Predictable Sharing of Last-level Cache Partitions for Multi-core Safety-critical Systems

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

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

ABSTRACT

Last-level cache (LLC) partitioning is a technique to provide temporal isolation and low worst-case latency (WCL) bounds when cores access the shared LLC in multicore safety-critical systems. A typical approach to cache partitioning involves allocating a separate partition to a distinct core. A central criticism of this approach is its poor utilization of cache storage. Today's trend of integrating a larger number of cores exacerbates this issue such that we are forced to consider shared LLC partitions for effective deployments. This work presents an approach to share LLC partitions among multiple cores while being able to provide low WCL bounds.

KEYWORDS

Last-level cache, Predictability, Cache partitioning

1 INTRODUCTION

Multicores in safety-critical systems offers an attractive opportunity to consolidate several functionalities onto a single platform with the benefits of reducing cost, size, weight, and power while delivering high performance. Although multicores are mainstay in general-purpose computing, their use in safety-critical systems is approached with caution. This is because multicores often share hardware resources to deliver their high performance, but since safety-critical systems must be certified, this makes guaranteeing compliance with safety standards increasingly challenging. The central reason behind this difficulty is that shared resources complicate worst-case timing analysis necessary for applications deemed to be of high criticality. For instance, the automotive domain uses the ISO-26262 standard, which identifies ASIL-D as the highest criticality application where a violation of its temporal behaviours may result in a significant loss of lives or injury.

One such shared hardware resource is the shared last-level cache (LLC) that multiple cores access when they experience misses in their private caches. For example, the Kalray MPPA 3 [2] has a 80 cores with 16 cores in a cluster that share 4MB of LLC. LLCs are essential in delivering high performance. However, multiple cores accessing the LLC introduces inter-core temporal interference where one core evicts the data of another's. These interferences complicate worst-case latency (WCL) analysis, and result in overly pessimistic worst-case bounds. LLC partitioning is a countermeasure to address these difficulties [3, 6, 7]. LLC partitioning allocates a part of the LLC to each core that it can use. This provides temporal isolation to tasks executing on a core from execution of other tasks on other cores. However, there are multiple downsides to LLC partitioning: (1) it can significantly affect average-case performance, (2) it can lead to underutilization of cache capacity, and (3) prevent coherent data sharing [1]. Downside (1) is a result of each core having a smaller part of the LLC. (2) happens when a core gets allocated a partition that it does not effectively use. For (3), conventional LLC partitioning disallows one core to access another core's partition; thus, coherent caching of shared data in the LLC is prohibited. With the continued increase in demand for functionalities, and their consolidation onto a multicore platform, we expect these downsides to overwhelm the benefits of LLC partitioning. This forces us to seriously consider sharing LLC partitions.

As a cautious step towards addressing these downsides, and possibly a refreshing alternative to traditional LLC partitioning approaches, we allow multiple cores to share partitions. This requires us to determine the WCL of memory accesses from cores that miss in their private caches, and access the shared partition. In this paper, we develop this WCL analysis. In doing so, we show that naively arbitrating cores' accesses to the shared LLC partitions results in a scenario where the WCL is unbounded. We correct this unbounded scenario by using a restricted version of time-division multiplexing (TDM) policy called 1S-TDM that results in a WCL bound. However, the resulting WCL bound is grossly pessimistic; it is proportional to the minimum of the cache capacity and LLC partition size of a given core and cube of the number of cores. By analyzing the critical instance that renders the WCL bound, we intuit a technique to significantly lower the WCL. This technique yields a WCL that eliminates the dependency on the cache and partition sizes. For a 4-core setup with a 16-way LLC with 128 cache lines, our approach results in a WCL that is 2048 times lower. We implement this technique in a hardware structure called the set sequencer. We show that careful sharing of cache partitions not only allows for a low WCL, but possibly higher average-case performance. We envision the proposed work to complement existing efforts on LLC partitioning where certain tasks have their own partitions, but others share partitions; all of which depends on their performance and real-time requirements. The following are our main contributions.

- We identify that naively using TDM to arbitrate accesses to the shared LLC partitions can result in an unbounded WCL. We establish that a 1S-TDM schedule prohibits this scenario.
- We develop a WCL analysis for a memory access to a shared LLC partition using the 1S-TDM arbitration policy.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530614

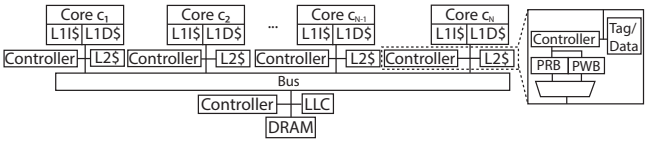


Figure 1: Overview of system model.

- We propose a hardware extension called the set sequencer that lowers the WCL when sharing a LLC partition. We evaluate the set sequencer using simulation.

