© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, “ZeroCost-LLC: Shared LLCs at No Cost to WCL,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

ZeroCost-LLC: Shared LLCs at No Cost to WCL

Zhuanhao Wu
University of Waterloo
Waterloo, ON, Canada
zhuanhao.wu@uwaterloo.ca

Anirudh Kaushik
Intel Corporation
Toronto, ON, Canada
anirudh.kaushik@intel.com

Hiren Patel
University of Waterloo
Waterloo, ON, Canada
hiren.patel@uwaterloo.ca

***Abstract*—ZeroCost-LLC (ZCLLC) is a shared inclusive last-level cache (LLC) architecture for predictable multicore platforms that does not incur additional cost to the worst-case latency (WCL) of memory requests when compared to the memory hierarchy without an LLC. Thus, the WCL remains the same as without an LLC in the memory hierarchy, but with the performance benefits of having an LLC, in the form of additional caching capacity. ZCLLC achieves this by eliminating all cache line invalidations, and proactively updating the main memory with cache lines to preserve an important vacancy invariant. Furthermore, ZCLLC does not impose any constraints on the way the LLC is used unlike other approaches such as LLC partitioning. Our analysis reveals that the WCL is 55.6%, 68.0%, and 80.2% lower, and the performance is 2.4%, 7.2%, and 25.6% better than the state-of-the-art LLC partition sharing techniques for 2, 4, and 8 cores, respectively.**

I. Introduction

Multi-cores are deployed in real-time domains such as advanced driver assistance systems (ADAS), avionics, and safety-critical robotics [1], [2]. Shared hardware components in the multi-core architecture such as interconnects and shared caches are sources of timing interference that affect the timing predictability of multi-core platforms. This is because the timing behavior of one core's access to a shared hardware resource affects the timing behavior of another core's access to the shared hardware resource [3], [4]. Such timing interference must be accounted for in worst-case execution time (WCET) analyses. This work focuses on addressing the timing interference arising from shared inclusive last-level caches (LLCs).

A key contributor to the timing interference in a shared inclusive LLC occurs when a core's memory request to a cache line misses in both its private cache and the shared LLC. If there is no space in the LLC to hold the requested cache line, then the LLC must make space by removing an existing cache line in the LLC. Depending on whether the cache line chosen for removal is present in the cores' private caches, this cache line removal from the LLC triggers additional invalidations of the cache line in the cores' private caches that have the cache line. We refer to this chain of invalidations as *back-invalidations* [5]. These back invalidations are a major contributor to the worst-case latency (WCL) of a memory request [6], [7].

One approach to reduce such back-invalidations is through LLC partitioning where each core has exclusive access to their allocated LLC partitions [8], [9]. However, a key downside of LLC partitioning is that they restrict cores' access to the full LLC resulting in missed data caching opportunities and as a result, impose some average-case performance penalty [8], [10]. Furthermore, LLC partitioning does not eliminate all back-invalidations. Recent work [7] addressed this limitation of LLC partitioning by proposing a technique wherein cores can share LLC partitions in a predictable manner. Their work showed that sharing LLC partitions between cores increases the occurrences of back-invalidations, and these back-invalidations contribute to the WCL. Further, their work showed that the WCL for a multi-core platform with shared LLC with no partitioning constraints is orders of magnitude larger than that with no LLC[1]. While allowing cores to share LLC partitions is a welcomed step towards higher average-case performance, its high WCL questions its applicability in real-time platforms.

In this work, we ask and answer the following question: *Can we design a shared inclusive LLC with the following features: (1) all cores can access the full LLC (no partitioning constraints), and (2) the WCL of a core's request is the same as that without an LLC?* Such an LLC architecture is appealing for real-time multi-core platforms as it offers the performance benefits of using an LLC, but it does not incur additional latency to the WCL caused by back-invalidations. In other words, an LLC can be added to a multi-core platform at *no cost to the WCL.*

We answer this question in the affirmative, and present a novel LLC architecture called ZeroCost-LLC (ZCLLC). ZCLLC uses a recent work called zero-invalidation-victim (ZIV) [11] to completely eliminate all back-invalidations. ZCLLC borrows the following key insight from ZIV: eliminating back-invalidations entails finding a location in the LLC to insert the requested cache line (that causes the back-invalidation) that is either vacant or has a cache line that is not present in any of the cores' private caches. Evicting such a cache line does not trigger a back-invalidation. Despite ZIV addressing a central issue related to back-invalidations, we discovered that directly employing ZIV with a TDM arbiter [12] resulted in an unbounded WCL. The reason for this was that the location found by ZIV, despite not being cached by any cores' private caches, could contain dirty data that had to be written back to the main memory from the LLC. This delayed the LLC from fetching the requested cache line from the main memory during which other cores'

[1]Note that back-invalidations happen under certain situations based on the state of the LLC and private caches. However, the worst-case scenario for a core's memory request is when it triggers a back-invalidation in the LLC.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

requests could intercept the soon-to-be-vacant cache entry for their own requests. This forced the original request to retry for a vacant cache entry indefinitely. We noticed that the work in [7] recognized a similar situation and proposed an ordering constraint on requests attempting to occupy the same vacant cache line entry called ROC, which bounded the WCL. A solution that combines ZIV and ROC bounds the WCL. The resulting bound, however, is quadratic with respect to the number of cores because the analysis still has to account for the dirty cache line to be written back to the main memory. Hence, a key contribution of our work with ZCLLC is ensuring that we always have a clean entry by enforcing a runtime invariant, *vacancy invariant*. This enforcement is accomplished by dynamically writing back dirty cache line entries from the LLC to the main memory. Our WCL analysis shows that ZCLLC does not introduce any additional cost to the worst-case latency of a memory request.

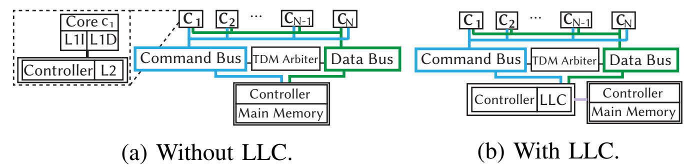
The main contributions of this work are:

- We analyze the impact of back-invalidations on the WCL and describe the ingredients necessary to eliminate back-invalidations.
- We describe a novel LLC architecture, ZCLLC, that recognizes the need to combine ZIV with ROC to eliminate back-invalidations while bounding the WCL, and enforces a runtime invariant to ensure the existence of clean entries in the LLC to further lower the cost on the per-request WCL.
- We present a WCL analysis of ZCLLC and show that ZCLLC does not introduce any additional cost to the WCL of a memory request.
- We evaluate ZCLLC in a detailed micro-architectural simulator and validate the worst-case bounds derived from our latency analysis. Our evaluation shows that ZCLLC offers 3.7%, 8.0%, and 25.0% better performance compared to the state-of-the-art LLC partition sharing techniques for 2, 4, and 8 cores setups, respectively.

II. System Model

A. Baseline Memory Hierarchy

Figure 1a shows our baseline system with N cores and a two-level inclusive cache hierarchy. Each core has a private L1 instruction cache (L1I) and a private L1 data cache (L1D), and a private unified set-associative L2 cache (L2). Each core's L1 caches connect to its L2 cache. All caches employ the least-recently-used (LRU) replacement policy. We further assume that there is at most one outstanding memory request per core. The cores' L2 caches communicate with the main memory via two shared buses: one for commands and one for data. We use a work-conserving time-division multiplexing (TDM) arbitration [13], [14] scheme that allocates one slot for each core to access the main memory. A TDM slot is long enough to complete one data transfer between the private L2 cache and the main memory. Such a deployment closely resembles platforms in [12], [15]–[17] without a shared LLC. Note that, our approach can be extended to other TDM schedules.



(a) Without LLC. (b) With LLC.

Fig. 1: System model.

The main memory controller accepts requests from the command bus and processes them in the order they are received. When a memory request misses in the L2, L2's cache controller issues a request to fetch the data from the main memory. The main memory sends the requested data on the shared data bus. For brevity, we refer to the core issuing commands on the bus or receiving responses to be analogous to its L2 cache controller issuing commands and receiving responses. Caches transfer data in the granularity of a *cache line*, which we assume to be 64-byte wide. We use cache line A to represent the cache line indexed by address A.

B. Memory Hierarchy with LLC

In Figure 1b, we augment the N-core baseline memory hierarchy with a *shared*, set-associative, write-back and write-allocate LLC. The cache hierarchy with the LLC is *inclusive*. This means that if a cache line is privately cached in L1 or L2, then this cache line is guaranteed to be cached in the LLC. If a cache line is not in the LLC then it is neither in the L1 nor L2. The LLC selects the LRU cache line (*victim* cache line) to replace on a conflict miss. The private L2 caches communicate with the shared LLC via the shared command and data buses. The LLC handles requests from the command bus in the order they are received and interacts with the main memory when a request misses in the LLC. We assume the capacity of the LLC to be greater than or equal to the aggregated capacity of all private caches [11]. An important component in our LLC is the tag array. Our tag array holds relocation information in addition to coherence-related states, and it is used for the same purpose as ZIV's sparse directory. For ZCLLC, the tag array is set-associative and mimics the organization of the private caches. This is similar to [18]. We refer the reader to ZIV's implementation for further details on the tag array's design [11], [19].

