Reusing Trace Buffers to Enhance Cache Performance

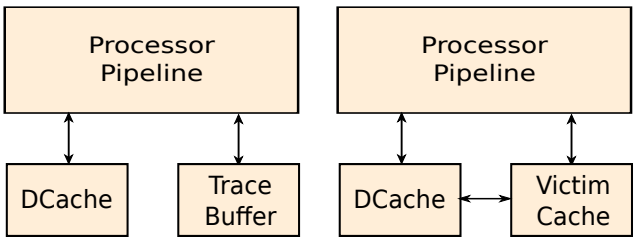
Neetu Jindal, Preeti Ranjan Panda, Smruti R. Sarangi
Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India
E-mail: {neetu, panda, srsarangi}@cse.iitd.ac.in

***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 overhead could be overcome 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. Experimental results on several benchmarks and trace buffer sizes show that the proposed approach can enhance the average performance by up to 8.3% over a baseline architecture. We also propose a strategy for dynamic power management of the structure, to enable saving energy with negligible impact on performance.**

I. Introduction

Decreasing feature sizes have caused ever increasing levels of on-chip component integration. The 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. 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 is non-trivial because it has to maximize the visibility while operating under very stringent area constraints, since it becomes vestigial once the chip is in production. The simplest DFD structure is the *trace buffer* which consists of memory elements and records the values of signals deemed critical by the designer. 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 the chip manufacturers because they have to strike a balance between the visibility and the area allocated to DFD structures. The two goals are competing; increasing visibility enables faster validation, but also increases the area overhead of the DFD hardware, which becomes unusable when the chip goes into production (i.e., normal operation in field). We address this challenge by repurposing the DFD hardware through reusing the trace buffer as a victim cache of the processor.



(a) Debug Phase (b) In-Field

Fig. 1: Trace Buffer reused as Victim Cache

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 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. Maintaining the victim cache active may enhance performance only during some phases of program execution. Depending on the application behavior, lookups in the victim cache may waste energy while not improving performance. To address this situation, we propose a dynamic power management technique to power gate the victim cache with minimal performance degradation.

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 amounts of research efforts have already been invested in the area of dedicated DFD hardware. Storing signal history helps in validating the chip as it provides visibility inside the chip. It is standard practice to maintain trace buffers inside the chip [1]–[3]. Only a selected set of signals can be stored in the trace buffer due to size limitations. Researchers have proposed techniques to identify the set of such critical signals [4]–[7]. Alternatively, several proposals aim to reuse existing architectural components to store traces, instead of using dedicated hardware. DeOrio et al. [8] 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, other researchers [9] 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. [10] 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. This line of work interferes in some way with the normal functionality of the memory system [11], 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. To the best of our knowledge, this is the first work proposing such reuse of validation hardware.

The victim cache idea was first introduced by Jouppi [12] as an auxiliary structure that is looked up when a data cache miss is encountered. All the evicted lines of the data cache are placed in the victim cache. 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. Bahar et al. [13] 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.

Another related research area is power optimization using performance counters. Gilberto et al. [14] suggested the use of performance counters such as IPC, data cache misses, data dependencies, and TLB misses to estimate run-time power consumption of CPU and memory. Chen et al. [15] proposed a DVFS algorithm to minimize energy. It uses a power model that utilises hardware performance counter values to adapt to application phase changes. These principles are used in our proposed optimization. In this work, we aim to save energy by the dynamic power management of the victim cache using performance counters.

III. Reusing the Trace Buffer

Our main 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: Figure 2 depicts the architecture of the LEON3 SPARC-based CPU. The standard design includes debug infrastructure in the form of a distributed set of trace buffers, organized as queues. The trace buffer in the processor core is used to store the pipeline trace data. The trace buffer controller 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).