2 RELATED WORKS

Cache partitioning [3] reserves a portion of the cache to a task or a core either via hardware or software techniques [10]. The key role of cache partitioning is to improve temporal isolation to simplify the WCL analysis. However, as the number of cores increase, allocating distinct partitions to each core or each task can affect average-case performance and cache under-utilization. Moreover, this prohibits deploying a large number of functionalities on the multicore as it may result in extremely small partitions to each functionality, which would adversely affect performance. The proposed work seeks a middle ground where designers can judiciously share partitions with a subset of cores, and isolate others. Prior works identified that accurately capturing the contention of multiple cores sharing the LLC is difficult [8], and attempts exist for shared cache for dual-core processor relying on knowledge of the application [9]. Our work does not rely on application-specific knowledge and does not constraint the number of cores. Our work also applies when all cores share the whole LLC which serves as a single partition.

Authors in [5] noticed that interference between tasks exists when multiple tasks or cores share the LLC. They proposed a time analysable shared LLC where the inter-task interference is bounded probabilistically. Given their proposed technique, the exact worst-case interference in the LLC and the exact worst-case latency is not discernible. Compared to [5], our analysis provides an exact (non-probabilistic) bound of a memory access in the LLC, and does not rely on MBPTA. Our work assumes that a TDM bus arbitration has one slot per core in each period, which is common in controlling access to resources in safety-critical systems [1, 4]. We also identified the worst-case scenario for the LLC evictions, and that in the worst-case the latency can be unbounded if the arbitration has no constraint. Moreover, our analysis does not rely on certain type of address mapping or replacement policy.

3 SYSTEM MODEL

System hierarchy. We assume a system with N cores and three levels of cache in the memory hierarchy (Figure 1). Each core has a private L1 instruction cache (L1I$) and a private L1 data cache (L1D$). The private L1 caches of a core are connected to a L2 cache (L2$). A set-associative last-level cache (LLC) connects to all L2 caches, and interfaces with a DRAM directly. The accesses from L2 to LLC is controlled by a shared bus. The bus implements a time-division multiplexing (TDM) arbitration which allocates slots of width SW to cores. Table 1 shows the symbols used throughout the paper. SW is long enough for a data transfer between the L2$ and the LLC. The L2 cache controller of each core only accesses the LLC in the core's allocated slot, and the LLC only responds to the L2 within the core's slot. Since the L2 cache is private to each core, a core accessing the bus implies that the core's L2 cache controller access the bus. An LLC partition isolates a part of the LLC for a specific core. One task can be mapped to one core.

Table 1: Symbols.

Symbol	Explanation	Symbol	Explanation
c_i, c_{ua}	Core i, core under analysis	$d^{c_i}_{c_j}, d^{c_i(l)}_{c_j}(x)$	Distance between cores
l_i, l	Cache lines in LLC set	SW	Slot width
N, n	# cores, # cores sharing a partition	$set_{LLC}(X)$	Cache set in the LLC that X maps to
$s^t_{c_{ua}}$	Start of the t-th slot of core c_{ua}	$m_{c_{ua}}, M$	Cache capacity of c_{ua}/LLC partition
w	# of ways in a partition	$\mathcal{P}$	Cache partition

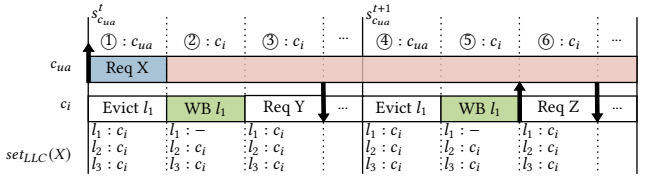


Figure 2: c_{ua} experiences an unbounded latency.

Similar to prior works [1], we assume each core has at most one outstanding memory request. Before a core's request or write-back is placed on the bus, the request is buffered in a pending request buffer (PRB), and the write-backs are buffered in a pending write-back buffer (PWB). A predictable arbitration such as round-robin between PRB and PWB chooses from a request or a write-back to send on the bus at the beginning of the core's slot.

Cache inclusion policy. We assume that the LLC is inclusive of L2. This is a common setup in existing platforms. Suppose that core c_i's request to the cache line at address X is a miss in all its private caches, and the LLC. Then, for the LLC to respond with the provided data for X, the LLC must ensure: (1) that there is a vacant entry in the set that cache line X maps to, (2) the cache line from a lower level memory in the memory hierarchy is fetched, and (3) the response to c_i is sent in c_i's slot. An important property of inclusive caches is that an eviction of a cache line in the lower-level cache requires eviction of cache lines for the same address in upper-level caches. For our setup, an eviction in the LLC forces evictions in both the L1 and L2 private caches that have the data.