Core memory operations. A core issues one of the following memory operations: Read, Write, or WriteBack. Read and Write correspond to loads and stores by the core. If a Read or Write hits in the private caches, the requested data is returned to the core. Otherwise, the Read or Write is forwarded to the LLC. The core issues WriteBack to relinquish its private copy of a cache line either on a capacity miss or in response to another core's request. Note that in our approach, the WriteBack of a clean cache line must notify the LLC.

LLC Read/Write memory operations. We illustrate LLC Read and Write memory operations under our system model with an example, through which we introduce a well-known issue, *back-invalidation*, that may impact the WCL of memory

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

requests when deploying an LLC. When the LLC receives a Read or Write request from a core, it performs a FFLM (**F**etch **F**rom **L**LC or **M**ain Memory) to fetch the requested cache line, and send it to the requesting core's L2 cache controller. Consider the core under analysis, c_{ua}, issues a request to address A, denoted as Read(A), or Write(A). We denote the cache set that A maps to in the LLC as $s_L(A)$. Hence, c_{ua}'s request to cache line A maps to a cache set, $s_L(A)$, in the LLC. If A is cached in $s_L(A)$, then c_{ua}'s request is a *hit*, and the data of A is returned to c_{ua}. Otherwise, it is a *miss* and the LLC must fetch A from the main memory. Before fetching A, the LLC must ensure a vacant entry in $s_L(A)$ for A. If $s_L(A)$ has no vacant entries, then the LLC selects a victim line in $s_L(A)$, l, and evicts l to make space for A. However, victim l may be privately cached by another core. To ensure the inclusion property of the memory hierarchy, the LLC coordinates the eviction of l from all cores that have a private copy before invalidating the victim line in the LLC. We categorize back-invalidations into the following two types. (1) Given a BI from core c_i to core c_j, when $c_i \neq c_j$, the back-invalidation is called *cross back-invalidation* (CBI). (2) Otherwise when $c_i = c_j$, the back-invalidation is called *self back-invalidation* (SBI).

Definition 1 defines the conditions under which a *back-invalidation* happens.

Definition 1. *A back-invalidation (BI) from core c_i to core c_j happens when (1) c_i's request to a cache line A misses in the private caches and the LLC, (2) the cache set corresponding to c_i's requested line in the LLC, $s_L(A)$, is full, and (3) the victim selected by LLC for eviction is privately cached in c_j's private cache.*

Continuing with the example, the victim line l may be *dirty* (the data is modified when compared to the copy in the main memory); thus, it must be updated in the main memory. The LLC updates the main memory with the victim before servicing the request for A. For clean victims, the LLC only marks the entry in the LLC as invalid (no updates to the memory are generated).

LLC WriteBack memory operations. A WriteBack request for cache line A is denoted as WriteBack(A). When the WriteBack is sent to the LLC, the LLC performs a WTLM (**W**rite **T**o **L**LC or **M**ain Memory) that writes back the cache line received from the core to the LLC, and to the main memory. The WriteBack is the result of an SBI or a CBI.

III. Background

ZIV [11] presents a technique to eliminate back-invalidations in inclusive caches in general purpose systems. ZIV assumes that the capacity of the shared LLC is greater than the aggregated capacity of all private caches as stated in the system model. This assumption is essential for an LLC to be inclusive because the LLC must have enough capacity to guarantee that every privately cached line caches is also cached in the shared LLC. Thus, there *always* exist cache line entries in the LLC that are not privately cached by any private caches

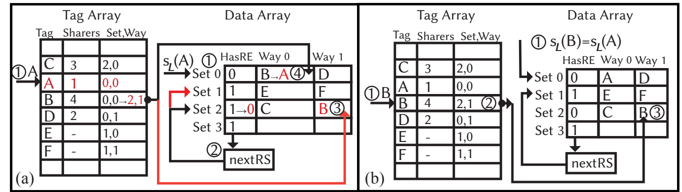


Fig. 2: (a) ZIV operation in LLC to avoid back-invalidations. (b) ZIV operation in LLC on a hit to a relocated cache line.

or vacant, which we refer to as *relocation entries* (REs). Note that an eviction of an RE does not incur any back-invalidation. The key insight in ZIV is that when the LLC selects a victim that is also privately cached, the LLC relocates the victim to an RE that is guaranteed to exist; thus, eliminating all back-invalidations. We use a register *nextRS* to track the location of an RE, and amend each LLC set with one additional bit, *HasRE*, to indicate the existence of at least one RE in that set.

We elaborate ZIV's operation using the example in Figure 2a. A core, c_{ua}, requests for A and ZIV checks the tag array. There is no tag entry with A (①); thus, the request is a miss in the LLC. Note that the red markings indicate the state after the handling of the miss for A. Each entry in the tag array in ZIV tracks the tag, sharers, and a location tuple of set index and way index that points to the cache line the entry is monitoring in the data array. For example, cache line B is owned by core c_4, and its data is stored in set 0 and way 0 in the data array. By changing the location tuple, ZIV allows a cache line to be relocated. The LLC queries the data array's set 0 ($s_L(A)$) and observes that there are no vacant entries since cache lines B and D occupy both ways. We assume that both lines are privately cached as well. ZIV selects B as the victim in $s_L(A)$. ② The LLC queries *nextRS* register for the location of a RE for B's target location. Suppose it points to set 2. ZIV uses an additional bit for each set in the data array, *HasRE*, to indicate a cache set with an RE. For example, set 0 does not contain an RE; thus, HasRE is 0. nextRS stores the set index of a cache set with an RE. Once a relocation happens, nextRS is updated to another cache set with RE in round-robin order, which is set 1. ③ Then, the LLC relocates B to the target location in set 2, and updates the location tuple of B's tag array entry to set 2, way 1. Future requests to the LLC that access B use the set and way indices in the tag array to correctly locate the data array entry. ④ Finally, the LLC fetches A from the main memory and use the relocated entry to store A. The state of the LLC after relocating cache line B is shown in Figure 2b.

In ZIV, the access to a non-relocated cache line, such as A, follows the conventional data path, where the tag array and data array are queried in parallel, revealing that the access hits in the LLC. The access to a relocated cache line, such as B, involves an extra data array query, as we show in Figure 2b. Similar to access to a non-relocated cache line, the access to cache line B starts by querying the tag array and data array in parallel (①). The query to the tag array is

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

a hit while the query to the data array is a miss. Recall from Figure 2a that cache line A and cache line B map to the same set ($s_L(A)$), and cache line B is relocated to set 2; hence, the miss in the data array. This indicates that cache line B was relocated in the data array. This means the LLC uses the location tuple from the array query to access the data array at set 2, way 1 (②). A second query to the data array will successfully retrieve the data (③). This second data array query lengthens the critical path to access a relocated cache line compared to non-relocated cache lines. However, ZIV [11] noted this to incur an additional latency of 3 cycles for extreme configurations such as a system with 768KB aggregated L2 capacity and a 1MB LLC, where the aggregated L2 capacity is close to the LLC capacity.This is negligible compared to the WCL, which is dominated by the main memory access. To avoid the large overhead of combinational logic for determining *nextRS*, ZIV [11] employs a bit-masking technique to determine *nextRS* such that it is practical to compute *nextRS*. Specifically, when there are multiple cache sets with REs available, the *nextRS* is updated to the cache set according to round-robin. ZIV reported that it took 6% of a 1MB LLC storage for supporting cache line relocation. The storage scales linearly with the LLC capacity. We use the same technique as [11] to enable cache line relocation and to remove back-invalidations.

IV. The Impact on WCL when Using a Shared LLC

Augmenting the memory hierarchy with a shared LLC for predictable multicore systems must be done carefully. This is because such a change in the memory hierarchy requires WCL analyses techniques to include the latency of accessing the LLC [7], [20], [21]. In this section, we show an alternative approach that incorporates a shared LLC with no partitioning constraints that has **no impact on the WCL**.

In this section, we use the asymptotic worst-case latency (AWCL) bound of a memory request [22] to present our insights and observations regarding state-of-the-art works on predictable LLC architectures and build to our solution. AWCL is the WCL asymptotically in terms of the number of cores and is a succinct formulation to understand how the WCL of a memory request under a certain mechanism scales with the number of cores. In Section VII, we derive the WCL of a memory request under our proposed approach.

A. Expository Setup

We begin our exposition with the model in Figure 1a that has no LLC. Then, we explore the effect of adding an LLC into the memory hierarchy resulting in Figure 1b.

Setup without LLC as in Figure 1a. Simply for illustration, suppose we restrict the cores in Figure 1a to only make accesses to distinct cache lines. This disallows accesses to shared data. With this setup, every memory request can complete within a slot. This is because the length of the slot includes the latency for transferring the data between the private caches and the main memory. Hence, the WCL for a memory request occurs when a memory request just misses the start of its TDM slot, and must wait for its next slot to perform the access. As a result, a core must wait for a TDM period in the worst-case before it successfully issues its request. Thus, the memory request suffers the worst-case arbitration latency of $O(N)$ slots resulting in an AWCL that is linear to the number of cores (N). There are several recent works [22], [23] that have a similar setup as this one, but they also allow accesses to shared data with an AWCL of $O(N)$. ZCLLC applies to those works as well.