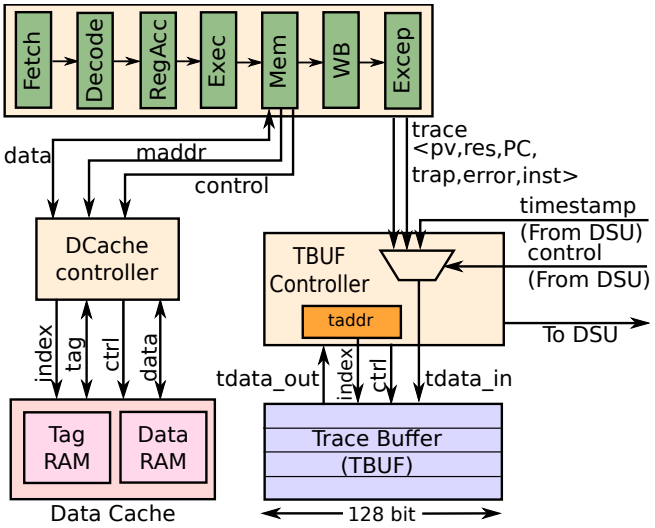


Fig. 2: Baseline architecture of LEON3

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 LEON3 architecture are highlighted in blue in Figure 3. A new component *victim cache controller* is added to the architecture to support the new functionality. Unlike the standard victim cache [12] which is architected as a small fully-associative structure, we use a set-associative structure, which is easier to adapt from the trace buffer. 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* is added, sent by the central DSU, 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.

Data Storage and Address Mapping: The default trace buffer is a monolithic single-port memory structure. To reuse it as a victim cache, we logically 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

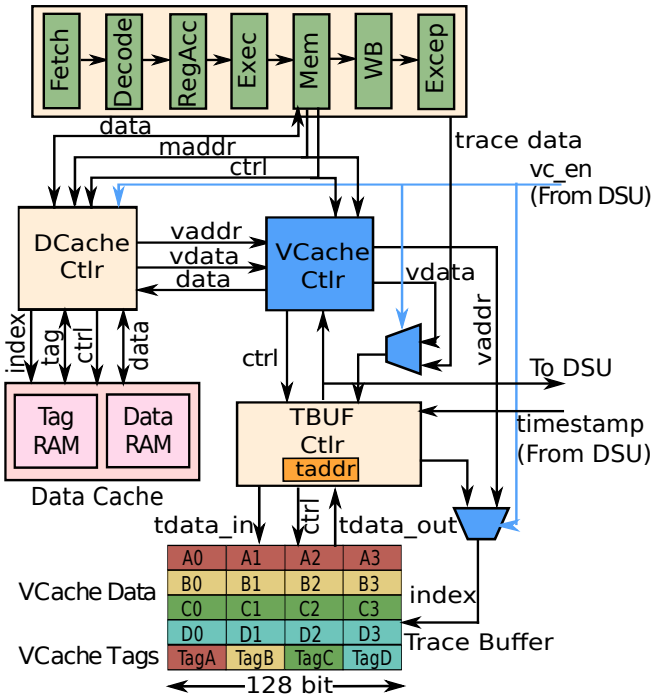


Fig. 3: Modified LEON3 architecture

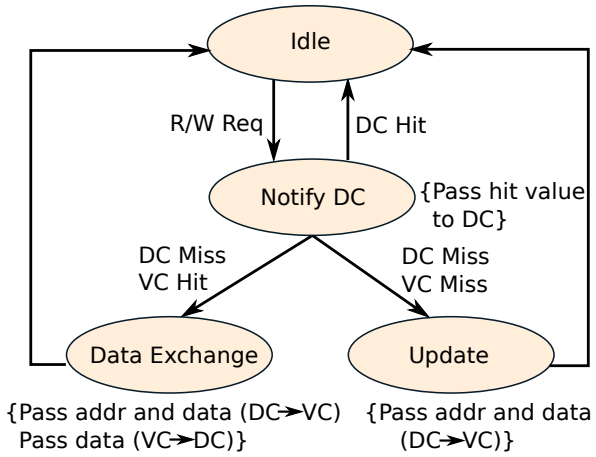


Fig. 4: FSM of the victim cache controller. DC: data cache, VC: victim cache

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.