4 WCL WITH SHARED PARTITIONS

4.1 An unbounded WCL scenario

We show that an undesired consequence of the inclusive property, and multiple cores making accesses to the LLC is a situation where the WCL is unbounded. Using Figure 2, we illustrate this scenario. We assume a TDM arbitration policy with one slot for c_{ua} and two slots for c_i. We use $s^t_{c_i}$ to denote the starting time of the t-th slot of core c_i. Unambiguously, $s^t_{c_i}$ also refers to the t-th slot of c_i. At $s^t_{c_{ua}}$, in ①, c_{ua}'s request to a cache line X misses in the private caches and the LLC. Cache line X is mapped to a full cache set $set_{LLC}(X)$ in the LLC; thus, the LLC evicts a cache line l_1 in $set_{LLC}(X)$, which is also privately cached by c_i denoted by $l_1 : c_i$. Note that a full cache set means that there are no empty cache lines in the set. In ②,

c_i writes back l_1, and frees an entry in $set_{LLC}(X)$. Then, in ③, c_i requests a cache line mapped to $set_{LLC}(X)$, and gets the response within its slot because l_1 is available. In c_{ua}'s next slot $s_{c_{ua}}^{t+1}$ (④), $set_{LLC}(X)$ is full again preventing c_{ua} from completing its request. This behavior can potentially repeat indefinitely, which shows that the interference at the LLC from other cores can cause unbounded WCL for the core under analysis. Note that allowing the write-back within the same slot as the request does not mitigate the issue because another core can issue a request between the write-back and the response, occupying the freed entry.

4.2 One-slot TDM schedule

The scenario where c_{ua} experiences unbounded latency happens when a core other than c_{ua} is allowed to access the LLC multiple times before c_{ua} can access the bus again. An effective solution to prevent another core from occupying a free entry in $set_{LLC}(X)$ is to constrain the TDM schedule to allocate only one slot per core in a period (Definition 4.1). Note that the crux of the issue is that an entry freed by c_i due to the eviction of cache line l_1 is occupied again by c_i again *before* c_{ua} can access the LLC. With 1S-TDM, we only allow one core to perform one access to the bus in a period.

Definition 4.1. A *one-slot TDM schedule* (1S-TDM) is a TDM schedule that has exactly one slot allocated to each core in every period.

4.3 Key observations

Although a 1S-TDM schedule guarantees a WCL bound for c_{ua}, we show that the WCL is proportional to the minimum of the cache capacity of c_{ua} and c_{ua}'s LLC partition size M, and cube of the number of cores. This results in a significantly large WCL. We illustrate this by making observations from two examples.

Consider the example in Figure 3 that has four cores, and a two-way set-associative LLC. The 1S-TDM schedule is $\{c_{ua}, c_2, c_3, c_4\}$. In ①, c_{ua} requests cache line X, which is not privately cached by c_{ua}. Hence, its L2 cache controller issues request for X at the beginning of $s_{c_{ua}}^{t}$ to the LLC. Cache line X maps to a cache set $set_{LLC}(X)$ in the LLC, and it also experiences a miss in the LLC. Since $set_{LLC}(X)$ has no empty lines, it must evict one cache line in $set_{LLC}(X)$. Suppose that $l_1 \in set_{LLC}(X)$ is selected for eviction. Note that l_1 is privately cached in c_3 denoted by $l_1 : c_3$. Hence, c_3 must also evict l_1 from its private caches in ②, resulting in a free entry in $set_{LLC}(X)$, denoted as $l_1 : -$. Clearly, the cache line to replace depends on the replacement policy. In this work, we assume a replacement policy that can select any of the cache lines. Even though we select l_1 in this example, our observation is agnostic of replacement policy. As a result, the observation applies to any replacement policy, including least-recently used (LRU).

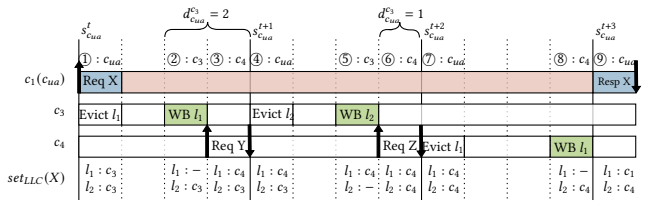


Figure 3: c_{ua}'s request to cache line X eventually is completes.

Next, in ③, c_4's L2 cache controller issues a request, and occupies the free entry, preventing c_{ua} from obtaining its response in slot $s_{c_{ua}}^{t+1}$. As a result, the LLC must evict a cache line in $set_{LLC}(X)$ to make space for c_{ua}, and the LLC evicts l_2 in slot $s_{c_{ua}}^{t+1}$ which is privately cached by c_3. As before, c_3 evicts l_2 in ⑤, but it is occupied by c_4 in ⑥. Note that in the period starting at $s_{c_{ua}}^{t+2}$, this pattern of interfering c_{ua} from receiving its response cannot continue. When c_4 evicts l_1 in ⑧, there is no other core that can occupy the free entry; thus, c_{ua} will get its response in $s_{c_{ua}}^{t+3}$ at ⑨. This is guaranteed to occur because whenever any core other than c_{ua} occupies a free entry in $set_{LLC}(X)$, c_{ua} gets *closer* to being able to occupy a free entry in $set_{LLC}(X)$. For example, in ⑦, both cache lines are privately cached by c_4, and any core making a request to $set_{LLC}(X)$ resulting in a miss requires c_4 to evict it from its private caches as well. Since, c_{ua}'s slot comes after c_4's, it is guaranteed to occupy the free cache line entry due to c_4's eviction. We characterize this *closer* effect by introducing a notion of distance (Definition 4.2).