Setup with LLC as in Figure 1b. When we add an inclusive shared LLC to the setup, we end up with the system model shown in Figure 1b. This addition requires us to investigate whether the WCL of a memory request has changed. Note that accessing the LLC itself does not increase the latency compared to accessing the main memory. This is because the access to the LLC can happen in parallel with the access to the shared memory once the LLC receives a request. Inevitably, the WCL analysis must account for interference caused by back-invalidations.

A recent work [7] showed that adding a shared inclusive LLC and enforcing an ordering constraint on which requests occupy vacant cache line entries yields an AWCL of $O(N^3)$. The $O(N^3)$ occurs because there are $O(N^2)$ BIs in the worst-case that cause interference on a request. We denote this ordering constraint as *request ordering constraint* (ROC). ROC prevents a younger request from occupying a vacant entry in the LLC that was released for an older request. The key intuition behind the cubic bound with ROC is that in the worst-case, there are $O(N^2)$ SBI and CBI that can be pending as a result of the ROC for a request, and each back-invalidation requires $O(N)$ slots to complete, which results in $O(N^3)$.

We highlight that by simply adding a shared inclusive LLC, the AWCL that was linear without the LLC ends up being cubic with the LLC. We find this to be a prohibitive cost to incur for adding a shared LLC. We acknowledge that there are other approaches that circumvent this cost. For example, a well practiced approach, LLC cache partitioning [8], offers an $O(N)$ AWCL. However, this comes with caveats that disallow data sharing, and require kernel and OS changes [9], [24]. More importantly, all the difficulties related to LLC partitioning such as partition underutilization, and poor scalability to larger number of cores continue to persist. Our hope in this work is to present an approach where we can incorporate an inclusive shared LLC without incurring additional overhead to the WCL, and correspondingly to the AWCL. The significance of this approach is that WCL analyses do not need to account for interference as a result of adding an LLC. Hence, the original WCL bounds as computed without the LLC hold.

B. Observations and Insights

Question 1. *Given that back-invalidations are the main sources of interference when introducing an LLC, can we eliminate all back-invalidations?*

As explained in Section III, the relocation strategy employed in ZIV can eliminate back-invalidations for multicore systems.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, “ZeroCost-LLC: Shared LLCs at No Cost to WCL,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

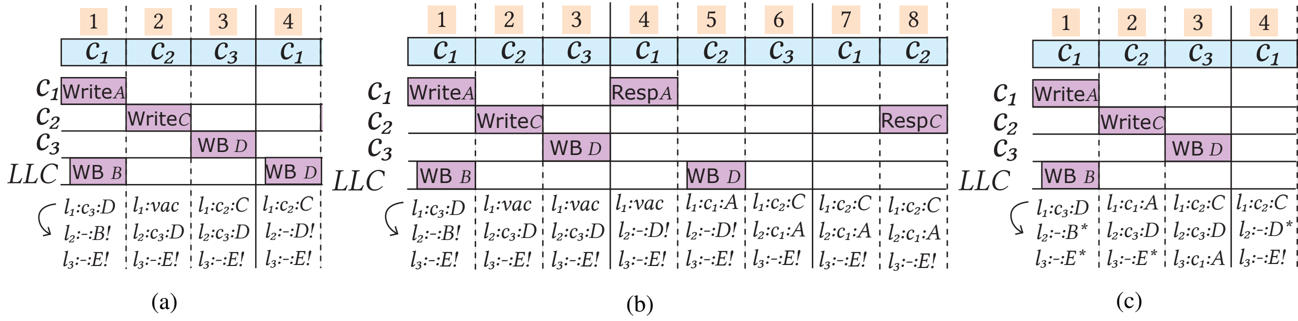


Fig. 3: Different approaches of incorporating ZIV in an LLC: (a) employing ZIV by itself results in unbounded WCL (b) ZIV with ROC (c) Lower WCL with memory update.

However, adopting ZIV for predictable multicore systems must be carefully examined for its impact on the WCL.

Unbounded WCL when using ZIV by itself. We discovered that employing ZIV by itself with a TDM arbitration scheme established in prior works [12] results in an unbounded WCL. The TDM arbitration scheme enforces that one TDM slot is large enough to conduct one data transfer between the private cache and the main memory. As a result, the LLC may fulfill a request across multiple TDM slots. Such a TDM arbitration scheme allows the LLC to interleave and handle requests non-atomically, yielding improved average-case performance. However, the TDM arbitration scheme only specifies the order in which the cores access the LLC, and does not specify the order in which the LLC handles the requests. *This separation, if left under-specified, can result in unbounded WCL.* In Figure 3a, consider that core c_1 issues a request to A in slot 1 that maps to $s_L(A)$, which has one cache line entry l_1 in the LLC. The LLC caches three cache lines B, D, and E, where D is privately cached while B and E are dirty lines, but not privately by any core. Since, l_1 has valid data D that is privately cached by c_3, $s_L(A)$ has no vacant entry. Since $s_L(A)$ has only one cache line, l_1 is selected as the victim line, and it must be relocated. Suppose that the target location selected by ZIV is l_2, which has a dirty entry (denoted by !). Before relocating l_1, the LLC must update the memory with contents of l_2. Recall that the length of a slot only permits one access to the main memory; hence, this update to memory consumes c_1's entire slot 1. At the end of slot 1, l_1 is vacant. However, in slot 2, c_2 makes a request to C that also maps to $s_L(A)$, and occupies this recently vacated entry l_1. In slot 3, c_3 writes back D in 3 for its next request. Notice that c_2 usurped the vacant entry l_1 that was freed for c_1's request to A. Hence, in c_1's next slot 4, c_1 has not completed its request to A; thus, the LLC attempts to complete the request. However, $s_L(A)$ still has no vacant entries causing the above steps to repeat. This pattern can continue indefinitely resulting in the WCL being unbounded. We observe that the key reason for the unbounded scenario is the same for which ROC was proposed as a remedy [7]. Therefore, we combine the enforcement of ROC with ZIV, and examine its effect on the WCL.

ZIV with ROC. Figure 3b shows an illustration where we combine ZIV with ROC (ZIV+ROC). We observe that a WCL bound exists, and the AWCL is O(N^2). We use the same memory requests in Figure 3a that resulted in the unbounded WCL in Figure 3b with ZIV+ROC. In slot 1, c_1 issues a request for A. The victim selected is l_1. ZIV relocates l_1 to l_2 to avoid back-invalidation. Since l_2 is dirty, the LLC updates the main memory with B. These steps are exactly the same as in Figure 3a. In slot 2, c_2 issues a request to C. Unlike before, ROC prevents c_2 from occupying l_1 even if it is vacant. This is because l_1 was freed as a result of c_1's request for A. In slot 3, c_3 updates the memory with D to make space for its next request. In slot 4, the LLC fetches A from the main memory, fills l_1 in the LLC, and sends A to c_1. Similarly, the LLC attempts to fulfill c_2's request in 5 and relocates l_1 to l_2, updating the dirty cache line D in the memory. Finally, c_2 obtains the response and completes its request in 8.

Although ZIV eliminates back-invalidations, ROC can delay a request. This occurs when the target location is dirty, and must first be updated in the main memory. This consumes the entire slot as shown in slot 1 in Figure 3b. While waiting for the core's next slot, other cores can issue requests, and ROC queues them. Thus, in the worst-case $O(N)$ requests to the same cache set such as the one in 2 in Figure 3b could be queued, and each one may need to update the main memory. This yields $O(N^2)$ AWCL. We do not provide the proof for computing this AWCL due to space constraints.

Insights 1. We derive two insights when using ZIV+ROC. (1) We have to assume the target location is dirty in the worst-case. This means the target location must be updated in the main memory before performing the relocation. (2) The AWCL is O(N^2). Therefore, using ZIV+ROC to add a shared LLC to the setup without the LLC increases the AWCL by $O(N)$ when compared to the setup with no LLC.

Question 2. *When relocating a victim line in ZIV+ROC, what happens if we select a target location that is guaranteed to be clean or vacant?*

Suppose we devise an approach that guarantees that on a relocation, there always exists a target location that is clean or vacant. Then, an update to the main memory is not needed in that slot. Consequently, the request that required the relocation

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

TABLE I: Symbols and names used in proofs and analysis.

Symbol	Explanation	Symbol	Explanation
SW	Slot width	T	L2 capacity
M	LLC capacity	N	Number of cores
c_i, c_{ua}	Core i, core under analysis	$s_L(A)$	LLC set cache line A maps to
$\mathcal{Q}$	Dirty lines cached in LLC and not cached in any of the private caches	$\mathcal{J}$	Clean lines cached in LLC but not cached in any private caches
WTLM	Write to LLC or main memory	FFLM	Fetch from LLC or main memory

could fetch the requested cache line in the same slot. Consider the example in Figure 3c, which shows the same requests from Figure 3b. The main difference is that l_2 and l_3 are clean with respect to the main memory, labeled as $l_2 : - : B^*$ and $l_3 : - : E^*$. Definition 2 formally defines such entries.