Victim Cache Controller: Figure 4 depicts the finite state machine based implementation of the victim cache controller for read and write requests. When a core initiates a memory access request, the victim cache controller 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:

1) *Data cache hit and Victim cache miss*: The data cache controller processes the request and the victim cache controller returns to idle state without performing any action.
2) *Data cache miss and Victim cache hit*: The two controllers exchange their data and the victim cache 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*: The data cache fetches a line from the next level and evicts one line. The victim cache 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 the base architecture simultaneously. However the victim cache controller in our proposed design is similar to the standard victim cache controller and memory of the victim cache can be extracted by switching to the validation phase.

B. Power optimization of victim cache

Utilizing the trace buffer as a victim cache incurs power overheads due to the buffer lookups. Since the actual performance improvement using the victim cache varies with program behavior, this provides a possibility for significant power optimization. Figure 5 plots the variation in the number of victim cache hits while executing the *perlbench* benchmark. Each point of the graph represents the number of victim cache hits after executing 0.1 million instructions. We observe that there are some phases with a high victim cache hit rate, contributing to significant improvements in performance, while there are other phases where the hit rate is relatively low. The power consumed by the victim cache in the low-hit-rate phases represents an overhead without any significant performance benefit. This overhead can be reduced by detecting low-hit-rate phases at run-time and applying the power gating optimization on the victim cache. To achieve this, we maintain a victim cache *hit counter* and a *store misses counter* to control power-gating of the victim cache using the DSU. If the number of recent victim cache hits is below a certain threshold, the DSU power gates the victim cache. A trigger is also required to turn the victim cache on when the program reaches a phase where it can benefit from the victim cache again. If the number of recent store misses (accumulated in the store misses counter) exceeds a threshold, the DSU turns the victim cache back on.

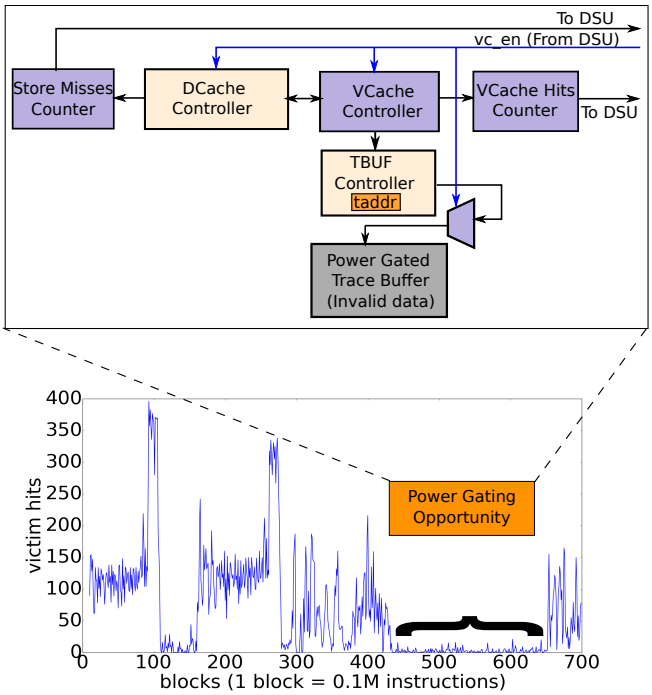


Fig. 5: Top: Proposed modification for power optimization. Bottom: victim cache hits during a sequence of 70M instructions of the *perlbench* benchmark

Algorithm 1 outlines the decision-making process for power gating the victim cache, made by the Debug Support Unit. A *block* refers to a sequence of executed instructions, and the decision for a possible change in power gating status is triggered on block boundaries. A *window* is a sequence of consecutive blocks during which the cache and victim cache statistics are monitored for possible power gating status change. Figure 6 illustrates an example scenario. When