Definition 4.2. For a 1S-TDM schedule S, the distance between two cores c_i and c_j, $d_{c_j}^{c_i}$, is the number of slots between the start of slot of c_i, and the start of c_j's next slot.

Corollary 4.3. *Given a 1S-TDM schedule S with N cores, the distance between any two cores c_i and c_j, $d_{c_j}^{c_i}$, $1 \leq d_{c_j}^{c_i} \leq N$.*

Given a cache line l, we will use $d_{c_j}^{c(l)}(x)$ as a convenience to return the distance between the core that has privately cached l, and c_j at x. Using Figure 3, with a TDM schedule of $\{c_{ua}, c_2, c_3, c_4\}$, $d_{c_{ua}}^{c_3} = 2$ and $d_{c_{ua}}^{c_4} = 1$. With Definition 4.2, the example in Figure 3 can be interpreted in terms of distance: the core that caches l_1 changes from c_3 in slot ①, with a distance of $d_{c_{ua}}^{c_3} = 2$, to c_4 in slot ③, with a distance of $d_{c_{ua}}^{c_4} = 1$, and finally freed in slot ⑧. Similarly, the core that caches l_2 changes from c_3 in slot ① to c_4 in slot ⑥, and thus the distance of the core that caches l_2 decreases from 2 to 1. These examples highlight the following key observations.

Observation 1. Given a 1S-TDM schedule S, the *distance* of cache lines in $set_{LLC}(X)$ decrease when c_{ua} does not perform write-backs after issuing its request to cache line X and before receiving its response for X.

The decreasing distance articulates the effect of the core under analysis getting *closer* to occupying a freed cache line entry in $set_{LLC}(X)$. A direct consequence of observation 1 is that c_{ua}'s request will eventually complete as described in observation 2.

Observation 2. c_{ua}'s request will eventually complete.

The main intuition behind this observation is that once the lines in $set_{LLC}(X)$ are privately cached by c_4, a request for X by c_{ua} will succeed in the following period (Figure 3). This is because c_4 must evict the privately cached line due to inclusive property, which results in a free entry in $set_{LLC}(X)$ that c_{ua} can occupy. When $n \leq N$ cores share a partition with a 1S-TDM schedule and there are w ways in $set_{LLC}(X)$, for c_{ua} to occupy an entry in $set_{LLC}(X)$ in the worst-case, the distances of all w cache lines must experience the largest decrements. Since the maximal distance is n when c_{ua} caches a cache line, and the minimal distance is 1, c_{ua} must wait for the distances of all w cache lines to decrease from n to 1, accounting for $w(n-1)$ decrements in the worst-case.

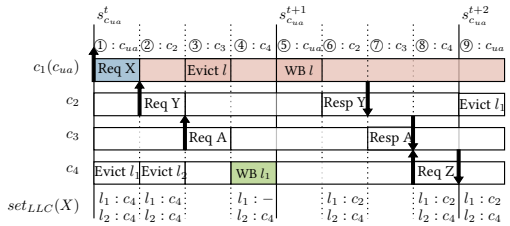


Figure 4: Distance of core caching l_1 increases after $s^{t+1}_{c_{ua}}$.

Note that we have not considered scenarios where c_{ua} performs write-backs before receiving the response for its request. When a core other than c_{ua} requests a cache line that is privately cached by c_{ua}, then c_{ua} would also need to perform write-backs due to inclusivity. The effect of write-backs on the distance is summarized in observation 3. After a write-back by c_{ua}, cache lines in $set_{LLC}(X)$ can be privately cached by a core with a larger distance compared to before c_{ua} performs the write-back.

Observation 3. Given a 1S-TDM, when c_{ua} performs a write-back after issuing its request to a cache line X and before receiving the response, the distances of cache lines in $set_{LLC}(X)$ increases.

