Invalidate or Update? Revisiting Coherence for Tomorrow's Cache Hierarchies

Citation for published version:
Zhu, M, Shahab, A, Katsarakis, A & Grot, B 2021, Invalidate or Update? Revisiting Coherence for Tomorrow's Cache Hierarchies. in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, pp. 226-241, 30th International Conference on Parallel Architectures and Compilation Techniques, 26/09/21. https://doi.org/10.1109/PACT52795.2021.00024

Digital Object Identifier (DOI):
10.1109/PACT52795.2021.00024

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)

In Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT'21)

Invalidate or Update?
Revisiting Coherence for Tomorrow's Cache Hierarchies

Mingcan Zhu, Amna Shahab, Antonios Katsarakis, Boris Grot
University of Edinburgh
FirstName.LastName@ed.ac.uk

***Abstract*—Shared on-chip last-level caches (LLCs) play a key role in capturing the large working sets of today's data-intensive workloads. However, they pose a fundamental scalability challenge in the transistor-limited post-Moore regime. Recent work has argued for Next-Generation LLCs (NG-LLC) based on private caches in die-stacked DRAM, which can provide hundreds of MBs of per-core LLC capacity at similar access latency to today's shared LLCs. While NG-LLCs offer a number of advantages, their private design exposes long-latency inter-core reads for read/write shared data, which hurt performance in parallel workloads. One way to eliminate the long latency of reads to read/write shared data is through the use of *updating coherence protocols* that eagerly push updates from a writer core into caches of recent readers. Alas, these protocols are known to generate excess cache and interconnect traffic that can be detrimental to overall performance. While hybrid protocols that try to alleviate the problem by combining invalidating and updating protocols have been proposed, we find their performance benefit to be small for NG-LLCs.**

This work observes that the number of writes to a read/write shared cache block is likely to be stable over several consecutive write/read iterations. Based on this insight, we propose the *1-Update* protocol that records the number of writes without an intervening read by a sharer, and subsequently uses the recorded value to send at most one update after that number of writes has taken place. We have formally verified 1-Update and show that it achieves high efficacy in covering remote misses for read/write shared cache blocks while minimizing excess cache and interconnect traffic.

***Keywords*-cache coherence; LLC; write invalidate/update;**

I. Introduction

As Moore's Law comes to its inevitable demise, the main question facing CPU designers is how to increase performance in the absence of doubling transistor budgets. With workloads in a variety of domains – from mobile to servers – being increasingly data driven, accommodating large datasets for fast access has emerged as a particular pain point. Today's CPUs address this need through the use of shared on-die LLCs, whose capacity tends to be proportional to core count. Problematically, shared on-die LLCs suffer from a number of shortcomings. First and foremost, their capacity is naturally constrained by the available area on a die. Unless dies can be made substantially larger, the end of transistor scaling spells an end to substantive LLC capacity increases, which is problematic in light of rapidly-growing dataset sizes. Even if die area could be enlarged, thus enabling greater LLC capacities, the LLC access latencies would suffer due to long on-chip wire spans. Yet another problem of shared LLCs is that they are prone to performance degradation due to contention whenever multiple workloads share a die. Having more cores on a CPU naturally facilitates greater degrees of workload consolidation and colocation, which exacerbates the contention problem.

While big advances in transistor density for planar dies seem unlikely in the foreseeable future, advances in semiconductor technology have enabled 3D stacking of memory dies. 3D die stacking naturally side-steps the planar density problem by growing capacity in the vertical dimension through layering multiple conventional dies on top of each other. Indeed, many existing DRAM memory products already take advantage of 3D stacking to increase density per package. For instance, JEDEC's recent die-stacked high bandwidth memory (HBM) specification, HBM2E, features up to 12, 2GB-DRAM dies in the stack [1].

Can advances in die stacking technology help solve problems of on-chip LLCs? Recent work says yes. Researchers have argued for stacking DRAM atop a CPU to provide massive LLC capacity. Doing so naively, however, would result in large access delays to the multi-GB stack, since LLC requests would still need to traverse large swaths of planar, wire-delay-dominated DRAM on their way to the destination bank. To avoid this problem, prior work proposes organizing stacked DRAM into vertical vaults, whereby each vault is *private* to a given core [2]. The effective result is an all-private cache hierarchy, where L1 and L2 caches are on-die (as in today's designs), while the LLC is also private per core but resides in die-stacked DRAM.

