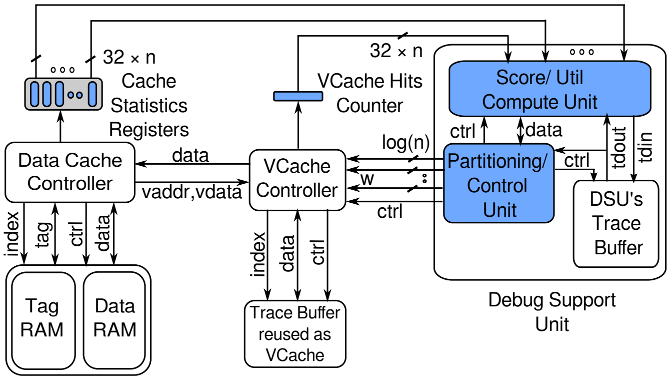


Fig. 6: High level architecture

### *C. Multi-Mode Victim Cache*

The victim cache is normally shared between the two threads and data of one thread can evict that of either thread, depending on the LRU policy decision. In this *shared mode*, a thread might insert excessive cache lines into the victim cache, potentially evicting useful data of the other thread. This may significantly degrade the latter's performance even while not necessarily improving its own performance. To alleviate this concern, we introduce the *exclusive mode*, in which the contents of the victim cache may be modified by only one thread, while permitting read access to both threads.

During execution, the partitioning controller uses collected statistics to determine whether the victim cache should operate in shared mode or exclusive mode. To dynamically make this decision, we use the number of victim cache hits and insertions as follows.

Execution is divided into sequences of dynamic instructions called *blocks*. For block $j$, the victim cache utilization for each thread is defined as follows:

$$VUtil_j^i = \frac{VHits_j^i}{VInsert_j^i} \tag{1}$$

where $VHits_j^i$ represents the number of victim cache hits and $VInsert_j^i$ represents the number of insertions performed by thread $i$ in the victim cache.

Since the memory access behavior of a hardware thread may change over time, the partitioning controller monitors the behavior of each hardware thread separately and keeps track of the utilization metric to select the mode that leads to the highest performance improvement.

Initially, the victim cache begins operation in the shared mode. Figure 7 shows how the DSU uses the victim cache utilization metric to switch between the shared and exclusive modes. The process shown in that figure is performed at the end of each block $j$. It takes as input the parameters required to compute the victim cache utilization and the current window and block number. Figure 8 illustrates an example scenario. If the victim cache is in shared mode, the DSU keeps track of the number of blocks in which each thread utilized the cache better by incrementing a counter associated with that thread. After a predetermined number of blocks, called the *window size* (*WSize*), it determines whether a switch to exclusive mode is required. If one thread utilized the victim cache significantly, the DSU sets the victim cache in exclusive mode so that only that thread inserts into the victim cache. The other thread would still have read access but cannot insert lines into the victim cache. This is determined using a fixed fraction $F$ with values in the range of $0.6 - 1.0$ to make sure that the condition is true for only one thread. Both $F$ and *WSize* are experimentally determined. Before switching to exclusive mode, the threshold (*th*) is computed, which is the average utilization of the selected thread (*sel*) that got the exclusive access. This threshold is used to determine when to switch back to the shared mode. Average Utilization ($AvgUtil_k^i$) of thread $i$ for the $k^{th}$ window is defined as follows:

$$AvgUtil_k^i = \frac{\sum_{j=0}^{WSize-1} VUtil_j^i}{WSize} \tag{2}$$

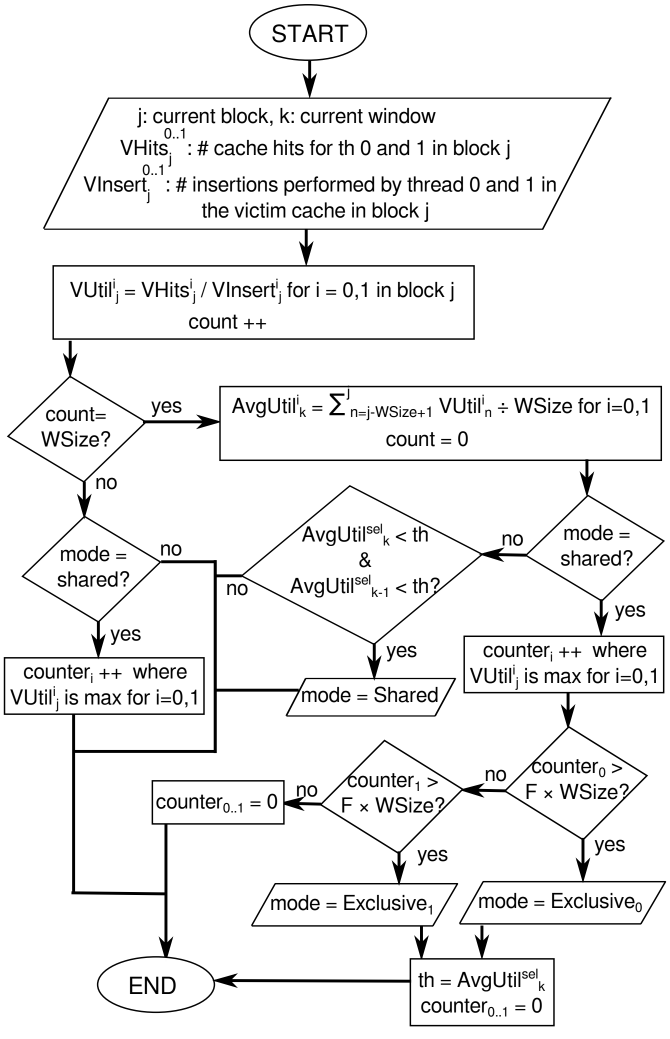

START
j: current block, k: current window
$VHits_j^{0..1}$ : # cache hits for th 0 and 1 in block j
$VInsert_j^{0..1}$ : # insertions performed by thread 0 and 1 in the victim cache in block j
$VUtil^i_j = VHits^i_j / VInsert^i_j$ for i = 0,1 in block j
count ++
count= WSize?
yes
$AvgUtil^i_k = \sum^j_{n=j-WSize+1} VUtil^i_n \div WSize$ for i=0,1
count = 0
no
mode = shared?
no
$AvgUtil^{sel}_k < th$ & $AvgUtil^{sel}_{k-1} < th$?
no
no
mode = shared?
yes
yes
counter$_i$ ++ where $VUtil^i_j$ is max for i=0,1
yes
mode = Shared
counter$_i$ ++ where $VUtil^i_j$ is max for i=0,1
counter$_{0..1}$ = 0
no
counter$_1$ > F × WSize?
no
counter$_0$ > F × WSize?
yes
yes
mode = Exclusive$_1$
mode = Exclusive$_0$
END
th = $AvgUtil^{sel}_k$
counter$_{0..1}$ = 0

Fig. 7: The victim cache mode selection process

where $VUtil_j^i$ represents the victim cache utilization value of thread $i$ for block $j$.

If the victim cache is in exclusive mode, the DSU maintains the average utilization of the selected thread for two consecutive windows. If the average utilization drops below *th* for two consecutive windows, the victim cache is switched to shared mode.

### *D. Victim Cache Partitioning*

A simple multi-mode victim cache policy would alleviate victim cache pollution caused by two hardware threads. However, a hardware thread may utilize the victim cache better than other threads, without necessarily requiring all of the victim cache. Exclusive access might degrade the performance of the remaining threads. In this section, we propose a technique to partition the cache proportionally with respect to utilization.