Definition 2. *A* clean relocation entry *(CRE) is a cache entry in the LLC that is a vacant entry or holds a clean cache line that is not privately cached.*

A CRE allows relocation of cache lines without triggering an update to the main memory. c_1's request for A can use the CRE l_2 on relocation. This allows the LLC to use the latency that was previously used for updating the dirty cache line in the main memory to instead fetch data from the main memory. Consequently, c_1 can receive its response in the same slot. Guaranteeing a CRE on a relocation results in an AWCL that is O(N).

Insights 2. Our main insight is that if we ensure the target location on relocation is always a CRE, then we can complete the request within that slot.

ZCLLC, uses ZIV+ROC and also ensures that a CRE is available on a relocation. ZCLLC ensures that in the worst-case, the number of CREs is always sufficient for the LLC to complete requests from cores without incurring updates to the main memory and back-invalidations. A key technique in maintaining a sufficient number of CREs is that ZCLLC performs memory update when a core voluntarily relinquishes its copy of cached data in its private cache resulting in a CRE. We explain these details in section V.

V. ZCLLC: Shared LLC at No Cost to WCL

The key novelty in ZCLLC is in being able to incorporate a shared inclusive LLC into the memory hierarchy without any cost to the WCL. ZCLLC does this by eliminating back-invalidations, and guarantees that no updates to the main memory are necessary on a relocation. Hence, ZCLLC ensures that there is always a CRE in the LLC when a relocation is done. ZCLLC enforces the *vacancy invariant* on a replacement (WriteBack) in the LLC. Guaranteeing the vacancy invariant requires dynamically deciding whether to update the main memory (UpdateMemory) or not. The main benefit is that when the vacancy invariant is satisfied, a request is guaranteed to finish within one slot. We explain how the vacancy invariant plays a role in ensuring the existence of a CRE in this section. Note that in our design, WriteBack of a clean cache line is not silent, and must be handled by the LLC. We collect the symbols used in this paper in Table I.

A. ZCLLC: Micro-architectural Extensions

ZCLLC reuses ZIV's approach to relocate cache lines to a new cache set and to access a relocated cache line. Hence, ZCLLC has the same storage overhead as ZIV for supporting cache relocation.

Counter. ZCLLC introduces a counter in the LLC to track the number of dirty cache lines that are valid in the LLC, but not cached in any private caches. Let $\mathcal{Q}$ denote the set of addresses of cache lines in the LLC that are not privately cached and dirty. ZCLLC tracks $|\mathcal{Q}|$.

Tracking CRE. We extend ZIV to support the tracking of CREs. Recall from Section III that ZIV tracks the cache set that has REs by augmenting each LLC set with a HasRE bit. ZCLLC reuses this bit to indicate that a cache set has at least one CRE, enabling the use of ZIV's hardware to relocate cache lines to CREs. Following ZIV [11]'s approach of tracking RE with a round-robin fashion, ZCLLC tracks CRE by augmenting each cache set in the LLC with one bit indicating whether there exists an entry that is clean and not privately cached. This bit evaluates to 1 when such an entry exists and 0 when there is no such an entry. For a w-way set-associative LLC with n sets, this amounts to an n-bit vector. Any set with a bit value of one contains a CRE and contains at least one entry that can be used by ZCLLC for relocation. In ZCLLC, we select the set that contains CREs in a round-robin order to minimize hardware cost. On every request, only bit positions corresponding to two sets are updated in the worst-case: (1) the cache set of the original request, and (2) the cache set of the relocation target. This incurs minimal cost in maintaining the bit vector.

B. ZCLLC: Dynamic Memory Update Policy on Replacements

ZCLLC dynamically enables main memory updates on replacements (WriteBack requests) from cores that relinquish their copy of data from the L2 cache. Note that a *main memory update*, or in short, a *memory update*, specifically means that the LLC synchronizes the contents of a dirty cache line in the LLC that is not privately cached by any core to the main memory, creating a CRE. On receiving a WriteBack, the LLC checks an invariant based on the counters to decide whether to update the main memory's copy of the addressed cache line that is dirty in the LLC. We call this the vacancy invariant (Definition 3). If this invariant is not satisfied, the LLC must perform a memory update. This ensures that when a future request results in a victim line that is privately cached, there is always a CRE to relocate to without triggering an update to the main memory.

Definition 3 (VINV)**.** *The vacancy invariant (VINV) is a predicate given by the following inequality:*

$$M - |\mathcal{Q}| \geqslant N \cdot T. \quad (1)$$

M is the capacity of the LLC, and T is the capacity of each of the private caches as shown in Table I. The left hand side

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

Algorithm 1: ZCLLC

```
State         : Q
Initialization: Q = ∅.
1  Loop
2    req ← ReceiveRequest();
3    ⟨type, addr, c_i⟩ ← ExtractRequest(req);
4    if type = WriteBack then
5      // Write-back caused by replacement in L2;
6      WTLM(addr, Q);
7    else
8      // The request is a Write or Read;
9      if IsMiss(addr) and not
         (HasVacantLine(s_L(addr)) or HasCleanLine(s_L(addr)))
         then
10       victim ← SelectVictim(s_L(addr));
11       if victim ∈ Q then
12         target ← SelectRelocationCRE();
13         Relocate(victim, target);
14       end
15     end
16     ZIVFFLM(req, c_i, Q);
17   end
18 EndLoop
```

Algorithm 2: WTLM

```
Input: addr, Q.
1 copy_count ← GetCachedCopies(addr);
2 if copy_count = 1 then
3   Q ← Q ∪ {addr};
4 end
5 if not VINVHoldTrue(Q) then
6   V ← SelectLine(Q);
7   UpdateMemory(V);
8   Q ← Q \ {V};
9 end
```

Fig. 4: Illustrative example.

of Inequality (1) is the number of cache entries in the LLC that is vacant, whose cache line is clean, or privately cached. These cache lines can be used or evicted without requiring an update to the main memory (UpdateMemory). The right hand side of Inequality (1) is the total number of entries in the private caches. When Inequality (1) holds, the LLC can always find a target location to relocate to without having to perform a UpdateMemory. Note that for an application that only performs Read operations, $\mathcal{Q}$ is always empty because there are no dirty cache lines, and VINV is trivially satisfied.

C. Implementing ZCLLC in the LLC Controller

The LLC controller implements ZCLLC. We present the key implementation details in Algorithms 1 and 2. ZCLLC starts by snooping a request req from the command bus (Line 2). The controller extracts the type of the request ($type$), the address of the requested cache line $addr$, and the core c_i that is issuing the request (Line 3). When the request type is a Read or Write and if the request is a miss, ZCLLC checks whether the cache set $s_L(addr)$ has a vacant or clean cache line with HasVacantLine and HasCleanLine (Line 9). If there is no vacant entry and there is no clean entry in $s_L(addr)$, ZCLLC selects a victim line in the set and relocates the victim to a CRE, and completes the request req (Lines 10 to 13). Otherwise, the access is a hit in the LLC, or $s_L(addr)$ has a vacant entry or clean entry. Then, ZCLLC invokes ZIVFFLM to fetch the requested cache line from the LLC or the main memory and send the data back to the requesting core c_i (Line 16). ZIVFFLM takes as input a request req, which is a Read or a Write, and fulfills it, assuming that either the request is a hit, or the request is a miss but there is a vacant entry or a clean entry in the cache set. Such conditions allow ZIVFFLM to fetch the data for the request without triggering a back-invalidation. Note that VINV guarantees that there is always a CRE in the LLC to relocate a privately cached victim without incurring a back-invalidation or memory update.

When the request is a WriteBack, the core is relinquishing a copy of the cache line. ZCLLC invokes WTLM to handle WriteBack and update main memory when necessary to create a CRE (Line 6). Algorithm 2 describes WTLM. WTLM accepts the request address $addr$, and the cache state $\mathcal{Q}$ as input. At a high-level, when a core issues a WriteBack for a cache line at address $addr$, this cache line can be (1) cached in multiple private caches and hence the cache line is clean and shared among multiple cores, or (2) cached only by the core that issues the WriteBack and is potentially dirty. In (1), WriteBack of the clean cache line does not alter $\mathcal{Q}$; while in (2), the WriteBack adds the dirty cache line to $\mathcal{Q}$, which may violate VINV as it is dependent on $\mathcal{Q}$. Hence, when a WriteBack happens, WTLM first checks whether $addr$ is cached by a single copy and modifies $\mathcal{Q}$ accordingly, and if VINV is violated, WTLM performs UpdateMemory for a cache line from $\mathcal{Q}$ to reduce the number of cache lines from $\mathcal{Q}$, maintaining the VINV. This is essential in ensuring that a CRE is available whenever a relocation is required.

WTLM first checks the number of cores that cache a copy of requested line $addr$ with GetCachedCopies (Line 1). If there is only one core that caches the copy, WTLM adds $addr$ to $\mathcal{Q}$ (Lines 2 to 4). Next, WTLM invokes VINVHoldTrue on Line 5 to decide whether a memory update is necessary to guarantee VINV to provide enough CREs. When VINVHoldTrue($\mathcal{Q}$) evaluates to False, meaning that the VINV is violated, there are insufficient CREs and the LLC must perform a memory update. Internally, VINVHoldTrue checks whether the Inequality (1) holds. Otherwise, there are sufficient CREs in the LLC and no memory updates are required.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

D. Illustrative Example

We use Figure 4 to explain ZCLLC's operation with a focus on ZCLLC's ability to enforce VINV.