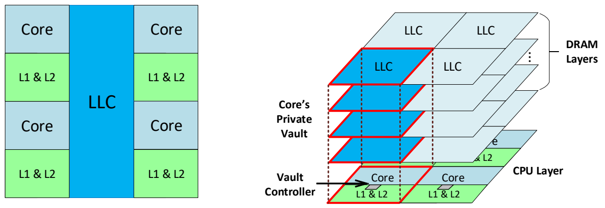
Figure 1 shows the conventional shared LLC and the die-stacked private alternative. The die-stacked private LLC arrangement, which we refer to as as Next-Generation LLC (*NG-LLC*), helps minimize planar wire delays, enabling LLC capacities in the hundreds of MBs at access latencies similar to today's shared LLCs. Combined with the fact that private caches are naturally immune to sharing-induced contention, NG-LLCs significantly improve performance on a wide range of workloads, making them a highly attractive design point for CPUs in the post-Moore world.

While NG-LLCs can dramatically improve LLC hit rates due to their vast capacities, their private nature presents a performance obstacle in parallel workloads that have read/write data sharing. Shared on-chip LLCs enable fast hits if the shared data resides in the LLC; if that is not the case, they can quickly redirect the access to the core caching it because the directory is often co-located with the LLC – thus, the directory lookup and LLC access can happen concurrently. However, in an NG-LLC, there is no shared LLC – as such, the private LLC must first be looked up, following that the distributed directory must be accessed, and after that the private cache hierarchy of the target core must be searched starting from the private LLC. The bottom line is that NG-LLC adds extra accesses in the critical path of a lookup for shared data, and for data that is actively read/write shared by two or more cores, the associated latency overhead may become significant.

One well-known method to reduce the latency of remote accesses for read/write shared data is for the writers to eagerly *push* the data to the likely readers. Protocols that do this are called *updating* cache coherence protocols and have been implemented in commercial shared-memory multiprocessors, including the DEC Firefly [3]. While updating protocols can be effective at reducing stall times for readers of read/write shared data, they are known to generate considerably more cache-to-cache traffic than invalidating protocols because many writes are not consumed by another core before another write happens [4]; as such, a significant amount of cache and interconnect bandwidth is wasted on purposeless update messages.

Several optimizations to updating protocols have been proposed over the years aimed at limiting the traffic overhead stemming from updates. A state-of-the-art scheme, called the *competitive update protocol*, reverts to an invalidating protocol after some number of consecutive updates for a block have been sent without an intervening read [5]. Unfortunately, competitive update and other such schemes are, at heart, updating protocols that default to sending updates for shared blocks and hence trade-off the amount of network traffic with coverage. For these schemes, higher coverage of remote reads necessitates sending more updates.

In this paper, we observe that in many instances, a writer core performs multiple writes to the block before it is requested by any of the readers. We further find that the number of writes by a writer core before a read by a different core is likely to be stable over multiple write/read iterations. Based on these two insights, we propose the *1-Update protocol*. For each read/write shared cache block, the protocol records the number of writes to the block before it is read by any other cores. The next time the block is modified by any core in the system, the protocol counts the number of writes to the block without an intervening read by a different core and, once this count reaches the previously-observed value, it sends an update to the recent readers of the block.



(a) Conventional shared LLC (b) Private LLC in 3D DRAM

Figure 1. Conventional shared LLC and NG-LLC in a 4-core CPUs. Figure from [2].

The 1-Update protocol is, at its core, an invalidation-based protocol whose invariant is that *at most* one eager update is sent for each write/read iteration. This guarantees that even if the update was premature, the amount of wasted bandwidth is minimized. The protocol requires straightforward extensions to the baseline MOESI protocol and just 7 additional bits of metadata per cache block. The metadata is independent of the number of cores in the system, making the protocol attractive for commercial settings. We formally specified and verified the correctness of 1-Update in TLA$^+$.

Simulating a 16-core CMP with NG-LLCs running PARSEC workloads, we show that:

- NG-LLCs can significantly improve performance over shared LLCs but at the price of increased latency on accesses to read/write shared data.
- An updating protocol has the potential to hide the read latency for reader cores; however, the traffic increase due to frequent unconsumed updates leads up to a 9% performance degradation.
- For a given read/write shared cache block, the number of writes before a read by a consumer core is stable across write/read iterations. Based on this insight, we propose the *1-Update* protocol that tracks the number of writes to a shared block and sends at most one update to the reader cores after the observed number of writes.
- The 1-Update protocol improves performance by 8%, on average, while increasing on-chip interconnect traffic by 9.5%. The 1-Update protocol is formally verified.