Figure 9 shows an illustrative example of victim cache partitioning for a 2-thread configuration. Initially, each hardware thread has been allocated half of the victim cache. At time $t$, the DSU determines that *thread*$_1$ is utilizing the victim cache better than *thread*$_0$, and it transfers a way from *thread*$_0$ to *thread*$_1$. Later, at time $t + \$$, the memory access behavior of

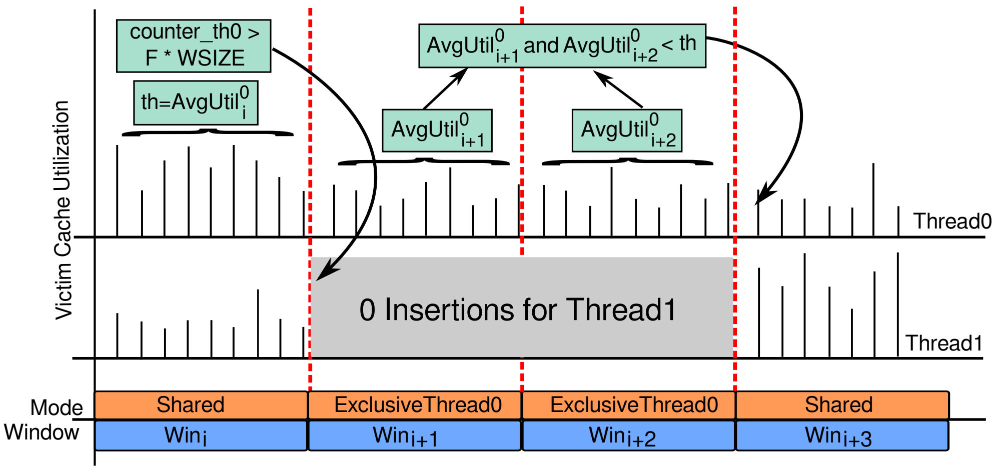


Fig. 8: Multi-mode Victim Cache

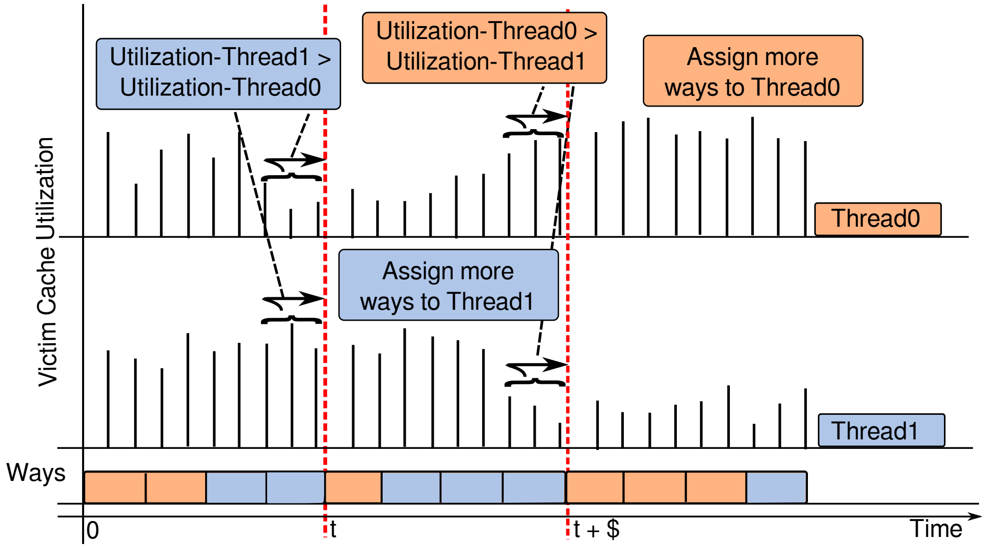


Fig. 9: Victim Cache Partitioning

the two threads changes, as a result of which the DSU gives more cache ways to $thread_0$.

Table 1(a) shows the metrics that the DSU uses to decide how many cache ways to give to each thread. The metrics are computed for each thread using:

1) Number of victim cache hits
2) Number of load misses
3) Number of store misses
4) Total number of misses
5) Total number of hits
6) Number of victim cache insertions

We assume that each thread maintains these counters in hardware performance registers and the DSU can access these registers. Such performance counters are typically present in modern architectures.

*1) Victim Cache Partitioning Technique:* We use a machine learning technique, *multiclass logistic regression* [7], [8], [40], to determine how much of the victim cache to allocate to each thread. Multiclass logistic regression takes as input a set of training instances of the form $\{(vh, lm, sm, cm, ch, insert), W_{ij}\}$ where $W_{ij}$ is the class label and $i$ represents the number of ways assigned to thread 0 and $j = 4 - i$. Table 1(b) shows the possible class labels and the corresponding partitioning for two hardware threads. We discuss the training set collection in Section IV-D2. The outputs are a set of 6 weights and an intercept for each class, which are stored in the trace buffer of the DSU. The weights are learned offline using the *liblinear* library [41]. At run-time, the DSU uses the following two equations to determine how to partition the cache:

$$\begin{aligned} logit(W_{ij}) = \alpha_0 + \alpha_1 vh + \alpha_2 lm + \alpha_3 sm + \\ + \alpha_4 cm + \alpha_5 ch + \alpha_6 insert \end{aligned} \tag{3}$$

$$P(W_{ij}) = \frac{1}{1 + e^{-logit(W_{ij})}} \tag{4}$$

where $\alpha_0$ is the intercept and $\alpha_1$ to $\alpha_6$ are the weights for class $W_{ij}$.

The DSU chooses the partitioning that corresponds to the class with the highest probability $P$. The computation of the probability could be expensive because of exponentiation operation; however, the exact probability need not be computed. Since the probability varies monotonically with $logit(W_{ij})$,

| Metric | Description |
|---|---|
| $vh$ | Ratio of $VHitsTh_0$ to $VHitsTh_1$ |
| $lm$ | Ratio of $LoadMissesTh_0$ to $LoadMissesTh_1$ |
| $sm$ | Ratio of $StoreMissesTh_0$ to $StoreMissesTh_1$ |
| $cm$ | Ratio of $CacheMissTh_0$ to $CacheMissTh_1$ |
| $ch$ | Ratio of $CacheHitsTh_0$ to $CacheHitsTh_1$ |
| $insert$ | Ratio of $InsertionsToVCacheTh_0$ to $InsertionsToVCacheTh_1$ |

(a) Features Description

| Class label | Number of ways | |
|---|---|---|
| | Thread0 | Thread1 |
| $W_{04}$ | 0 | 4 |
| $W_{13}$ | 1 | 3 |
| $W_{22}$ | 2 | 2 |
| $W_{31}$ | 3 | 1 |
| $W_{40}$ | 4 | 0 |

(b) Class labels and the corresponding partitioning

TABLE I: Description of the features and the class labels

we can choose the class with the highest $logit(W_{ij})$ instead.

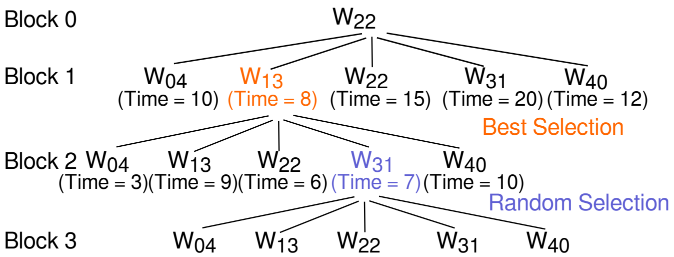


Fig. 10: Selection of class labels during training: alternating best and random selection

*2) Data Collection Strategy for Training:* We need to construct a training set that correlates the metrics shown in Table 1(a) with one of the partitioning classes shown in Table 1(b). To do this, we ran benchmark pairs for 1000 blocks. The victim cache begins with the partition that corresponds to class $W_{22}$. The construction of the training set is demonstrated in Figure 10. After one block, we can either continue in the same class or change to one of the other four classes ($W_{04}$, $W_{13}$, $W_{31}$, and $W_{40}$). To construct a representative training set, we need to exercise all the different possible class transitions. The number of possibilities grow exponentially, making it prohibitive to cover a large number of blocks exhaustively. To prune the search space, we could choose the class that yields a locally optimal performance. Therefore, class $W_{13}$ is selected as this class requires the minimum number of cycles to execute the instructions of Block 1. However, this process may not necessarily cover all the class transitions, which is necessary to train the system comprehensively. Therefore, class $W_{31}$ is selected at random after Block 1 instead of class $W_{04}$. In order to ensure a good coverage, after each block, we choose either the current best class or a random class, with a probability of 0.5 each. At the end of each block, the chosen class together with the measured features are added to the training set.