Example setup. This example has three cores following the TDM schedule of $[c_1, c_2, c_3]$. Each private cache contains only one cache line. Figure 4 shows the content of each private cache in $Prv\$$ row. We use an underscore (_) to represent a vacant entry. The LLC has four sets and each set contains one cache line labeled as l_1, l_2, l_3 and l_4, respectively ($M = 4$ and $T = 1$). All private caches and the LLC start out not caching any cache lines ($\mathcal{Q} = \emptyset$). Without loss of generality, all requests considered in this example map to l_1.

Slots 1 to 3. In slot 1, c_1 issues a request to A that completes in the same slot because the vacant entry l_1 is available for use. In slot 2, c_2 issues a request to B, which maps to l_1. Following line 9 to 13 in Algorithm 1, ZCLLC relocates the contents of l_1 to the vacant cache line l_2; thus, no back-invalidation. Hence, c_2's request completes in slot 2 with B occupying l_1, and A being relocated to l_2. Similarly, c_3 issues a request in slot 3 that relocates l_1 to l_3.

Slots 4 to 9. In 4, c_1 wishes to request cache line D, but c_1's private cache is full. Hence, c_1 issues a WriteBack to write back A to the LLC. At the start of 4, on receiving the WriteBack request (Line 4 in Algorithm 1), the LLC checks if VINV holds if A in the LLC is *not* written to the main memory (Line 5 in Algorithm 2). Suppose that A is not written to the main memory, at the end of the slot, A will no longer be privately cached. The VINV from Inequality (1) holds: $4 - 1 \geq 3$. Thus, the LLC does not need to write A back to the main memory. In slot 5, c_2 issues a request to E, requiring a replacement on B; thus, generating a WriteBack to the LLC. Similarly, the LLC checks whether VINV holds if B is not written to the main memory. If B is *not* written to the main memory, at the end of 5 $\mathcal{Q} = \{A, B\}$. In this situation, VINV does not hold: $4 - 2 \geq 3$. Hence, the LLC updates the main memory with B selected by SelectLine (Lines 6-8 in Algorithm 2). This is essential because if B is not updated in the main memory, there will be 2 dirty cache lines, A and B in the LLC leaving l_4 as the only CRE. However, there are 2 vacant entries in the private caches of c_1 and c_2, allowing them to make 2 requests. With B written back to the main memory, VINV holds. In slot 6, c_3 performs a replacement to make space for its memory request. Similar to 5, the LLC must write back C. Now, the LLC ends up with $\mathcal{Q} = \{A\}$ that still needs to be written back to the main memory, and $\{B, C\}$ are clean with respect to the main memory. In slot 7, c_1 issues its request to D that maps to l_1. The LLC can safely use l_1 in the LLC because its content was updated in the memory in slot 6. Then, the LLC fetches D from the main memory and sends D to c_1. c_1's request finishes within 7 without any back-invalidation. In 8, c_2's requests E. ZCLLC relocates D to l_4, to vacate l_1 and fetches E from the main memory. Note that although ZCLLC could choose l_3 since it is clean, it chooses the vacant entry l_4 to maximize LLC utilization. In 9, c_3's request to F maps to l_1 and the LLC relocates E to l_3, which is a CRE with clean cache line. No back-invalidation occurs.

When handling a WriteBack, the LLC ensures VINV holds if it does not perform a memory update. If this invariant does not hold, for example, when the LLC does not update the main memory with B in slot 5, the LLC will end up with only two CREs, l_1 and l_4, for relocation at the end of slot 6. Two CREs for this example cannot guarantee back-invalidation-free service for three requests in slots 7, 8 and 9.

E. Characterizing ZCLLC's Write Policy

ZCLLC is, by definition [25], a write-back cache and not a write-through cache, because a Write to the LLC always updates content in the cache and not the main memory. Further, the main memory only gets updated when ZCLLC performs a WriteBack. To maintain VINV, we make ZCLLC behave like a write-through cache. This occurs when a WriteBack triggers an UpdateMemory to guarantee that a specific number of CREs are always available as dictated by VINV. For ZCLLC to act like a write-through cache, the LLC state must be as follows: cache lines in the LLC are either privately cached or not privately cached, but dirty. If there are enough CREs as specified by VINV, a WriteBack does not trigger any main memory updates, and ZCLLC acts the same as a write-back LLC. It is noteworthy that ZCLLC will not be stuck in the LLC state that forces write-through cache behaviors. This is because Read requests from cores and their corresponding WriteBack will eventually increase the number of CREs even if VINV holds. In this work, we do not include a quantitative comparison with write-through caches. This is because prior works on predictable cache coherence [12], [22], [23] assume write-back caches. Extending predictable cache coherence protocols with write-through caches requires non-trivial coherence protocol changes whose correctness needs careful investigation, which is beyond the scope of this work.

VI. ZCLLC's Correctness

In this section, we show that ZCLLC maintains the VINV in the LLC in the following way. First, we show that servicing Read and Write requests does not violate the VINV in Lemma 1. Next, we show that ZCLLC also maintains VINV when completing WriteBack requests in Lemma 2. Theorem 1 combines the lemmas and shows that VINV holds in ZCLLC.

A. Ensuring Vacancy Invariant in ZCLLC

We describe the LLC state that ZCLLC uses as a tuple $\mathcal{S} = (\mathcal{Q}, \mathcal{J})$. Note that $\mathcal{J}$ is the set of cache lines that are only cached in the LLC and are clean (Table I). We use the syntax $\mathcal{S} + \mathsf{Req(A)}$ to represent the LLC state after performing a request Req on address A according to Algorithm 1, where $\mathsf{Req} \in \{\mathsf{Write}, \mathsf{Read}, \mathsf{WriteBack}\}$. Further, we use VINVHoldTrue($\mathcal{Q}$) to determine whether Inequality (1) holds given state $\mathcal{S}$.

Lemma 1. *Given an LLC state $\mathcal{S} = (\mathcal{Q}, \mathcal{J})$, and its state after performing a request, $\mathcal{S} + \mathsf{Req(A)} = (\mathcal{Q}', \mathcal{J}')$,*

$$\mathit{VINVHoldTrue}(\mathcal{Q}) \Rightarrow \mathit{VINVHoldTrue}(\mathcal{Q}'),$$

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

where Req $\in \{Read, Write\}$.

Proof. Assume VINVHoldTrue($\mathcal{S}$) is true, that is, Inequality (1) holds:

$$M - |\mathcal{Q}| \geqslant N \cdot T \tag{2}$$

Let core c_k issue request Req. This request can be a miss or a hit in the LLC.

Case 1: LLC miss. When Req is a miss, $A \notin \mathcal{Q} \cup \mathcal{J}$. The LLC must fetch A from main memory and send A to c_k. After Req, $\mathcal{Q}' = \mathcal{Q}$, and $\mathcal{J}' = \mathcal{J}$. Substituting $\mathcal{Q}'$ into Inequality (2) gives VINV of $\mathcal{S} +$ Req:

$$M - |\mathcal{Q}| \geqslant N \cdot T.$$

Both sides of the expanded inequality remain the same compared to Inequality (2); thus, VINV holds.

Case 2: LLC hit. If Req is a hit in the LLC, A can be in $\mathcal{Q}$, $\mathcal{J}$ or privately cached by at least a core $c_j \neq c_k$.

(1) If A is privately cached by at least a core c_j, after Req, $\mathcal{Q}' = \mathcal{Q}$, and $\mathcal{J}' = \mathcal{J}$. The left-hand side VINV remains unchanged; thus, VINV holds.

(2) If $A \in \mathcal{Q}$, after Req, $\mathcal{Q}' = \mathcal{Q} \setminus \{A\}$, and $\mathcal{J}' = \mathcal{J}$. Note that $\mathcal{Q} \setminus \{A\}$ refers to the set difference between set $\mathcal{Q}$ and set $\{A\}$.

The left-hand side of VINV increases by 1; thus, VINV holds.

(3) If $A \in \mathcal{J}$, after Req, $\mathcal{J}' = \mathcal{J} \setminus \{A\}$, and $\mathcal{Q}' = \mathcal{Q}$. Notice that $\mathcal{J}$ is not in Inequality (2); thus, VINV is unaffected.

In (1), (2), and (3), because Inequality (2) holds, the VINV for $\mathcal{S} +$ Req also holds. Hence, VINVHoldTrue($\mathcal{Q}'$). □

Key takeaway. *Lemma 1 shows that servicing Read and Write requests in ZCLLC does not violate the VINV. This means that a Read or Write request from a core that reaches ZCLLC does not result in back-invalidations.*

Lemma 2. *Given an LLC state $\mathcal{S} = (\mathcal{Q}, \mathcal{J})$, and its state after performing a WriteBack, $\mathcal{S} + WriteBack(A) = (\mathcal{Q}', \mathcal{J}')$,*

$$\textit{VINVHoldTrue}(\mathcal{Q}) \Rightarrow \textit{VINVHoldTrue}(\mathcal{Q}').$$

Proof. For a WriteBack request to A, Algorithm 1 only modifies $\mathcal{Q}$ in its invocation to WTLM (Line 6). WTLM is described in Algorithm 2. Hence, it suffices to show that VINV holds after performing WTLM in Algorithm 2.