Figure 4 shows a scenario with c_{ua} performing write-backs. There are four cores with 1S-TDM schedule of $\{c_{ua}, c_2, c_3, c_4\}$. Cache lines l_1 and l_2 are in $set_{LLC}(X)$, and are initially privately cached by c_4. In ①, c_{ua} issues a request to cache line X that misses in the private caches and LLC. This is followed by c_2 issuing a request to cache line Y such that $Y \in set_{LLC}(X)$ in ②, and it also misses in the private caches and LLC. The LLC selects to evict another line l_2, which needs to be evicted by c_4. In ③, c_3 issues a request to cache line A that causes c_1 to evict a cache line l. In ④, c_4 writes back l_1 freeing up an entry in $set_{LLC}(X)$. In the write-back slot of $s^{t+1}_{c_{ua}}$ ⑤, c_1's request to X cannot be satisfied because c_1 has to perform an eviction. The free entry is thus occupied by c_2. Note that the core that is caching l_1 has changed from c_4 to c_2 and thus the distance of the core caching l_1 increased from $d^{c_4}_{c_1} = 1$ to $d^{c_2}_{c_1} = 3$. In general, write-backs from c_{ua} allow a core with a larger distance to occupy a free entry in $set_{LLC}(X)$ that would have satisfied c_{ua}'s request. Therefore, the distance does not always decrease as in the case of Observation 1 when c_{ua} performs write-backs. We combine these two observations to develop a WCL analysis of a request. Henceforth, we assume the worst-case replacement policy: the LLC always replaces line with maximal distance for generality. Replacement policies such as LRU may tighten the bound but only by a constant factor.

4.4 WCL analysis for 1S-TDM schedule

We prove that the distances in $set_{LLC}(X)$ only decrease when c_{ua} does not perform a write-back in Corollary 4.5 (Observation 1). We bound the latency required for the distance to decrease. In Lemma 4.6, we show that the distance increases when c_{ua} writes back cache lines (Observation 2). Hence, when c_{ua} waits for its response, the distances in $set_{LLC}(X)$ show a pattern of decreasing and increasing. Finally, Theorem 4.8 combines Corollary 4.5 and Lemma 4.6 to express the WCL of c_{ua}'s request. We use our key observations to formulate an analysis to compute the WCL. Consider a multicore configuration with N cores interacting over the shared bus using a 1S-TDM schedule S, and n cores sharing a partition $\mathcal{P}$ in the LLC ($n \leq N$) with c_{ua} being one of the n cores. Throughout the analysis, we assume that c_{ua}'s request for cache line X misses in its private caches and the LLC, and $set_{LLC}(X)$ is full before c_{ua}'s request to cache line X is completed.

Lemma 4.4. *If c_{ua}'s request is not completed at slot $s^{t+T}_{c_{ua}}$, c_{ua} does not perform any write-backs, and $l_x \in set_{LLC}(X)$ is evicted in response to c_{ua}'s request in $s^t_{c_{ua}}$, then*

$$\forall l \in set_{LLC}(X) : \; d^{c(l)}_{c_{ua}}(s^{t+T}_{c_{ua}}) \leq d^{c(l)}_{c_{ua}}(s^t_{c_{ua}}).$$

Proof. We prove the lemma by contradiction and assume that $\exists l \in set_{LLC}(X) : d^{c(l)}_{c_{ua}}(s^{t+T}_{c_{ua}}) > d^{c(l)}_{c_{ua}}(s^t_{c_{ua}})$. Then, before $s^{t+T}_{c_{ua}}$, there must exist two cores c_i and c_j, such that c_i frees the entry l and c_j occupies l after c_i frees l. The freeing-then-occupying by c_i and c_j increases the distance of l to be greater than $d^{c(l_x)}_{c_{ua}}(s^t_{c_{ua}})$, that is, $d^{c_i}_{c_{ua}} \leq d^{c(l_x)}_{c_{ua}}(s^t_{c_{ua}}) < d^{c_j}_{c_{ua}}$. Assume that c_i frees l in $s^q_{c_i}$ and c_j occupies l in $s^r_{c_j}$, then $s^q_{c_i} < s^r_{c_j}$. Furthermore, because $d^{c_i}_{c_{ua}} < d^{c_j}_{c_{ua}}$ and 1S-TDM is deployed, $s^r_{c_j}$ must be in the next period of $s^q_{c_i}$. There must be a slot of c_{ua}, $s^p_{c_{ua}}$, such that $s^q_{c_i} < s^p_{c_{ua}} < s^r_{c_j}$. Thus, there is a free entry in $set_{LLC}(X)$ in slot $d^p_{c_{ua}}$. Because c_{ua}'s request is not completed in slot $s^{t+T}_{c_{ua}}$, it is not completed in slot $d^p_{c_{ua}}$. The only reason that c_{ua}'s request is not completed in its slot when there is a free entry is that c_{ua} is performing a write-back, which contradicts the hypothesis that c_{ua} does not perform any write-backs. □

Corollary 4.5. *If c_{ua} does not perform any write-backs, $l_x \in set_{LLC}(X)$ is evicted in response to c_{ua}'s request in $s^t_{c_{ua}}$ and c_{ua}'s request is not completed in $s^{t+2(n-1)}_{c_{ua}}$ then*

$$d^{c(l_x)}_{c_{ua}}(s^{t+2(n-1)}_{c_{ua}}) < d^{c(l_x)}_{c_{ua}}(s^t_{c_{ua}})$$