*3) The Partitioning Controller:* The partitioning controller operates according to the state machine shown in Figure 11. The initial state is S0. At the end of every block, the partitioning controller computes the *logit* function for each class. Then, it compares the number of ways assigned to thread 0 for the class label that corresponds to the highest *logit* value with the current class label's thread 0 way value. For example, if the current state is S0, and it turns out that the computed class label, say $W_{22}$, has more ways assigned to thread 0 than the current class label, say $W_{13}$, the partitioning controller transitions to state S2. At the end of the next block, if the computed class label is, say $W_{31}$, which also has more ways for thread 0 than the current class label (still $W_{13}$), the partitioning controller transitions to state S0 and changes the current class to $W_{22}$.

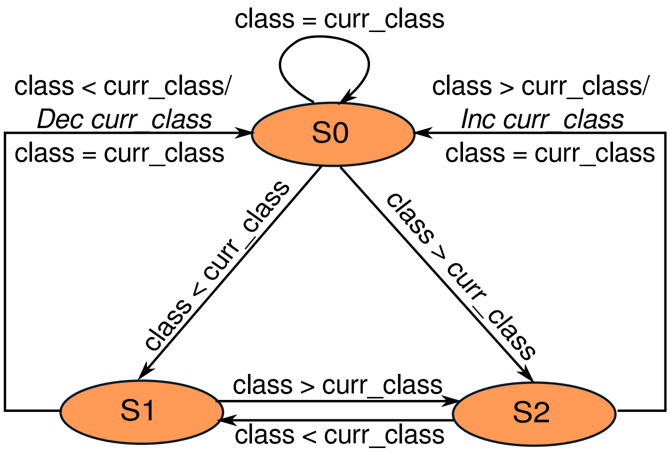


Fig. 11: Finite state machine for partition controller. class and curr_class refers to the number of ways assigned to thread0 for the corresponding class label

It takes at least two successive blocks (two state changes) to change the current partitioning. We observed experimentally that if the partitioning changed at the end of each block (one-stage prediction), the performance may not significantly improve and it might even degrade because the learned model might choose a class that is not suitable for the future behavior of the hardware threads. Having the current class change after two blocks introduces a hysteresis that alleviates the impact of such mispredictions. Similarly, the way-allocation changes the ways assigned to each thread by at most 1 at the end of every block for the same reason.

Computing the *logit* function requires the hardware necessary to perform division and multiplication. We use a simple circuit consisting of a single adder and a single shifter to perform sequential implementations of the multiplication and division operations in the partitioning controller. The computation time is, nevertheless, small in comparison to the block execution time.

*4) Generalization to multiple threads:* We discussed above the victim cache partitioning scheme in a system with two threads and four victim cache ways. A logistic regression classifier with five classes (Equation 3) was used to determine the partition value. The weights for those five classes were learnt offline and stored in the DSU's trace buffer.

$$logit(W^{ij}_{t,w-t}) = \alpha_0 + \alpha_1 \frac{vh^i}{vh^j} + \alpha_2 \frac{lm^i}{lm^j} + \alpha_3 \frac{sm^i}{sm^j} + \\ + \alpha_4 \frac{cm^i}{cm^j} + \alpha_5 \frac{ch^i}{ch^j} + \alpha_6 \frac{insert^i}{insert^j} \quad (5)$$

We generalize the above technique for any number of threads and any associativity of the victim cache. However, partitioning is still performed by considering two threads at a time. To partition a $w$-way victim cache among two threads, the total number of possible partitions would be $w + 1$ and a classifier with $w + 1$ classes is sufficient to determine the number of ways assigned to each thread. The DSU computes the logit value using Equation 5 for all the possible $w + 1$ classes and for every pair $(i, j)$ of threads. In Equation 5, $t$ and $w - t$ are the number of ways assigned to thread $i$ and $j$, respectively, $\alpha_0$ is the intercept, and $\alpha_1$ to $\alpha_6$ are the weights learnt for class $W_{t,w-t}$. The total number of logit values computed is ${}^{n}C_2 \times (w+1)$. A class is defined only by the partitioning, not by the pair of threads under consideration. Therefore, the weights are the same for all pairs of threads as long as the partitioning is the same. On the other hand, the values of the features (vh, lm, sm, ch, cm, insert) change according to the thread behavior.

Figure 12 shows the computation of the current victim cache assignment for each thread. This process is carried out at the end of each block. $n$ denotes the number of threads and $w$ denotes the number of ways. At each iteration of *Loop 1*, two threads are selected and the corresponding logit values are computed. An assignment of all the ways of the cache on the two threads is chosen according to the class that corresponds to the maximum logit value. When the loop ends, each $P[i]$ contains the relative demand of the corresponding thread. *Loop 2* compares the current partitioning with the new thread requirements and takes a decision of either increment (1), decrement (-1) or no change (0) for each thread. Let us consider an example, shown in Table II, with 4 threads ($th0$, $th1$, $th2$, $th3$) and an 8-way set associative victim cache. The example starts with the current assignment of two ways to each thread. In the first iteration, where $th0$ is paired with $th1$, the logit value for all the 9 classes are computed, and the maximum logit value is for class $W_{6,2}$. Therefore, $P[0]$ and $P[1]$ are updated to 6 and 2, respectively. Similarly, all $P[i]$ values are updated for all the possible six pairs of threads in the following iterations.

The DSU performs the partitioning recommended by the output of that process if it can find a (Inc, Dec) pair of two threads. Otherwise, the partitioning does not change. Here also, the DSU uses a 2-stage prediction technique similar to the one discussed in Section D.3.

*5) Replacement policy:* We have used a variant of the LRU replacement policy in the victim cache controller. The LRU indices in the victim cache are updated only at the time of insertion, unlike the data cache where the LRU indices are updated at the time of insertion as well as upon cache hit.