Suppose VINV holds before the execution of Algorithm 2; hence, $M - |\mathcal{Q}| \geq N \cdot T$. We use $\mathcal{Q}^\star$ to denote the contents of $\mathcal{Q}$ after executing lines 1-3 in Algorithm 2. Depending on how many cores have a copy of A in their private caches, there are two cases.

Case 1: More than one core has A in their private caches. If more than one core has A in their private caches, then LLC copy of A is not dirty. Hence, the condition on line 2 is not satisfied and $\mathcal{Q}^\star$= $\mathcal{Q}$. This means that no further modifications to $\mathcal{Q}$ are performed; hence, $\mathcal{Q}$' = $\mathcal{Q}$ after WriteBack(A). Since $\mathcal{Q}$' = $\mathcal{Q}$ and VINVHoldTrue($\mathcal{Q}$) holds, VINVHoldTrue ($\mathcal{Q}$') also holds.

Case 2: Only one core has A in its private cache. There are two sub-cases to consider.

Case 2.1: A is not dirty in the LLC. In this case, the core that has a copy of A in its private cache has not modified its contents; hence, A in the LLC is not dirty. This is similar to Case 1; hence, VINVHoldTrue ($\mathcal{Q}$') holds.

Case 2.2: A is dirty in the LLC. Since A is dirty in the LLC, WriteBack(A) results in $\mathcal{Q}^\star \leftarrow \mathcal{Q} \cup \{A\}$ shown in Line 3 in Algorithm 2. Since, $\mathcal{Q}^\star \neq \mathcal{Q}$, the LLC must check whether VINVHoldTrue($\mathcal{Q}^\star$) holds as shown in Line 5 in Algorithm 2. If VINVHoldTrue($\mathcal{Q}^\star$) does not hold, then a cache line is selected from $\mathcal{Q}^\star$ for eviction, creating a CRE (Line 6). We now show that it is always possible to to select a cache line V from $\mathcal{Q}^\star$ for UpdateMemory that creates a CRE.

Notice that $M \geq N \cdot T$ (Section II) and $|\mathcal{Q}^\star| = |\mathcal{Q}| + 1$. Rearranging the terms and substituting $|\mathcal{Q}| = |\mathcal{Q}^\star| - 1$ in Inequality 1, we get $|\mathcal{Q}^\star| = M - N \cdot T + 1 > 0$. This means that $\mathcal{Q}^\star$ is not empty and it is always possible to select a cache line V from $\mathcal{Q}^\star$ for UpdateMemory that creates a CRE. Finally, Line 8 removes V element from $\mathcal{Q}^\star$. Hence, after WTLM, $\mathcal{Q}' = \mathcal{Q}^\star$ and $|\mathcal{Q}| = |\mathcal{Q}'|$. Thus, VINVHoldTrue($\mathcal{Q}'$) holds. □

Key takeaway. *Lemma 2 shows that ZCLLC ensures VINV holds after servicing WriteBack requests. This means that any subsequent Read and Write requests from cores that miss in the private caches and LLC will not result in back-invalidations as there is always a CRE available.*

Theorem 1 (invariant)**.** *Given an LLC state $\mathcal{S}$, ZCLLC maintains the VINV.*

Proof. (By induction on requests.) **Base case.** Initially, the LLC is empty, that is $\mathcal{Q} = \emptyset$. Inequality (1) reduces to $M \geqslant N \cdot T$. This is an assumption required to guarantee inclusion property for the LLC.

Induction hypothesis. The invariant holds for the first y requests.

Inductive step. We prove that the invariant holds for the $(y+1)$-th request. The $(y+1)$-th request is either a WriteBack, a Write, or a Read. Lemma 1 shows that VINV holds for a Write or a Read. Lemma 2 shows that VINV holds when WriteBack is performed on a cache line. The two lemmas cover all scenarios of a request processed by the LLC. As a result, VINV holds for the $(y+1)$-th slot. □

Key takeaway. Theorem 1 combines Lemma 1 and Lemma 2 and shows that ZCLLC maintains VINV.

Corollary 1. *Using ZCLLC, in the worst-case, the number of back-invalidations and main memory update in the LLC that interfere with the request from the core under analysis is* 0.

Proof. As shown in Theorem 1, VINV always holds in the LLC. Hence, WriteBack always completes in one slot. Read and Write also complete in the same slot without causing a back-invalidation or a write-back. Therefore, no back-invalidation or write-back causes interference. □

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, “ZeroCost-LLC: Shared LLCs at No Cost to WCL,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

VII. Worst-case Latency Analysis of ZCLLC

Our latency analysis focuses on FFLM and WTLM of the LLC. The FFLM services a Read or Write request from a core under analysis c_{ua} while a WTLM services a WriteBack request from c_{ua}. We use L_{Arb} to denote the worst-case latency required for a core to be serviced a slot to perform either a Read, Write or a WriteBack. Our system setup assumes a TDM arbitration scheme, where the schedule is deployed with one slot per core $L_{\text{Arb}} = (N-1) \times SW$.

Theorem 2. *For ZCLLC, the WCL of performing WTLM for c_{ua}'s request in the LLC, WCL_{WTLM}, is*

$$WCL_{WTLM} = L_{Arb} + SW.$$

Proof. The WCL_{WTLM} is composed of the arbitration latency L_{Arb} for c_{ua}'s request to be issued and the latency to perform WTLM. WTLM results in either the LLC updating the LLC data cache or updating the main memory, both of which can complete within SW. □

Theorem 3. *For ZCLLC, the WCL of performing FFLM for c_{ua}'s request in the LLC is*

$$WCL_{FFLM} = L_{Arb} + SW.$$

Proof. The WCL_{FFLM} is composed of the arbitration latency L_{Arb} for c_{ua}'s request to be issued and the latency to perform FFLM. Since ZCLLC does not incur any back-invalidations or memory updates (Corollary 1), FFLM can fetch data for c_{ua}'s request within a single slot, which costs SW. □

Table II contrasts ZCLLC with other approaches to limiting the number of back-invalidations and the number of memory updates in the worst-case, together with the resulting AWCL. ZCLLC is the only approach that achieves no back-invalidations and no memory updates.

TABLE II: Worst-case latency of various configurations.

Configurations	# of BI	# of memory update	AWCL
ROC [7]	$(N-1)(N-1)$	$N-1$	$O(N^3)$
ZIV+ROC	0	$N-1$	$O(N^2)$
ZCLLC	0	0	$O(N)$

VIII. Empirical Evaluation

We integrate our proposed mechanism in gem5 [26] micro-architectural simulator. We simulate configurations with 2, 4, and 8 cores for ROC, ZIV+ROC, and ZCLLC. In addition, we implement a general purpose data sharing coherent mechanism that has no guarantees on timing predictability (GP). We assume that each core is equipped with 4kB L1 private instruction and data caches, and 16kB private L2 caches. The private caches are backed by a shared 2MB LLC. The private caches and the shared LLC communicate through a shared bus deploying TDM arbitration with a slot width of 128 cycles, allowing one data transfer between the L2 and the main memory. We simulate the main memory with a DDR3 model with an access latency of 100ns.

TABLE III: Observed WCL (Obs.) and Analytical WCL (Analyt.) for different configurations (in cycles).

Core	ZCLLC		ZIV+ROC		ROC		GP
	Obs.	Analyt.	Obs.	Analyt.	Obs.	Analyt.	Obs.
2	612	640	1124	1152	1124	1152	1114
4	1124	1152	3172	3200	3164	3200	3316
8	2148	2176	9314	10368	10198	10368	24166

Our evaluation includes synthetic benchmarks to exercise the WCL and the SPLASH-3 [27] multi-core workloads. We also make our source code publicly available [28].

A. Worst-case Latency

We begin by investigating the WCL properties of different configurations.

Synthetic benchmarks. We develop synthetic benchmarks in an attempt to exercise the WCLs for different configurations. Our synthetic benchmarks involve the cores randomly generating memory accesses to locations that map to the same cache set in the LLC. Table III shows that for all configurations, the analytical WCLs safely bound the observed WCLs for all configurations with mechanisms to ensure timing predictability. We also show the observed WCLs for the GP configuration. The GP configuration exhibits a WCL of 840, 3566, and 18824 cycles, for 2, 4, and 8 cores, respectively. Note that the observed WCLs of 2-core, 4-core, and 8-core GP configurations exceed the analytical WCLs of the 2-core, 4-core, and 8-core ZCLLC configurations, respectively. The observed WCLs of the 4-core and the 8-core GP exceed the analytical WCLs of ZCLLC, ZIV+ROC, and ROC. Among the configurations, ZCLLC shows the lowest analytical WCLs of 640, 1152, and 2176 cycles compared to ROC and ZIV+ROC.