Proof. Assume that at $s^t_{c_{ua}}$, cache line l_x is evicted, but it is privately cached by c_i such that $d^{c_i}_{c_{ua}} = d^{c(l_x)}_{c_{ua}}(s^t_{c_{ua}})$. At a later slot for c_i, $s^q_{c_i}$ where l_x is written back, $q \leq t + 2(n-2) + 1 < t + 2(n-1)$. This is because there can be at most $(n-1)$ pending write-backs in c_i's PWB including the write-back for l_x. Before c_{ua}'s next slot, another core c_j must occupy l_x so that c_{ua}'s request is not completed. Due to 1S-TDM, if c_j occupies l_x, $d^{cj}_{c_{ua}} < d^{c_i}_{c_{ua}} \leq d^{c(l_x)}_{c_{ua}}(s^t_{c_{ua}})$. Applying Lemma 4.4, $s^{t+2(n-1)}_{c_{ua}} \leq d^{cj}_{c_{ua}} < d^{c(l_x)}_{c_{ua}}(s^t_{c_{ua}})$. □

Lemma 4.6. *Given a slot $s^t_{c_{ua}}$ where c_{ua} performs write-back, then there exists an execution such that*

$$\forall l \in set_{LLC}(X) : d^{c(l)}_{c_{ua}}(s^{t+1}_{c_{ua}}) \geqslant d^{c(l)}_{c_{ua}}(s^t_{c_{ua}}).$$

Proof. In $s^t_{c_{ua}}$, let us assume that c_{ua} writes back a cache line as a response to an eviction caused by another core. Since c_{ua} is performing a write-back, it cannot issue a request; thus, its request cannot complete. Hence, for each of the free cache line entry l in $set_{LLC}(X)$ at $s^t_{c_{ua}}$, a core c_j can make a request to l after $s^t_{c_{ua}}$ which completes within one slot. Then, $d^{c(l)}_{c_{ua}}(s^{t+1}_{c_{ua}}) = d^{c_j}_{c_{ua}} > d^{c(l)}_{c_{ua}}(s^t_{c_{ua}})$. For other cache lines $l' \in set_{LLC}(X)$ that are privately cached by

other cores that are not evicted due to accesses made by some other cores, $d_{c_{ua}}^{c(l)}(s_{c_{ua}}^{t+1}) = d_{c_{ua}}^{c(l)}(s_{c_{ua}}^{t})$ holds trivially. □

Corollary 4.5 and Lemma 4.6 provide the cornerstone to derive the worst-case latency for c_{ua} in Theorem 4.7 and Theorem 4.8

Theorem 4.7 (critical instance). *Let $m = \min(m_{c_{ua}}, M)$, where $m_{c_{ua}}$ is the cache capacity of c_{ua}. The worst-case memory latency occurs when c_{ua} issues a request to line X such that: (1) before c_{ua} receives its response, all privately cached lines in c_{ua} are written back; (2) after each of these write-backs by c_{ua}, the distances of all cache lines in $set_{LLC}(X)$ increase to n; (3) before each of these write-backs by c_{ua}, the distances of all cache lines in $set_{LLC}(X)$ decrease from n to 1; and, (4) after the last write-back by c_{ua}, the distances of all cache lines in $set_{LLC}(X)$ decrease from n to 1.*

Proof. We prove the critical instance by construction. Suppose c_{ua} makes a request to X that maps to $set_{LLC}(X)$. Before c_{ua} receives its response, multiple write-backs from c_{ua} can occur in between as a result of other cores making requests. The maximal number of such write-backs, and the conditions under which they experience the maximum number of distance changes determines the critical instance. We have shown that the distances of lines in $set_{LLC}(X)$ only decrease between two write-backs while increases only when a write-back happens. The critical instance thus maximizes the number of distance changes between write-backs by means of *maximizing* the four parts: (1) The number of write-backs c_{ua} can perform in the worst-case as a result from other cores' requests. (2) The number of distance changes between any two write-backs by c_{ua}. (3) The number of distance changes before the first write-back of c_{ua}. (4) The number of distance changes after the last write-back of c_{ua} until it receives its response.

For (1), cores other than c_{ua} can force $m = \min(m_{c_{ua}}, M)$ write-backs for c_{ua}. This happens when m lines are privately cached by c_{ua}, and other cores make accesses that happen to map to the same sets as those m lines, which forces c_{ua} to write them back due to inclusion. This amounts to the maximal number of lines c_{ua} can cache in the partition $\mathcal{P}$. For (2), distance for a cache line can increase and decrease. Lemma 4.6 shows that after a write-back, $d_{c_{ua}}^{c(l)}$ is n in the worst-case for all $l \in set_{LLC}(X)$. Note that the distance cannot exceed n because this is the maximal number of cores that share $set_{LLC}(X)$ in $\mathcal{P}$. Lemma 4.4 ensures that $d_{c_{ua}}^{c(l)}$ strictly decreases and will not increase because of the absence of write-backs in between. In the worst-case, all $d_{c_{ua}}^{c(l)}$ decrease from n down to 1. For (3), in the worst-case, the distances of all w cache line entries in $set_{LLC}(X)$ are n when c_{ua} issues its request, and each distance decreases from n down 1 *before the first write-back* in the worst-case. Note that the distance cannot increase due to Lemma 4.4 and the absence of intermediate write-backs. Similar to (3), for (4), *after the last write-back*, the distances of all w cache line entries in $set_{LLC}(X)$ decrease from n down 1 before c_{ua} receives its response followed with one slot for c_{ua} to receive its response. □