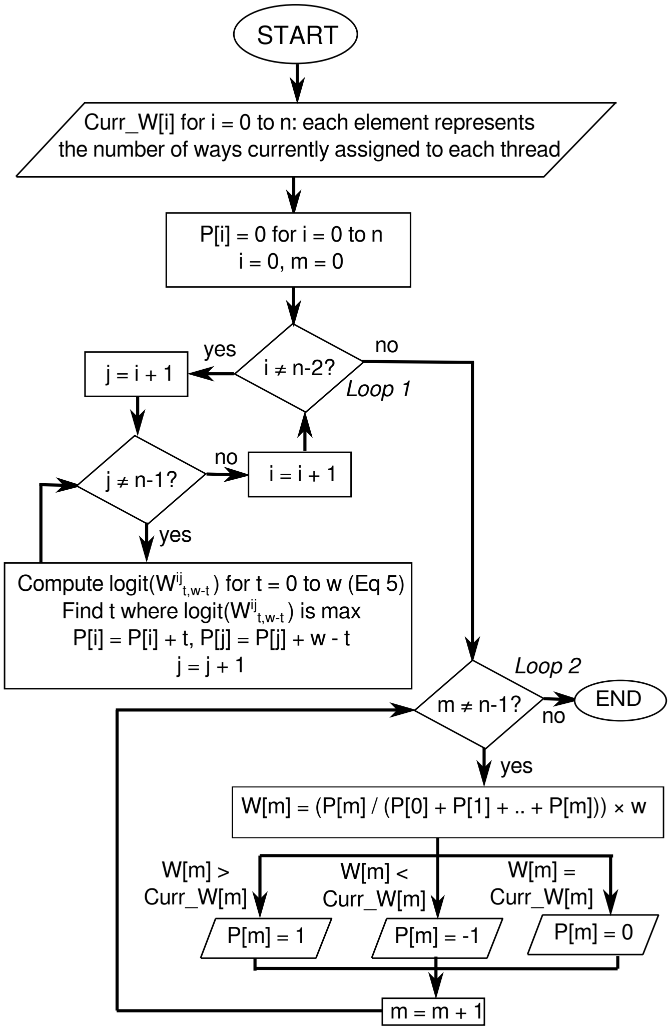

START
Curr_W[i] for i = 0 to n: each element represents
the number of ways currently assigned to each thread
P[i] = 0 for i = 0 to n
i = 0, m = 0
i ≠ n-2?
yes
no
Loop 1
j = i + 1
j ≠ n-1?
no
i = i + 1
yes
Compute logit(W^ij_t,w-t) for t = 0 to w (Eq 5)
Find t where logit(W^ij_t,w-t) is max
P[i] = P[i] + t, P[j] = P[j] + w - t
j = j + 1
Loop 2
m ≠ n-1?
no
END
yes
W[m] = (P[m] / (P[0] + P[1] + .. + P[m])) × w
W[m] > Curr_W[m]
W[m] < Curr_W[m]
W[m] = Curr_W[m]
P[m] = 1
P[m] = -1
P[m] = 0
m = m + 1

Fig. 12: Computation of the new victim cache assignment

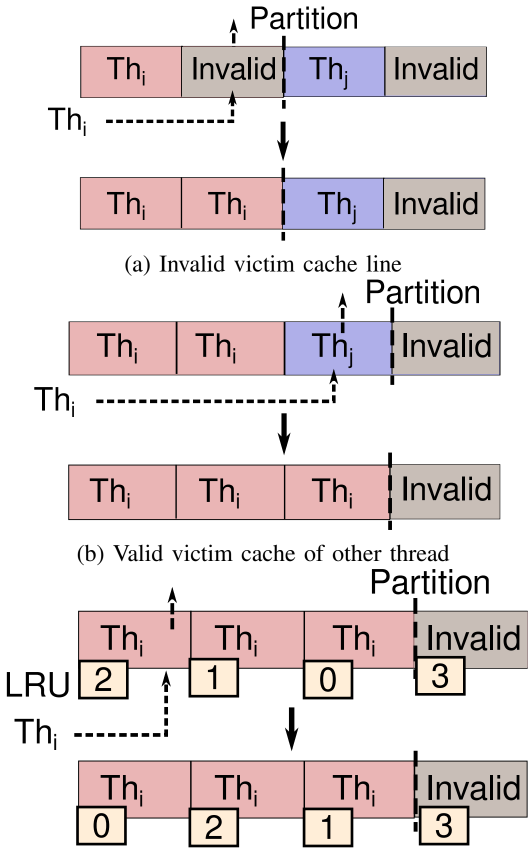
| Threads | th0 | th1 | th2 | th3 |
|---|---|---|---|---|
| CurrAssn | 2 | 2 | 2 | 2 |
| i=0, j=1 | 6 | 2 | | |
| i=0, j=2 | 4 | | 4 | |
| i=0, j=3 | 8 | | | 0 |
| i=1, j=2 | | 2 | 6 | |
| i=1, j=3 | | 2 | | 6 |
| i=2, j=3 | | | 8 | 0 |
| Sum | 18 | 6 | 18 | 6 |
| WaysReq | $\frac{18\times8}{48} = 3$ | $\frac{6\times8}{48} = 1$ | $\frac{18\times8}{48} = 3$ | $\frac{6\times8}{48} = 1$ |
| | Inc (1) | Dec (-1) | Inc (1) | Dec (-1) |

TABLE II: An example of executing Algorithm 2 for 4 threads and 8 ways

When there is a hit in the victim cache, the requested line is returned to the data cache and invalidated in the victim cache.

For any thread, the victim cache controller can search the data in all the $w$ ways, but can insert only in the thread's partition.

One simple way is to use a separate LRU indexing domain for each partition, which results in a timing overhead whenever the partition changes for the latency-critical victim cache. To overcome this, we have used a single indexing domain as was



(a) Invalid victim cache line

(b) Valid victim cache of other thread

(c) Least recently used victim cache line

Fig. 13: Replacement Policy

maintained without any partitioning. For any insertion request by $thread_i$, there are three choices as shown in Figure 13:

1) If there is an invalid cache line in the partition of $thread_i$, the new cache line is inserted there (Figure 13(a)).
2) If there is a valid cache line that was inserted by the other thread at some point in the past when the corresponding cache way was owned by the other thread, the new line is inserted there (Figure 13(b)).
3) Otherwise, the least recently used line in the partition of $thread_i$ is replaced with the new cache line (Figure 13(c)).

Whenever a new cache line is inserted into the victim cache, all the LRU indices are updated accordingly.

## V. Experiments

### A. Setup

We implemented our victim cache design on the *LEON3* [39], a synthesizable VHDL model of a 32-bit SPARC V8 processor. The standard design consists of an instruction trace buffer on every core; the trace buffer is a circular queue of 128-bit width and a configurable size ranging from 1KB to 64KB. We synthesized our design using Cadence Encounter *RTL compiler*with a 90nm technology standard cell library to understand the area and timing costs of our proposal. As the large SPEC benchmarks could not be simulated with the detailed VHDL model, we modeled the hardware separately in the *Sniper* full system simulator [42] to study the performance effects. We varied the size of the L1 data cache across our experiments and used a 512KB L2 cache. We evaluated our proposed architecture using several SPEC 2006 benchmarks and for each benchmark, we used the Simpoint [43] tool to identify a representative dynamic instruction sequence of length one billion.

### B. Impact of Trace Buffer Reuse

We first evaluate the overall system performance improvements obtained by using the trace buffer as a victim cache compared to the base architecture. Figure 14 shows the speedup attained in sixteen benchmarks for different data cache sizes while fixing the trace buffer at 2.5KB and 5KB, a small size compared to the data cache, which highlights the usefulness of reusing the trace buffer as a victim cache. The relatively high speedup of 14% (Trace Buffer size = 2.5KB) and 17% (Trace Buffer size = 5KB) for *gamess* is due to the high fraction of memory operations performed by it. In the case of *povray* for 32KB cache, we observe that the fraction of data cache lines that are used after eviction is around 90%. These observations are independently corroborated by other studies [44]. In the case of *bzip2*, we observe that around 55% of the lines that are evicted from the data cache are unused, resulting in small performance improvements. We observe no significant improvements with *mcf* and *hmmer*, although both are cache intensive programs. The two applications exhibit a thrashing behavior, and more than 55% of the evicted lines are not reused. Therefore, they do not benefit from the victim cache. *lbm* and *gromacs* are compute intensive applications with 35% and 40% memory operations, respectively. We do not observe any improvement with *lbm* as more than 60% of the evictions are unused. However, we observe an improvement of 2.1% with gromacs because more than 80% of the evictions are useful.