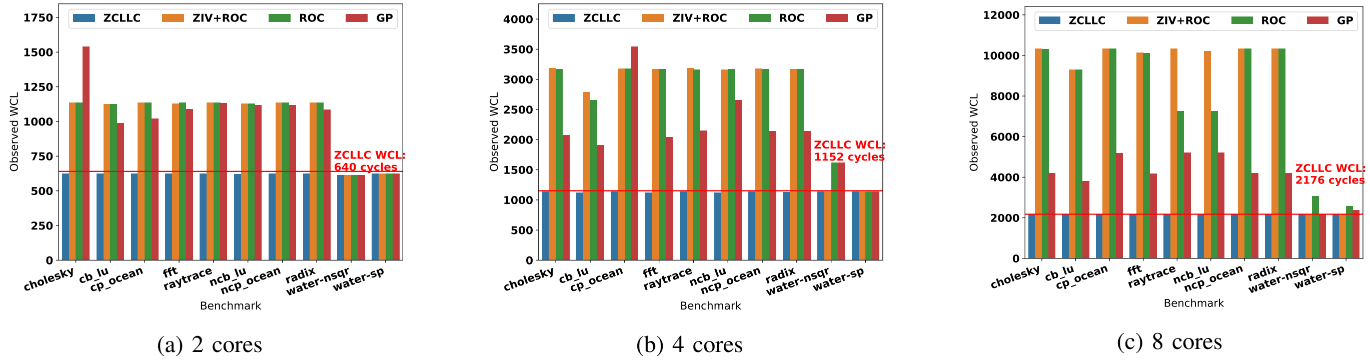
SPLASH-3. We also evaluate the observed WCL with the SPLASH-3 benchmarks. SPLASH-3 benchmarks contain a wide variety of complex parallel multi-threaded applications that exercise data sharing and synchronization across cores. We use SPLASH-3 benchmarks to study the impact of these inter-core interactions on performance and WCL when a system deploys ZCLLC. Our results again show that the observed WCL of ZCLLC, ZIV+ROC, and ROC are within the analytical WCL. Our computed analytical WCLs as shown in Table III shows that ZIV+ROC and ROC have strictly larger analytical WCL compared to ZCLLC, Figure 5 shows that the observed WCLs of ZIV+ROC and ROC are also larger than or equal to the observed WCLs of ZCLLC for 2, 4, and 8 core configurations across all benchmarks.

B. Performance

We evaluate the performance impact of ZCLLC on SPLASH-3 workloads. We observe that both ZIV+ROC and ZCLLC show a performance benefit compared to ROC. Figure 6 shows the speedup of ZCLLC, ZIV+ROC, and ROC over the general purpose coherence mechanism where no

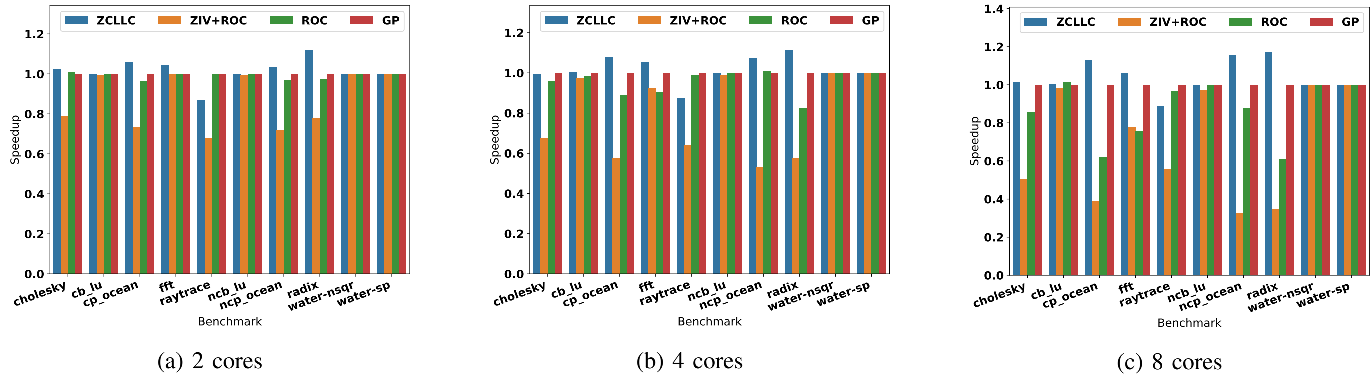
© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.



(a) 2 cores (b) 4 cores (c) 8 cores

Fig. 5: Observed worst-case latencies (in cycles).



(a) 2 cores (b) 4 cores (c) 8 cores

Fig. 6: Speedup of different configurations. Baseline configuration is GP.

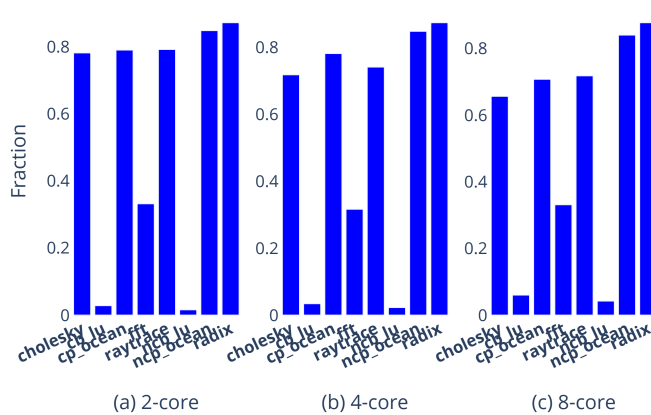


Fig. 7: Fraction of WriteBack that causes UpdateMemory.

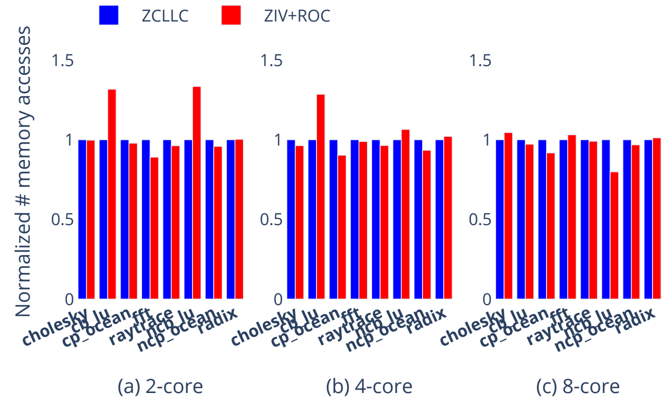


Fig. 8: Number of main memory accesses normalized to ZCLLC.

special mechanism guarantees timing predictability. For 2, 4, and 8 cores, ZCLLC shows an average speedup of 1.01×, 1.02×, and 1.04× compared to GP. ZCLLC also shows an average speedup of 1.02×, 1.07×, and 1.25× over ROC. The performance benefit shown in ZCLLC is a result of the removal of back-invalidations. Moreover, we observe that employing ZCLLC does not cause performance degradation compared to GP. Thus, ZCLLC preserves the performance benefits of ZIV-ROC while not suffering from the enlarged WCL.

C. WriteBack-induced Main Memory Accesses

In an inclusive LLC, as the execution proceeds, the LLC will be filled with dirty cache lines written back by the private caches. As a result, $\mathcal{Q}$ in VINV increases, and it is more likely for the LLC to perform an UpdateMemory to maintain VINV when processing WriteBack requests from the cores. Figure 7 validates this observation by showing the fraction of WriteBack that is converted into UpdateMemory (Line 7 in Algorithm 2). Note that benchmarks not presented indicate

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, "ZeroCost-LLC: Shared LLCs at No Cost to WCL," in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

that the WriteBack does not introduce any UpdateMemory in the LLC. Since ROC and ZIV+ROC do not incur WriteBack when processing UpdateMemory, their fractions will always be 0. For 2-core, 4-core, and 8-core setups that exhibit UpdateMemory, the fractions of WriteBack converted to UpdateMemory can be as high as 87.1%, 87.3%, and 87.7%, respectively. Figure 8 shows the number of main memory accesses normalized to ZCLLC. For 2-core, 4-core, and 8-core setups, the numbers of main memory accesses performed by ZIV+ROC are $1.06\times, 1.02\times$, and $0.97\times$ compared to the number of main memory accesses performed by ZCLLC. Hence, although ZCLLC shows a high UpdateMemory fraction, the total number of main memory accesses does not increase compared to ZIV+ROC.

Relationship to performance. At a high level, the total number of UpdateMemory affects performance directly since more UpdateMemory leads to more main memory transfer. However, a high fraction of WriteBack transformed into UpdateMemory does not indicate an increase in the total number of UpdateMemory. Instead, the fraction indicates the percentage of main memory transfers that are *moved* from Read and Write requests to WriteBack requests, which is not directly related to the performance. Specifically, the UpdateMemory experienced by ZCLLC for WriteBack would occur in the Read and Write requests in ZIV+ROC or ROC where a UpdateMemory or BI must be employed to create a vacant entry.

Relationship to write-policy. In our opinion, ZCLLC is a write-back cache. Although the UpdateMemory to WriteBack ratio can be high, we would like to point out that the UpdateMemory to WriteBack ratio identifies the ratio of WriteBacks that end up having to update the main memory. This does not suggest whether ZCLLC is either a write-back cache or a write-through cache. Consider the following scenario where we focus on writes to one cache set A in ZCLLC. Suppose that all entries in set A have had prior Writes to them; thus, every entry has valid data that has been modified. Any subsequent Writes to A that miss will require evicting a cache line before completing the request. Since this cache line is dirty with modified data, the main memory would need to be updated with this evicted cache line. In this case, the ratio will be 1.0, but this is not an indication of the LLC being write-through. The update to memory only happens in this scenario because set A is full. Therefore, we feel that ZCLLC is a write-back cache. ZCLLC could behave like a write-through cache when the number of CREs is insufficient for guaranteeing VINV, and it would perform UpdateMemory to create a CRE under this scenario.