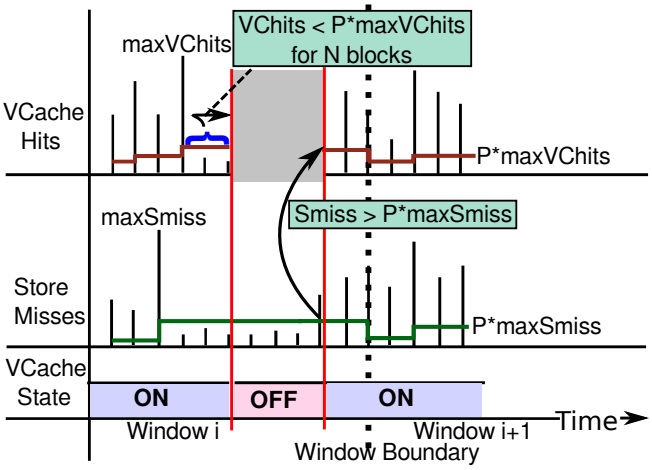


Fig. 6: Power gating of the Victim cache. N = 2

VCache hits drop for N consecutive blocks, to below a fraction P of the maximum hits in a window or an absolute limit T, the VCache is power-gated. When the store misses exceed fraction P of the maximum misses in a window, the VCache is turned back on. Lines 1-10 keep track of the maximum values for VCache hits and store misses. Lines 11-16 implement the power gating logic. The values of block size, window size, P, T, and N are experimentally determined.

To implement our power optimization technique, two relatively narrow (commensurate with the maximum expected victim cache hit count) registers are required on the core's victim cache controller to maintain the victim cache hit count and store misses count, and four registers and a comparator on the DSU.

Algorithm 1 Victim cache control

Inputs: *storeMisses*: #store misses in a block, *VCacheHits*: #VCache hits in a block, P: Power gating threshold fraction

Output: vc_en : Control signal to VCache

1: **if** *VCacheHits* $>$ *maxVHits* **then**
2: *maxVHits* $\leftarrow$ *VCacheHits*.
3: **end if**
4: **if** *storeMisses* $>$ *maxSMisses* **then**
5: *maxSMisses* $\leftarrow$ *storeMisses*.
6: **end if**
7: **if** *windowBoundaryEvent* **then**
8: *maxSMisses* $\leftarrow$ *storeMisses*.
9: *maxVHits* $\leftarrow$ *VCacheHits*.
10: **end if**
11: **if** (*VCacheHits* $<$ ($P \times$ *maxVHits*) for N consecutive blocks) OR (*VCacheHits* $< T$) **then**
12: *vc_en* $\leftarrow$ OFF
13: **end if**
14: **if** *storeMisses* $>$ ($P \times$ *maxSMisses*) **then**
15: *vc_en* $\leftarrow$ ON
16: **end if**

IV. Experiments

A. Setup

We implemented our victim cache design on the *LEON3*, 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 of 1KB-64KB. We synthesized our design using Cadence Encounter *RTL compiler*with a 90nm technology standard cell library to understand the area, timing, and power costs of our proposal in the normal and power gated mode. 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 [16] to study the performance effects and to keep track of time intervals in which the victim cache was power gated. 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 [17] tool to identify a representative dynamic instruction sequence of length one billion.

B. Performance improvement over baseline architecture

We first evaluate the overall system performance improvements obtained by using the trace buffer as a victim cache compared to the base architecture without applying the power gating optimization of Algorithm 1. Figure 7 shows the speedup attained in ten benchmarks for different data cache sizes while fixing the trace buffer at 2.5KB, 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% for the *Gamess* benchmark 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 the eviction is around 90%. These observations are independently corroborated by other studies [18]. 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 improvement with the *mcf* benchmark, although it is a cache intensive program. The reason for this is, 50% of the evicted lines are not reused and therefore, do not benefit from the victim cache.