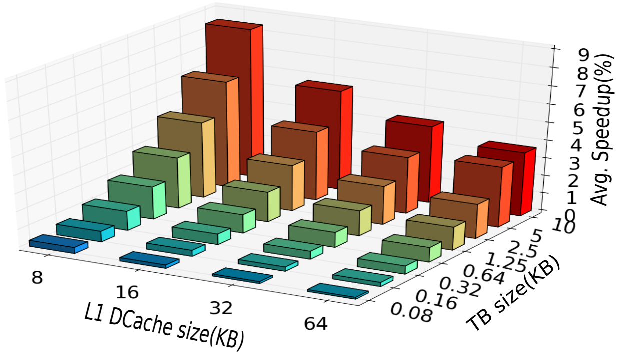
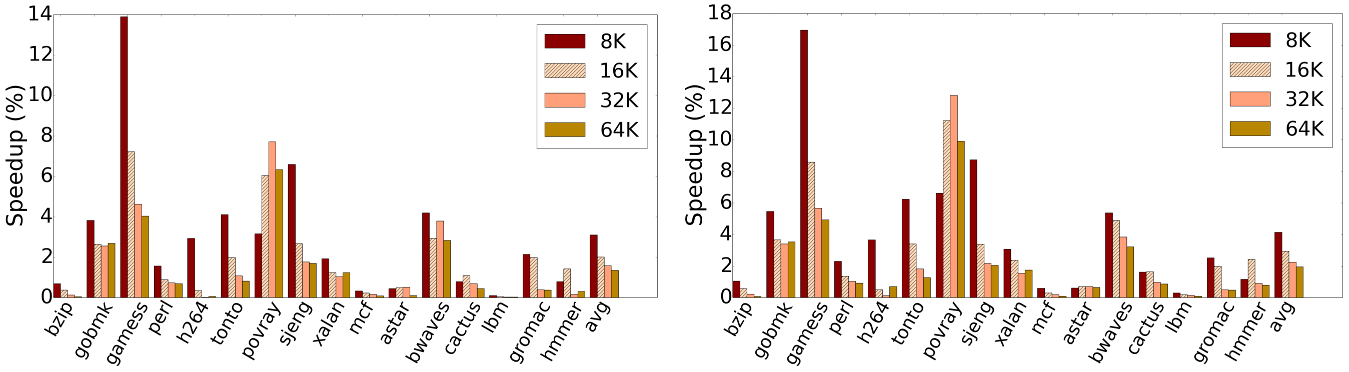


Fig. 15: Impact of varying trace buffer sizes

Figure 15 shows the performance improvements when varying both the trace buffer and data cache sizes, averaged over all the sixteen benchmarks. We observe that the victim cache is more effective for smaller caches as compared to larger caches. This is expected, as the small trace buffer translates to a higher relative size increment for smaller caches. However, even relatively small trace buffer sizes result in non-trivial



(a) Trace Buffer size = 2.5KB

(b) Trace Buffer size = 5KB

Fig. 14: Performance improvement for different data cache sizes

performance gains, which is significant considering that the DFD structure would have gone unutilized without this reuse.

We observe no significant improvement by increasing the associativity of the victim cache from 4 to 8 while keeping the overall size of the victim cache the same. Note that the access time of a 4-way set-associative victim cache is 5 cycles. To increase the associativity from 4 to 8, the trace buffer needs to be read twice as the two lines of the tag region are mapped to the same set, which increases the access time to 6 cycles.

### *C. Dynamic Victim Cache Partitioning*

We first evaluate the overall system performance improvement obtained by partitioning the victim cache between two hardware threads compared to the baseline architecture where the victim cache is in the shared mode. The two hardware threads run two different SPEC 2006 benchmarks, which are classified into three categories: compute-intensive, memory-intensive, and mixed. We consider 23 different combinations of benchmarks and different data cache sizes while keeping the size of the trace buffer fixed at 2.5KB. We have selected pairs of benchmarks from the same category and from different categories to cover different possible behaviors. The window size for multi-mode is 10 where each block is of 1 million instructions and the fraction used to compute the threshold is 0.7. Different sets of benchmark pairs (not included in the evaluation) were used for training data collection.

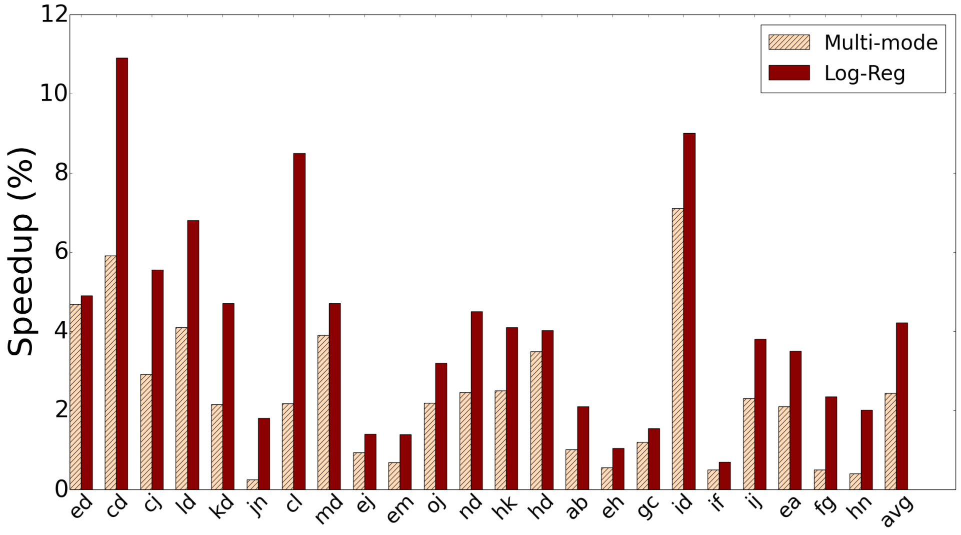
Figure 16 shows the comparison of the speedups attained using multi-mode and logistic regression for an 8KB data cache. Some benchmarks, such as *gamess*, *povray* and *tonto*, utilize the victim cache much better than others, such as *bzip2*, *perlbench* and *h264ref*. Therefore, when running *bzip2* and *gamess*, the partitioning controller detects that the utilization of *gamess* is substantially better than that of *bzip2*. Consequently, it exclusively provides the victim cache to *gamess* most of the time in both the techniques. This also explains the performance improvement for *hmmer* and *gamess*. *hmmer* exhibits the thrashing behavior and evicts the useful data of gamess benchmark, therefore the partitioning controller gives more preference to the *gamess* benchmark in both the techniques. However, when running *povray* and *gamess* together, the multi-mode victim cache gives exclusive access to one of the benchmarks at a time, even though both of them could benefit from the victim cache. In shared mode, the two benchmarks evict each other's data and neither makes efficient use of the victim cache. This is eliminated in the multi-mode victim cache, leading to a performance improvement. On the other hand, in victim cache partitioning, both benchmarks get to utilize part of the victim cache, resulting in a significantly higher speedup. This also explains the performance improvement for *povray* and *tonto*. We observe no significant improvement with *hmmer* and *lbm* as *lbm* is a compute-intensive application and does not make much use of the victim cache.

Figures 17 and 18 show the speedup attained by using the multi-mode victim cache and victim cache partitioning, respectively, for different data cache sizes. The average speedup due to multi-mode victim cache is 2.45% and the average speedup due to victim cache partitioning using logistic regression is 4.2% for an 8KB data cache. For larger data cache sizes, the victim cache becomes less beneficial because more data can fit in the data cache.

We observe that some of the benchmarks do not show significant improvement when run separately, however, mixes of these applications show significant improvements. For example, the *povray-tonto* mix shows a relatively high speedup of 8.2%. However, when the benchmarks run separately (Figure 14), they show a speed up of only 3% and 4%, respectively, for an 8KB data cache. The reason for this is, when *povray* and *tonto* run separately, there is no interference in the data cache and the victim cache is not very useful. Therefore, the victim cache is particularly useful for the mix scenario. This also explains why the gain is significant for the 8KB data cache and not for the 16KB data cache.