IX. Related Works

We broadly classify the rich body of research on estimating timing interference with LLCs into two main approaches: (1) methods that use static cache or measurement-based analyses [29]–[33], and (2) software and hardware techniques that minimize timing interference when using LLC such as LLC partitioning [8].

Static cache analysis of multi-level caches [29]–[33] estimates WCET of tasks by performing must-may and persistence analyses. However, these analyses assume there is no inter-core interference in the shared cache, which is in practice achieved by cache partitioning. ZCLLC does not impose any such constraints on the LLC. Further, with ZCLLC, we provide a per-request WCL that we envision could be used as an input to static cache analyses approaches. Building this into a WCL analysis framework is reserved for future work.

Cache partitioning [8] is often used to partition the LLC on predictable real-time multicore platforms. The main benefit of LLC partitioning is that it removes inter-core or inter-task timing interference by isolating a portion of the LLC to a core or task. As a result, LLC partitioning simplifies timing analyses [34] by allowing it to leverage existing single-core analysis approaches without having to incorporate the effect of accesses made by other cores. Cache partitioning can be implemented in software through page coloring mechanisms managed by the operating system [8], [35]–[37] or in hardware through hardware logic managed by the LLC controller [37]–[39]. A recent approach called cache bleaching [40] reduces address fragmentation and run-time recoloring overhead when page coloring is used to perform LLC partitioning. Cache bleaching uses programmable logic to change the addresses of requests from LLC before the requests arrive to the main memory, such that addresses of the same color map to a contiguous region in the main memory, allowing efficient recoloring in the OS without costly data migration. ZCLLC is orthogonal to cache bleaching when page coloring is used to partition the LLC.

Realizing the importance of data sharing between real-time tasks, authors in [9], [37] proposed techniques to enable data sharing between real-time tasks using producer/consumer buffers and wait-free buffers implemented in the LLC and main-memory. However, in order to eliminate the interference caused by back-invalidations, these works impose constraints on their caching behavior such as locking the buffers in the LLC (in order to prevent their eviction and hence, back invalidation) and disallow caching of buffers altogether in the cache hierarchy. On the other hand, ZCLLC takes a novel approach in eliminating back-invalidations without imposing any data caching constraints.

X. Conclusion

In this paper, we present ZCLLC, a shared inclusive LLC that does not incur additional cost to the WCL of memory requests when added to the memory hierarchy without an LLC. ZCLLC can be shared and does not suffer from the limitations of LLC partitions. Our results show that ZCLLC preserves the performance benefit of ZIV while showing a performance benefit of up to 25% compared to the state-of-the-art technique of sharing LLC partitions.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Z. Wu, A. M. Kaushik, and H. Patel, “ZeroCost-LLC: Shared LLCs at No Cost to WCL,” in proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2023, pp. 1–11.

REFERENCES

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, “A comprehensive survey of industry practice in real-time systems,” *Real-Time Systems*, nov 2021.

[2] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, “Multi-core devices for safety-critical systems: A survey,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–38, 2020.

[3] V. Suhendra and T. Mitra, “Exploring locking & partitioning for predictable shared caches on multi-cores,” in *Proceedings of the 45th annual Design Automation Conference*, 2008, pp. 300–303.

[4] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, “Making shared caches more predictable on multicore platforms,” in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 157–167.

[5] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, “NCID: A Non-Inclusive Cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies,” ser. CF ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 121–130.

[6] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, “Time-analysable non-partitioned shared caches for real-time multicore systems,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[7] Z. Wu and H. Patel, “Predictable Sharing of Last-Level Cache Partitions for Multi-Core Safety-Critical Systems,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1273–1278.

[8] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, “A Survey on Cache Management Mechanisms for Real-Time Embedded Systems,” vol. 48, no. 2, nov 2015.

[9] Chisholm, Micaiah and Kim, Namhoon and Ward, Bryan C. and Otterness, Nathan and Anderson, James H. and Smith, F. Donelson, “Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*, vol. , no. , 2016, pp. 57–68.

[10] El-Sayed, Nosayba and Mukkara, Anurag and Tsai, Po-An and Kasture, Harshad and Ma, Xiaosong and Sanchez, Daniel, “KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, vol. , no. , 2018, pp. 104–117.

[11] M. Chaudhuri, “Zero Inclusion Victim: Isolating Core Caches from Inclusive Last-level Cache Evictions,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 71–84.

[12] A. M. Kaushik, M. Hassan, and H. Patel, “Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems,” *IEEE Transactions on Computers*, pp. 1–1, 2020.

[13] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, “Shedding the Shackles of Time-Division Multiplexing,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 456–468.

[14] F. Hebbache, F. Brandner, M. Jan, and L. Pautet, “Work-Conserving Dynamic Time-Division Multiplexing for Multi-Criticality Systems,” *Real-Time Syst.*, vol. 56, no. 2, p. 124–170, apr 2020.

[15] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, “CARP: A Data Communication Mechanism for Multi-core Mixed-Criticality Systems,” in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 419–432.

[16] Renesas, “RH850/C1M-AX.” [Online]. Available: https://www.renesas.com/

[17] NXP Semiconductors, “Ultra-reliable MPC574XB/c/G mcus for automotive and industrial control and Gateway.” [Online]. Available: https://www.nxp.com/

[18] S. Srinivasan and W. L. Walker, “Shadow tag memory to monitor state of cachelines at different cache level,” 2018, US Patent 10,073,776.

[19] M. Chaudhuri, “Zero Directory Eviction Victim: Unbounded Coherence Directory and Core Cache Isolation,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 277–290.

[20] W. Zhang, M. Lv, W. Chang, and L. Ju, “Precise and Scalable Shared Cache Contention Analysis for WCET Estimation,” in *In proceedings of Design Automation Conference (DAC)*, 2022, pp. 1–6.

[21] H. Kim and R. R. Rajkumar, “Predictable Shared Cache Management for Multi-Core Real-Time Virtualization,” *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, dec 2017.

[22] A. M. Kaushik and H. Patel, “A Systematic Approach to Achieving Tight Worst-Case Latency and High-Performance Under Predictable Cache Coherence,” in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 105–117.

[23] S. Hessien and M. Hassan, “The Best of All Worlds: Improving Predictability at the Performance of Conventional Coherence with No Protocol Modifications,” in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 218–230.

[24] A. Bansal, J. Singh, Y. Hao, J.-Y. Wen, R. Mancuso, and M. Caccamo, “Reconciling Predictability and Coherent Caching,” in *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, 2020, pp. 1–6.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator,” vol. 39, no. 2, p. 1–7, aug 2011.

[27] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, “Splash-3: A properly synchronized benchmark suite for contemporary research,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[28] “Zcllc.” [Online]. Available: https://github.com/caesr-uwaterloo/ZCLLC

[29] D. Hardy and I. Puaut, “WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches,” in *2008 Real-Time Systems Symposium*. IEEE, Nov. 2008.

[30] ——, “WCET Analysis of Instruction Cache Hierarchies,” vol. 57, no. 7, p. 677–694, aug 2011.

[31] B. Lesage, D. Hardy, and I. Puaut, “WCET Analysis of Multi-Level Set-Associative Data Caches,” in *9th International Workshop on Worst-Case Execution Time Analysis (WCET’09)* , ser. OpenAccess Series in Informatics (OASIcs), N. Holsti, Ed., vol. 10. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 1–12, also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

[32] Z. Zhang, Z. Guo, and X. Koutsoukos, “Handling Write Backs in Multi-Level Cache Analysis for WCET Estimation,” ser. RTNS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 208–217.

[33] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt, “Cache behavior prediction by abstract interpretation,” *Science of Computer Programming*, vol. 35, no. 2, pp. 163–189, 1999.

[34] Altmeyer, Sebastian and Douma, Roeland and Lunniss, Will and Davis, Robert I., “OUTSTANDING PAPER: Evaluation of Cache Partitioning for Hard Real-Time Systems,” in *2014 26th Euromicro Conference on Real-Time Systems*, vol. , no. , 2014, pp. 15–26.

[35] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, “Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches.” ACM, 1994, p. 158–170.

[36] Y. Ye, R. West, Z. Cheng, and Y. Li, “COLORIS: A dynamic cache partitioning system using page coloring,” in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, pp. 381–392.

[37] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson, and F. D. Smith, “Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12.

[38] Intel, “Improving real-time performance by utilizing cache allocation technology,” *Intel Corporation*, 2015.

[39] S. Srikantaiah, M. Kandemir, and M. J. Irwin, “Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors,” *SIGARCH Comput. Archit. News*, vol. 36, no. 1, p. 135–144, mar 2008.

[40] S. Roozkhosh and R. Mancuso, “The Potential of Programmable Logic in the Middle: Cache Bleaching,” in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 296–309.