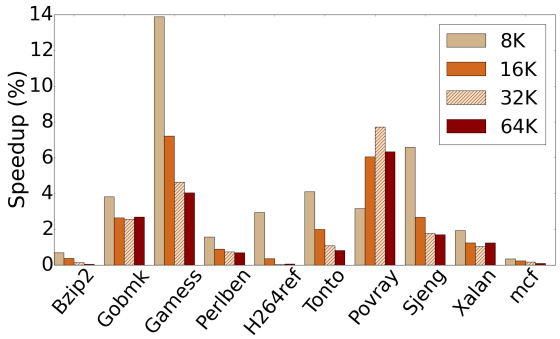


Fig. 7: Performance improvement for different data cache sizes. Trace buffer size = 2.5KB

Figure 8 shows the performance improvements when varying both the trace buffer and data cache sizes, averaged over all the ten 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 performance gains, which is significant considering that the DFD structure would have gone unutilized without this reuse.

C. Impact on Energy Delay Product

We evaluate the EDP gains obtained by enabling the power gating optimization of Algorithm 1. This optimization saves energy but could potentially reduce performance because when the victim cache is power gated, the data cache misses that could have been served by the victim cache have to be directed to the memory.

We performed extensive exploration over an independent set of applications to determine suitable values for the various

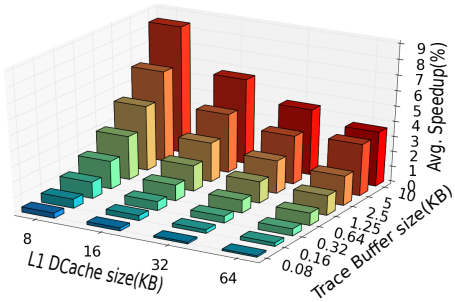


Fig. 8: Impact of varying trace buffer sizes

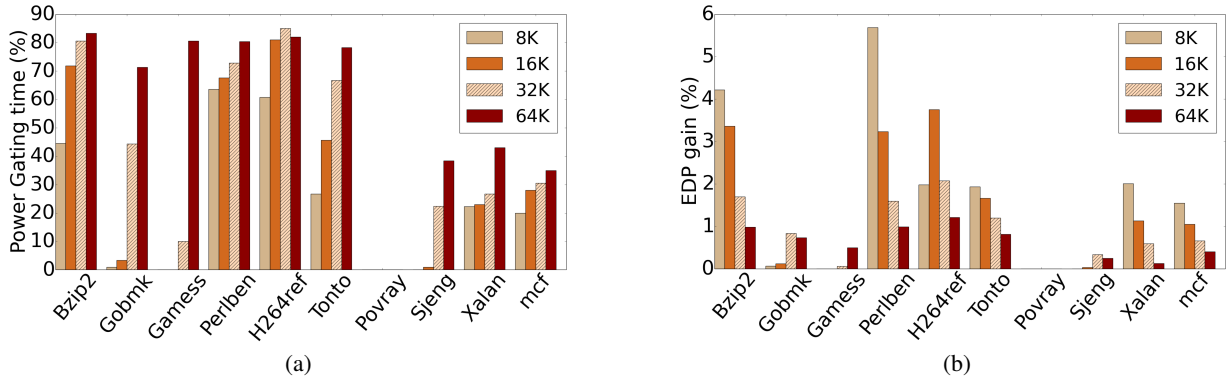
parameters of Algorithm 1. We omit the details due to lack of space. In summary, we used block size = 0.1 million, window size = 400, $P = 0.1$, $T = 20$, and $N = 2$.

Figure 9a shows the victim cache power gating duration as a fraction of the total execution time. These results are consistent with the observations in Figure 7; the performance of the benchmarks *bzip2*, *h264ref*, *perlbench*, and *xalancbmk* does not improve significantly with the victim cache addition; hence, its power gating saves energy. Figure 9b reports the EDP gain obtained for the overall design (including the victim cache and all other processor subsystems) for different data cache sizes, keeping a 2.5KB trace buffer. The relatively higher overall EDP gain for the *perlbench* benchmark is due to the higher extent of power gating achieved without losing significant performance. The high power gating duration for *H264ref* is accompanied by some performance loss, leading to lower overall energy savings. The energy savings for *Gamess* benchmark is negligible which is consistent with the significant performance improvement already obtained with the victim cache (Figure 7).