II. Motivation

A. LLC: Shared or Private?

Modern chip-multiprocessors (CMP) employ large on-die cache capacities in an attempt to capture the large instruction and data working sets of today's applications. The large cache capacity is deployed as a multi-level hierarchy where the first two levels (L1 and L2) are private per core, while the third level (LLC) is logically shared by all the cores. Physically, the shared LLC is split into slices and distributed across the processor die, linked through the on-chip network (NOC).

A shared LLC maximizes effective LLC capacity as only a single copy of data blocks used by multiple cores is kept. It also facilitates data sharing between threads at low latency. Despite these benefits, shared LLC designs also have significant drawbacks. Accessing an LLC slice requires routing the request over slow wires and the multi-hop NOC topology, which leads to a high average LLC access latency. Furthermore, co-running workloads contend for the shared LLCs capacity, potentially leading to performance interference and degradation of quality-of-service for user-facing applications [6], [7].

Private LLCs offer an alternative to address the shortcomings of shared LLCs. Accessing a core's private LLC does not require multi-hop traversal over the NOC, hence keeping the access latency low. Per-core private LLCs naturally eliminate cache contention between co-running workloads and provide strong performance isolation for the applications. However, private LLCs statically partition the total LLC capacity between the cores at design time. This wastes LLC capacity whenever the number of active cores is fewer than the total number of cores or for co-running workloads with skewed LLC capacity requirements. Moreover, private LLCs suffer from lower effective LLC capacity due to the replication of data blocks used by multiple cores. The two aforementioned factors make the cost of moving from shared to private LLCs in current processors with on-chip LLCs prohibitive. For example, Intel's 3rd generation Ice Lake Xeon server processors are equipped with a shared LLC with under 2MB of capacity per core. While the effective shared LLC capacity – on the order of 22–32 MBs in typical configurations – is able to capture a significant part of the instruction and data working sets, in a private configuration, it may be grossly insufficient.

B. Next-Generation LLCs

Die area and power constraints limit the LLC capacity that can be afforded on a planar die – a problem that is bound to get worse as Moore's Law grinds to a halt. Meanwhile, the rapidly maturing die-stacking technology offers multiple layers of tightly-integrated memory cells to alleviate the area constraints of planar designs [8], [9]. Indeed, JEDEC's recent die-stacked HBM specification, HBM2E, and the upcoming HBM3 feature up to 12 layers in the stack, with each layer providing up to 2GB capacity [1].

Recent work has argued for NG-LLCs that exploit the vast capacity offered by die-stacked memory technology to provide *per-core private LLCs* in DRAM stacked directly atop a CPU die [2]. The proposed design uses a multi-layered DRAM stack vertically integrated with the CPU die, and vertically partitioned into slices called *vaults*, a-la the Hybrid Memory Cube (HMC) [10]. NG-LLC eschew a shared, on-chip LLC in favour of die-stacked per-core vaults, which serve as the private LLC for that core. In this organization, the advantages of a private LLC may be enjoyed without severely restricting per-core LLC capacity as would be the case with planar on-die LLCs.

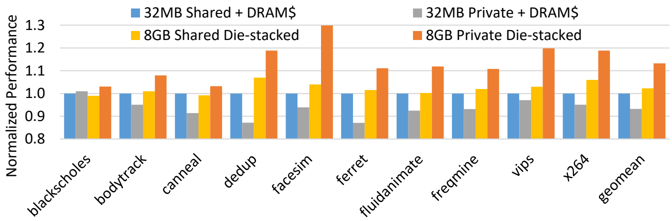


Figure 2. Performance of shared vs private LLCs as a function of capacity.

We study the performance impact of shared vs private LLC organizations in a 16-core setup (details available in Section V). We consider two aggregate LLC capacities: (i) a planar on-chip arrangement with 32MB, which amounts to 2MB per core and is representative of current processors, and (ii) die-stacked DRAM with 8GB, which amounts to 512MB per core, obtained through technology modeling under constrained die area in [2]. For the fairest comparison, we augment the 32MB LLC with an 8GB on-package DRAM cache similar to Intel's Xeon Phi Knights Landing [11], shared by all processor cores. The conventional DRAM cache is hardware-managed and uses a page-based arrangement. While conventional DRAM caches are shown to have access times similar to that of main memory, we optimistically assume the access latency to be 20% faster than main memory. We evaluate the resulting configurations on PARSEC-3.0 [12] workloads. Refer to Section V for more details on the evaluation methodology.