Theorem 4.8. *Let $m = \min(m_{c_{ua}}, M)$, where $m_{c_{ua}}$ is the cache capacity of c_{ua}. The worst-case latency in number of slots of the request of the core under analysis c_{ua}, WCL, is given by:*

$$WCL = \big((m+1) \cdot A \cdot N + 1\big) \cdot SW, \tag{1}$$

where $A = 2(n-1) \cdot w \cdot (n-1)$.

Proof. Corollary 4.5 manifests $d_{c_{ua}}^{c(l)}$ decreases by 1 every $2(n-1)$ periods in the worst-case for $l \in set_{LLC}(X)$. Between two out of $(m-1)$ write-backs from c_{ua}, all w cache lines' distances decrease from n to 1, taking $A = 2(n-1) \cdot w(n-1)$ periods. Before the first and after the last write-back, the distances of all w cache lines decrease from n to 1. c_{ua} receives its response in one slot. □

4.5 Set sequencer: Lowering the WCL

We propose a hardware extension called a set sequencer that significantly lowers the WCL. Recall that the WCL for a core under analysis c_{ua} is proportional to the minimum of either the cache capacity or c_{ua}'s LLC partition size, and cube of the number of cores. This is clearly a large bound. When using the set sequencer, the WCL bound is *independent* of the minimum of the c_{ua}'s cache capacity and c_{ua}'s LLC partition size M. The set sequencer has two structures: a Queue Lookup Table (QLT) and a Sequencer (SQ). The set sequencer contains one entry in the QLT for each set in the partition that has at least one pending LLC request. The entry maps the set to a queue in SQ. When there are multiple cores requesting a cache line from the same set, set sequencer stores the order in which the requests arrived at the LLC (broadcast order on the shared bus). Our key observations and the WCL analysis revealed that the distance increases whenever c_{ua} is prevented from occupying a free cache line entry due to another core intercepting it. By maintaining order using the set sequencer, we can guarantee that does not happen. Since the set sequencer only tracks outstanding memory requests for each core, QLT and SQ each contain up to N entries, respectively. Each QLT entry tracks the set index, the queue index, and each queue entry in SQ tracks the core index. We present the WCL when using the set sequencer in Theorem 4.9.

Theorem 4.9. *The WCL of a request of the core under analysis c_{ua} when using the set sequencer, WCL_{ss}, is given by:*

$$WCL_{ss} = \big(2(n-1) \cdot n + 1\big) \cdot N \cdot SW. \tag{2}$$

Proof. In the worst-case, all other $(n-1)$ cores issue their request before c_{ua} sends its request to cache line X to the LLC, and it is the last request in the set sequencer for a full set $set_{LLC}(X)$. For each request in the set sequencer, including c_{ua}, it takes $2(n-1)$ slots for the core caching cache lines in $set_{LLC}(X)$ to write back a cache line and free an entry as a core performs $(n-1)$ write-backs in the worst-case, and the evicted cache line is written-back last.

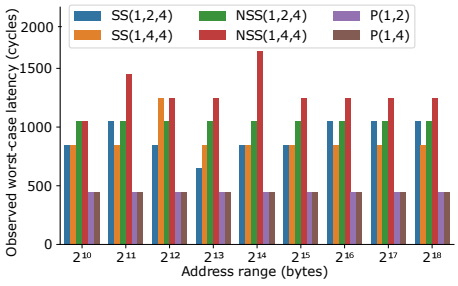
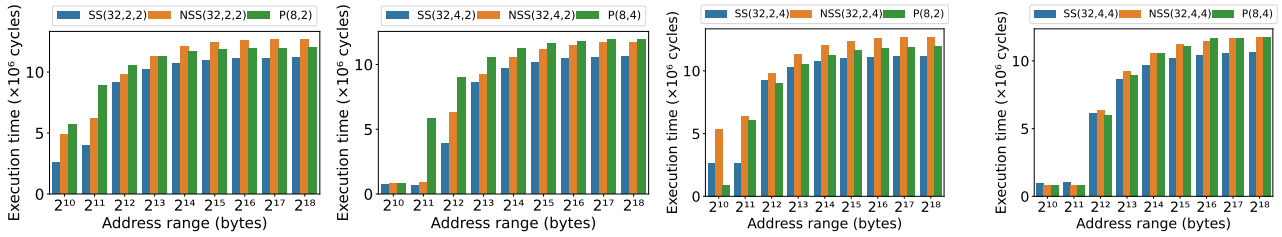


Figure 5: The observed WCL of SS, NSS and P.