Figure 19 shows the speedup comparison due to two strategies: (i) one-stage prediction, and (ii) changing the partition at the end of two blocks as decided by the finite state machine in Figure 11. It is clear from the graph that the latter finite state machine prediction is always better than one-stage prediction. One-stage prediction even degrades the performance for some cases as the learned model might have chosen a wrong class which is not suitable for future behavior. Delaying the reconfiguration by waiting for an additional block alleviates the impact of such mispredictions.

To our knowledge, there are no cache partitioning techniques proposed for victim caches as they are traditionally



| Label | Benchmark |
|---|---|
| a | bwaves |
| b | cactus |
| c | povray |
| d | gamess |
| e | bzip2 |
| f | lbm |
| g | gromacs |
| h | xalan |
| i | hmmer |
| j | sjeng |
| k | gobmk |
| l | tonto |
| m | perlbench |
| n | h264ref |
| o | astar |

Fig. 16: Speedup comparison of multimode and Logistic Regression

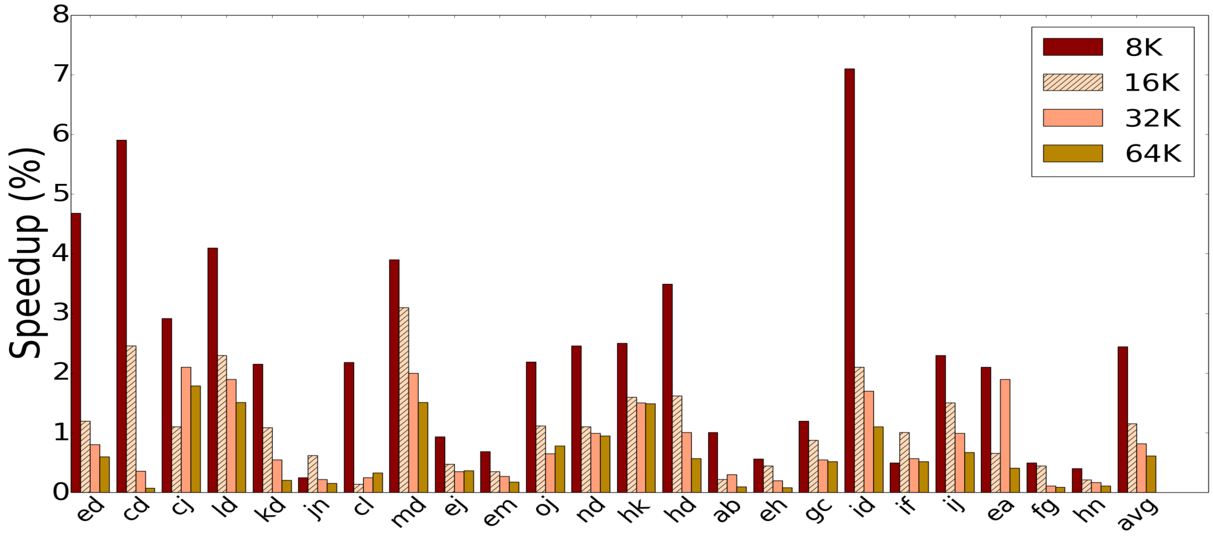


Fig. 17: Performance Improvement using Multimode victim cache partitioning

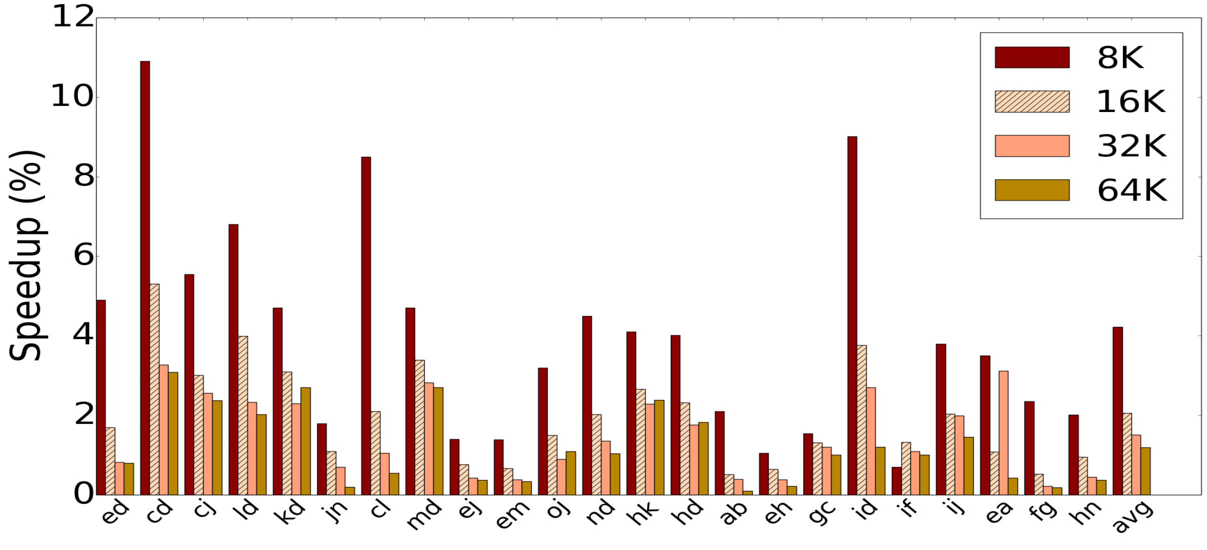


Fig. 18: Performance Improvement using Logistic Regression

fully-associative. We compare our results against the utility-based cache partitioning (UCP) technique [37]. The technique was originally proposed for the last level cache; however, we implemented it for the victim cache. Figure 20 shows the speedup comparison. The UCP technique provides at least one way to each benchmark and partitions the remaining ways according to the marginal utility value. However, in our proposed architecture, the victim cache is not placed on the main path and operates in parallel to the data cache. Therefore, it is not necessary to give at least one way to each thread. Our technique takes advantage of this opportunity by providing the complete victim cache to one of the threads during execution, resulting in significant performance improvements in many cases. For some of the benchmarks, UCP performs better because of its higher prediction accuracy. The UCP technique incurs a significant area overhead. We have also considered 32 sets to maintain the profiling information and 24 bits (3 bytes) to store the tag for the UCP implementation. Therefore the total area overhead to maintain the profiling information is $32 \times (3 \times w) \times n$ bytes, where $w$ is the number of ways and $n$ is the number of threads. For a system with 2 threads and 4-way set associative cache, the profiling area overhead of UCP is 768 bytes (4% of the total area). It is larger with more threads and higher associativity. In comparison, our proposed technique uses performance counters already present in processors, and reuses the DSU's trace buffer, leading to a smaller area footprint.

Figure 21 shows the performance improvement for 4 threads and an 8-way set associative victim cache, where the size of the trace buffer is 5KB. The relatively high speedup of 13% for the *hmmer-lbm-sjeng-gobmk* mix is due to the thrashing behavior of *hmmer*. In shared mode, it evicts a lot of useful data of *sjeng* and *gobmk*, which is avoided by the victim cache partitioning controller. This also explains the improvement for *hmmer-bwaves-povary-gamess*. The speedup for *hmmer-sjeng-gobmk-tonto* is less than *hmmer-lbm-sjeng-gobmk*. It is because of compute-intensive nature of lbm and the complete victim cache is available for *sjeng* and *gobmk*. However, in the other mix, *tonto* does require a significant share of the victim cache. All the four benchmarks in the *gamess-sjeng-gobmk-tonto* mix are memory-intensive and the corresponding performance improvement is 7.5% compared to the shared mode. We observe no significant improvement with *xalan-hmmer-h264ref-astar* because of the compute-intensive nature of the three applications and *hmmer* exhibits a thrashing behavior. The average speedup for 4 threads and an 8-way set associative victim cache is 3.97%, where the trace buffer size is 2.5KB. The results are averaged over the fifteen benchmark mixes considered in Figure 21.