Figure 10 shows the impact of the power gating optimization, where we compare the EDP gain with optimization turned off and on, over the base architecture. Using Algorithm 1, we were able to reduce the EDP overhead for most benchmarks and configurations without affecting performance. It is worse with the 8KB cache because of the high power overhead observed (Section IV-D). The power gating optimization leads to slightly higher latencies on victim cache misses. However, our experiments showed that with the power optimization turned on, the average loss observed is only 0.15% over all the benchmarks and cache sizes.

D. Synthesis Results

We observe an area overhead of 0.41% for a 3KB trace buffer mainly because of victim cache controller, and minor changes in the data cache controller design. The power overhead for different data cache sizes are 10.1% (8 KB), 5.35% (16 KB), 2.45% (32 KB) and 1.41% (64KB), mostly due to the additional dynamic power when the trace buffer is treated as victim cache. There is no change in the critical path, and hence, no cycle time overhead. The latency overheads due



(a) (b)

Fig. 9: (a) Power Gating duration and (b) System EDP gain for different DCache sizes, P=0.1 and window size=400

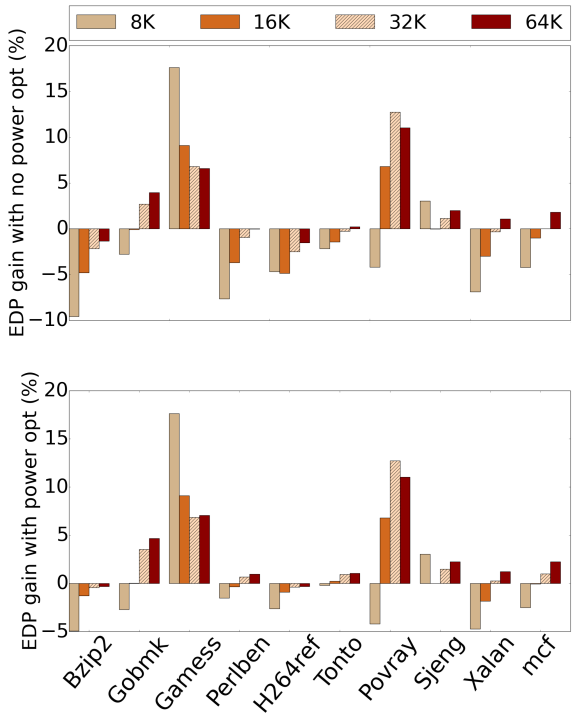


Fig. 10: Impact of victim cache power optimization for varying data cache sizes. Top: EDP gain without power opt. Bottom: EDP gain with power opt. P=0.1, window size=400

to power gating are very small in comparison to the gating durations, and have been ignored.

V. 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. Since the victim cache is of variable utility to applications, we also proposed a technique for identifying stretches of time when it is not very useful, and integrated this into a power gating optimization. The result is a non-standard victim cache design which reuses the storage area of the trace buffer, that improves both overall system performance and EDP. In the future we plan to study other reuse possibilities of on-chip debug structures, along with generalized algorithms for dynamic multiplexing between specific reuse scenarios.

Acknowledgment

This work was partially supported by a research grant from Freescale and Semiconductor Research Corporation.

References

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

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

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

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

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

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

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

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

[9] R. Abdel-Khalek and V. Bertacco, "Post-silicon platform for functional diagnosis and debug of networks-on-chip," *TECS*, vol. 13(3s), 2014.

[10] C.-H. Lai, Y.-C. Yang, and I.-J. Huang, "A versatile data cache for trace buffer support," *TCAS I*, vol. 61, no. 11, 2014.

[11] Y. Chen, T. Chen, L. Li, R. Wu, D. Liu, and W. Hu, "Deterministic replay using global clock," *TACO*, vol. 10, no. 1, 2013.

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

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

[14] G. Contreras and M. Martonosi, "Power prediction for Intel XScale® processors using performance monitoring unit events," in *ISLPED*, 2005.

[15] X. Chen, C. Xu, and R. P. Dick, "Memory access aware on-line voltage control for performance and energy optimization," in *ICCAD*, 2010.

[16] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, 2011.

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

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