(a) 2-core, 4096B partition. **(b) 2-core, 8192B partition.** **(c) 4-core, 4096B partition.** **(d) 4-core, 8192B partition.**

Figure 6: The execution time of synthetic workload with fixed total partition size.

Note that each such slot accounts for one period, which is $N \cdot SW$ Finally, c_{ua} requires one slot SW to receive its response, which accounts for another period. □

5 EVALUATION

Our empirical evaluation is performed with an in-house trace simulator that simulates the cache subsystem of a four-core system with the memory hierarchy as described in section 3. The L2 cache is a 4-way set-associative cache with 16 sets and the L3 cache is a 16-way set-associative cache with 32 sets that can be partitioned across the four cores. The cache line size is 64-byte.

Workload generation. We use synthetic workloads consisting of memory requests to random addresses within various address ranges. Each synthetic workload assumes the same address range for all cores. The memory address of each core is randomly generated such that they fall in the address range. We then apply offsets to the addresses of these memory requests to exclude data sharing between any two cores. For a certain address range, a core issues the same memory addresses across different partitioned configurations.

Notation. We use the following syntax to express partitioned configurations. (1) `SS(s,w,n)`: a partition shared among n cores with s sets and w ways with set sequencer, (2) `NSS(s,w,n)`: a partition shared among n cores with s sets and w ways and LLC services contending requests with best effort, and (3) `P(s,w)`: a partition with s sets and w ways that is uniquely occupied by a core. For `P(s,w)`, each core is assigned equally-sized partition.

5.1 Worst-case latency

Workload setup. To exercise the worst-case, we enforce a partition size of one set for all configurations. This forces as many conflicts as possible.

Results. Figure 5 confirms that the observed WCL of all configurations are within the analytical WCLs, which are 5000 cycles for SS, 979250 cycles for NSS, and 450 cycles for P. Although a distinct partition P yields the lowest WCL, recall that we wish to share partitions for cores whose real-time requirements are met with sharing. However, there might be others that need distinct partitions P, which do indeed provide the lowest WCL. This is essential when the number of required functionalities deployed onto a single multicore increases. In the case of cores sharing a partition, the bound for SS can be employed. NSS shows a higher observed WCL compared to SS across all address ranges because distance can increase as mentioned in Observation 3.

5.2 Partition sharing and utilization

We evaluate the impact of partitioning when cores share a partition.

Workload setup. We use a 2-core and a 4-core setup with a fixed cache capacity that is partitioned. In SS and NSS, all cores share the same partition while in P, the fixed cache capacity is divided equally between all cores, and the set-associativity is fixed. Figure 6a shows that when the address range is 1024-byte or 2048-byte, the execution time is the same across SS, NSS and P. This is because the address range is less than or equal to the partition size.

When the address range exceeds the partition size, SS exhibits improved performance when compared to both NSS and P. In the 2-core setup with 4096-byte of partition size, SS achieves an average speedup of 1.34× as is shown in Figure 6a. When the capacity is 8192-byte, SS achieves an average speed up of 2.13× (Figure 6b). Such performance persists in the 4-core setup where SS features an average speedup of 1.10× for 4096-byte partition size (Figure 6c) and an average speed up of 1.02× for 8192-byte partition size (Figure 6d).

6 CONCLUSION

We present an approach to predictably share LLC partitions. We use a constrained TDM policy with a set sequencer hardware extension, which results in a predictable and low WCL. Our empirical evaluation showed that we could reduce the WCL by 2048 times compared to a naive approach with the set sequencer.

REFERENCES

[1] A. Kaushik et al. 2020. Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems. *In IEEE TC* (2020), 1–14.
[2] B. Dupont de Dinechin. 2019. Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore. In *ACM/IEEE DAC*. 1–4.
[3] G. Gracioli et al. 2015. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.* 48, 2, Article 32 (2015), 36 pages.
[4] H. Rihani et al. 2015. WCET Analysis in Shared Resources Real-Time Systems with TDMA Buses. In *RTNS*. 183–192.
[5] J. Cerrolaza et al. 2020. Multi-Core Devices for Safety-Critical Systems: A Survey. *ACM Comput. Surv.* 53, 4, Article 79 (2020), 38 pages.
[6] M. Lv et al. 2016. A Survey on Static Cache Analysis for Real-Time Systems. *Leibniz Transactions on Embedded Systems* 3, 1 (Jun. 2016), 05:1–05:48.
[7] S. Altmeyer et al. 2014. Evaluation of Cache Partitioning for Hard Real-Time Systems. In *ECRTS*. 15–26.
[8] V. Suhendra et al. 2008. Exploring locking amp; partitioning for predictable shared caches on multi-cores. In *DAC*. 300–303.
[9] W. Zhang et al. 2012. Static Timing Analysis of Shared Caches for Multicore Processors. *JCSE* 6, 4 (2012), 267–278.
[10] X. Wang et al. 2017. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *HPCA*. 121–132.