Victim cache partitioning always improves performance to a greater extent than the multi-mode victim cache. However, it has a higher associated area overhead, leading to an area-performance trade-off between the two techniques. The preferred technique may depend on the space constraints on the system. We also implemented and evaluated partitioning the L1 data cache alone and simultaneously with the victim cache; this did not improve over the results delivered of our proposal and the overheads were higher. The possibility of improving performance through a tighter co-ordination of the two partitionings is a topic of future research.

### D. Synthesis Results

The area and power overheads of the individual controllers are shown in Table III. Traditionally, the trace buffers are power gated, however, the infield-reuse of trace buffer (size = 3KB) incurs power overhead of 81 mW. The enhancements related to the victim cache do not intersect with the critical path, and we do not observe any cycle time overhead.

## VI. Conclusion and Future Work

We proposed and evaluated an approach to reuse the trace buffer as a victim cache in order to enhance in-field performance. The result is a non-standard victim cache design which reuses the storage area of the trace buffer. We also proposed and evaluated our approaches to manage the victim cache in a 2-way SMT processor core. Multi-mode victim cache provides exclusive victim cache access to a hardware thread while victim cache partitioning provides the required share of the victim cache to each thread. Although victim cache partitioning always performs better than the multi-mode victim cache, its implementation requires additional hardware. In the future, we plan to reuse the trace buffer beyond the L1 cache in other levels of the memory hierarchy. For architectures that already have a victim cache, the proposed reuse could be applicable in the form of backup storage for the main victim cache.

## VII. Acknowledgment

This research was partially supported by a research grant from Freescale Semiconductor and Semiconductor Research Corporation. We are grateful for their support. We would also like to thank the anonymous reviewers for their valuable comments on the manuscript.

## References

[1] A. Adir, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "Leveraging pre-silicon verification resources for the post-silicon validation of the IBM POWER7 processor," in *DAC*. IEEE, 2011.

[2] A. Nahir, A. Ziv, R. Galivanche, A. Hu, M. Abramovici, A. Camilleri, B. Bentley, H. Foster, V. Bertacco, and S. Kapoor, "Bridging pre-silicon verification and post-silicon validation," in *DAC*. ACM, 2010.

[3] N. Nicolici and H. F. Ko, "Design-for-debug for post-silicon validation: Can high-level descriptions help?" in *HLDVT*. IEEE, 2009.

[4] S.-B. Park and S. Mitra, "Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *DAC*. ACM, 2008.

[5] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA*. IEEE, 1990.

[6] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[7] H.-F. Yu, F.-L. Huang, and C.-J. Lin, "Dual coordinate descent methods for logistic regression and maximum entropy models," *Machine Learning*, vol. 85, no. 1, pp. 41–75, 2011.

[8] C.-J. Lin, R. C. Weng, and S. S. Keerthi, "Trust region newton method for logistic regression," *Journal of Machine Learning Research*, vol. 9, no. Apr, pp. 627–650, 2008.

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

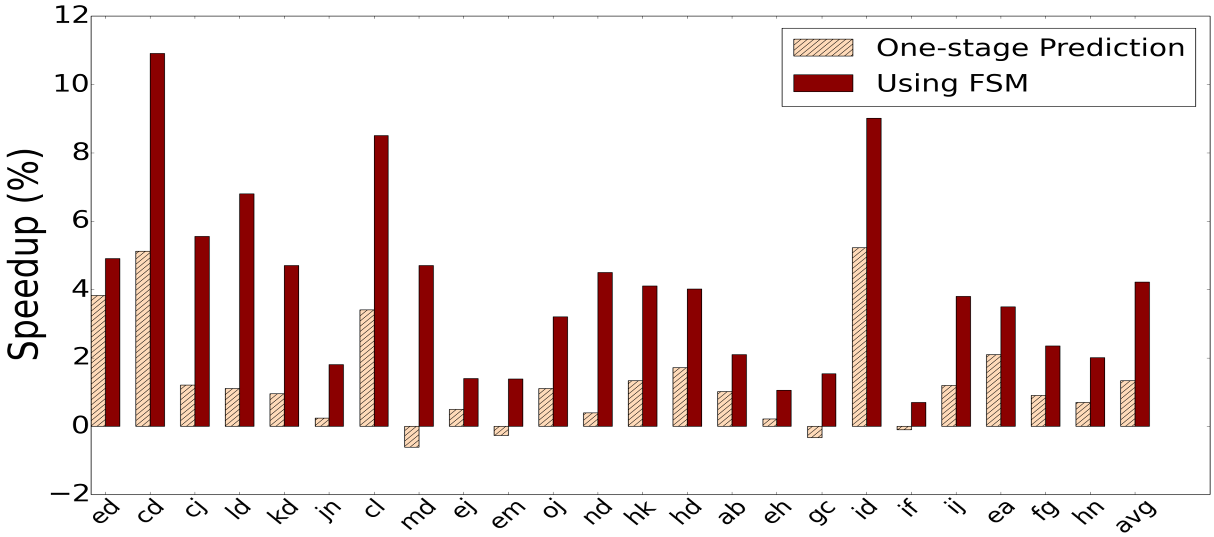


Fig. 19: Speedup comparison of partitioning using one-stage prediction and the FSM (Figure 11

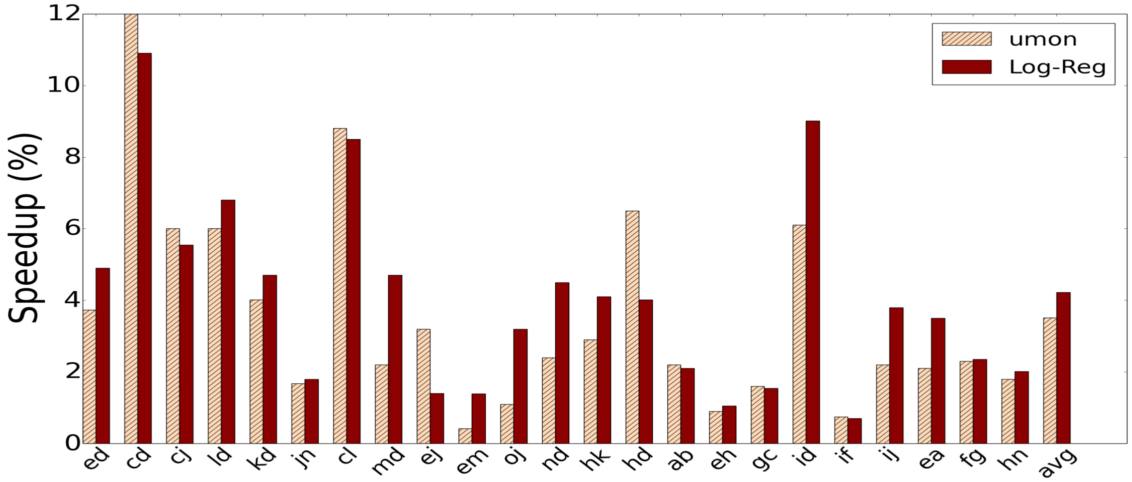
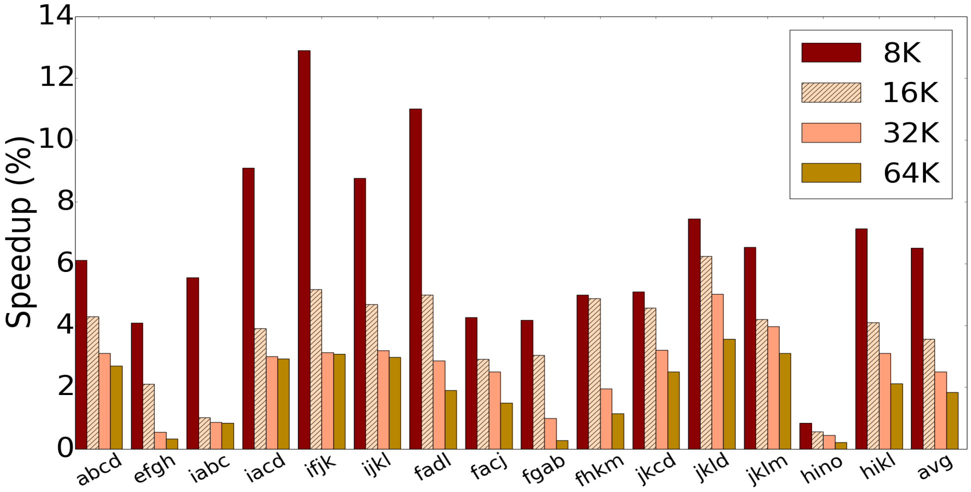


Fig. 20: Speedup comparison of partitioning using utility-based cache partitioning [37] and logistic regression)



| Label | Benchmark |
|---|---|
| a | bwaves |
| b | cactus |
| c | povray |
| d | gamess |
| e | bzip2 |
| f | lbm |
| g | gromacs |
| h | xalan |
| i | hmmer |
| j | sjeng |
| k | gobmk |
| l | tonto |
| m | perlbench |
| n | h264ref |
| o | astar |

Fig. 21: Performance Improvement using Logistic Regression for 4 threads and 8-way set associative victim cache

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

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

| Controller | Victim Cache (Figure 4) | Multimode-2th (Section IV) | Logistic Reg-2th (Section IV) | Logistic Reg (Only FSM in Figure 11) | Logistic Reg-4th (Section IV-D4) |
|---|---|---|---|---|---|
| Area ($mm^2$) | 0.0155 | 0.0033 | 0.0055 | 0.0008 | 0.011 |
| Power (mW) | 0.91 | 0.33 | 0.38 | 0.09 | 0.86 |

TABLE III: Area and power overhead for different controllers

[12] E. Larsson, B. Vermeulen, and K. Goossens, "A distributed architecture to check global properties for post-silicon debug," in *ETS*. IEEE, 2010.

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

[14] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan, "Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection," in *ICCAD*. IEEE Press, 2015.

[15] K. Rahmani, S. Proch, and P. Mishra, "Efficient selection of trace and scan signals for post-silicon debug," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 313–323, 2016.

[16] K. Rahmani, S. Ray, and P. Mishra, "Postsilicon trace signal selection using machine learning techniques," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 570–580, 2017.

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

[18] H. F. Ko and N. Nicolici, "Mapping trigger conditions onto trigger units during post-silicon validation and debugging," *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1563–1575, 2012.

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

[20] S. Chandran, P. R. Panda, S. R. Sarangi, A. Bhattacharyya, D. Chauhan, and S. Kumar, "Managing trace summaries to minimize stalls during postsilicon validation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 6, pp. 1881–1894, 2017.

[21] D. Lin, S. Eswaran, S. Kumar, E. Rentschler, and S. Mitra, "Quick error detection tests with fast runtimes for effective post-silicon validation and debug," in *DATE*. European Design and Automation Consortium, 2015.

[22] K. Basu and P. Mishra, "Efficient trace data compression using statically selected dictionary," in *VTS*. IEEE, 2011.

[23] A. Vishnoi, P. R. Panda, and M. Balakrishnan, "Cache aware compression for processor debug support," in *DATE*. European Design and Automation Association, 2009.

[24] A. DeOrio, D. S. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *ICCAD*. IEEE, 2011.

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

[26] R. Abdel-Khalek and V. Bertacco, "Post-silicon platform for the functional diagnosis and debug of networks-on-chip," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, p. 112, 2014.

[27] C.-H. Lai, Y.-C. Yang, and J. Huang, "A versatile data cache for trace buffer support," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 11, pp. 3145–3154, 2014.

[28] B. W. Mammo, V. Bertacco, A. DeOrio, and I. Wagner, "Post-silicon validation of multiprocessor memory consistency," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 1027–1037, 2015.

[29] C.-H. Lai, F.-C. Yang, and J. Huang, "A trace-capable instruction cache for cost-efficient real-time program trace compression in SoC," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1665–1677, 2011.

[30] Y. Chen, T. Chen, L. Li, R. Wu, D. Liu, and W. Hu, "Deterministic replay using global clock," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, p. 1, 2013.

[31] A. Basak, S. Bhunia, and S. Ray, "Exploiting design-for-debug for flexible soc security architecture," in *DAC*. IEEE, 2016.

[32] R. I. Bahar, G. Albera, and S. Manne, "Power and performance tradeoffs using various caching strategies," in *ISLPED*. IEEE, 1998.

[33] D. Stiliadis and A. Varma, "Selective victim caching: A method to improve the performance of direct-mapped caches," *IEEE transactions on Computers*, vol. 46, no. 5, pp. 603–610, 1997.

[34] C. Yu and P. Petrov, "Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms," in *DAC*. ACM, 2010.

[35] W. Wang, P. Mishra, and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *DAC*. ACM, 2011.

[36] S. Mittal, Y. Cao, and Z. Zhang, "Master: A multicore cache energy-saving technique using dynamic cache reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 8, pp. 1653–1665, 2014.

[37] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*. IEEE, 2006.

[38] R. Jain, P. R. Panda, and S. Subramoney, "A coordinated multi-agent reinforcement learning approach to multi-level cache co-partitioning," in *DATE*. IEEE, 2017.

[39] A. Gaisler, "Leon3 processor. http://www.gaisler.com."

[40] C. Bishop, "Pattern recognition and machine learning: springer new york," 2006.

[41] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *Journal of machine learning research*, vol. 9, no. Aug, pp. 1871–1874, 2008.

[42] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2011.

[43] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS*, 2003.

[44] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

**Neetu Jindal** is a research scholar in the Department of Computer Science and Engineering at Indian Institute of Technology, Delhi. She received her Bachelors' degree in Computer Science and Engineering from Kurukshetra University. Her research interests include post-silicon validation methodologies, architectural design-space exploration and machine learning applications to computer architecture optimizations.

**Preeti Ranjan Panda** received his B. Tech. degree in Computer Science and Engineering from the IIT Madras and his M. S. and Ph.D. degrees from the University of California at Irvine. He is currently a Professor in the Department of Computer Science and Engineering at IIT Delhi. He has previously worked at Texas Instruments and Synopsys, and has been a visiting scholar at Stanford University. His research interests are in the areas of Embedded Systems and Design Automation. He is the author of two books: *Memory issues in Embedded Systems-on-chip: Optimizations and Exploration* and *Power-efficient System Design*. He is a recipient of an IBM Faculty Award and a Department of Science and Technology Young Scientist Award. Prof. Panda has served on the editorial boards of IEEE TCAD, ACM TODAES, IEEE ESL, and IJPP, and as Technical Program co-Chair of CODES+ISSS and VLSI Design. He has also served on the technical program committees and chaired sessions at several conferences including DAC, ICCAD, DATE, CODES+ISSS, ISLPED, and EMSOFT.

**Smruti R. Sarangi** is an Assistant Professor in the Department of Computer Science and Engineering, IIT Delhi, India. He has spent four years in industry working in IBM India Research Labs, and Synopsys. He graduated with a M.S and Ph.D in computer architecture from the University of Illinois at Urbana-Champaign in 2007, and a B.Tech in computer science from IIT Kharagpur, India, in 2002. He works in the areas of computer architecture, parallel and distributed systems. Prof. Sarangi is a member of the IEEE and ACM.