Figure 2 plots the results. The system performance is normalized to the system with a 32MB shared LLC. We observe that in systems with a 32MB LLC, the shared configuration outperforms the private configuration by 7%, on average. In fact, a shared 32MB LLC outperforms a private design with the same aggregate capacity on all evaluated workloads except `blackscholes`, which has a very small working set.

In contrast, given an LLC with 8GB of total capacity, the private configuration greatly outperforms the shared LLC. Across all the workloads, the 8GB private LLC configuration (representing NG-LLC) yields a geomean performance improvement of 13% over the 32MB shared LLC configuration, with a maximum improvement of 30%. In comparison, the 8GB shared LLC configuration yields only a 2% geomean performance improvement over the 32MB shared LLC. The reason for the small gain from a shared 8GB LLC is that it exposes long interconnect delays that offset the benefit of high aggregate capacity of a shared cache.

To summarize, shared LLCs are the superior choice when cache capacity is greatly constrained. However, when LLCs are deployed in high-capacity memory stacks vertically integrated on the CPU die, the large per-core LLC capacity, together with fast access, makes private LLCs the winning configuration.

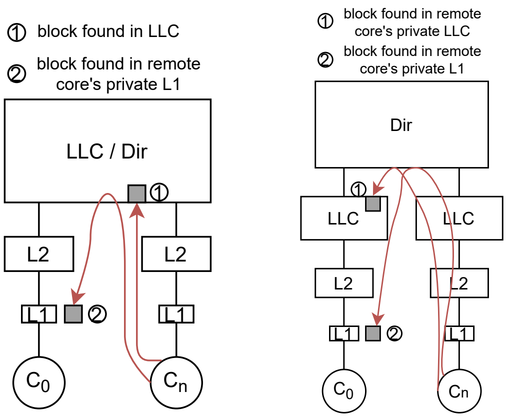
C. *Cache Coherence for NG-LLCs*

In an all-private cache hierarchy, the caches can be kept coherent through a conventional directory-based protocol. For example, SILO [2] leverages a conventional directory-based protocol to maintain coherence among the private caches. Coherence needs to be maintained after the last private cache level, which in the case of an all-private hierarchy is the LLC. Therefore, the directory is logically located between the LLC and the main memory. Physically, the directory is address-interleaved and resides in the same DRAM vaults as the LLC, which means that a fraction of the die-stacked DRAM's capacity is reserved for the directory.

As proposed by Shahab et al. [2], NG-LLC uses a MOESI coherence protocol, which extends the MESI protocol with the Owned (O) state [13]–[15]. In a shared LLC, a dirty-eviction-triggered writeback from a core's final private level of cache requires an access to the on-chip LLC, which is the coherence point. However, in a system equipped with an all-private cache hierarchy, the coherence point is main memory, with writebacks requiring expensive main memory accesses. To circumvent the need for an expensive main memory writeback, the O state in a MOESI protocol designates an 'Owner' for a particular block. A block in O state is valid, dirty and Owned; that is, the cache that has the block in this state must respond to coherence requests for the block. In this way, sharing of a modified data block is allowed without necessitating a writeback of the block to main memory.

Figure 3 illustrates accessing read/write shared data from different cores. When a read access request to a data block misses in a core's private LLC, an access to the cache block's directory slice is triggered. The directory node sits in the stacked DRAM but in a potentially different vault than the requesting core's LLC slice. Once the request is routed to the appropriate vault and the directory access is complete, the request is directed either to a memory controller (if the data is not cached by any of the cores) or to a cached copy identified by the directory. Assuming the data is found in another core's LLC slice, an access to shared data incurs a total of three DRAM accesses and three separate NOC traversals (to the directory, to the destination slice, then back to the requester) in the critical path of the read. In contrast, in a shared LLC organization, if the requested data resides in the LLC, no additional accesses are needed. If the data resides in another core's private cache, the fact that directory is co-located with the LLC in a shared setup again results in fewer cache accesses (no separate directory access and no additional LLC access at the destination core since the LLC is shared) and fewer NOC traversals than in an all-private hierarchy.

When a core wishes to write to a data block, it sends a request to the corresponding directory node to invalidate all remote copies of the data block and assign the writer as the Owner of the data block. Subsequent reads from other cores must contact the directory to find the location of the latest version of the data, which triggers an indirection in the form of a long-latency remote read as described earlier.


① block found in LLC
② block found in remote core's private L1
LLC / Dir
①
L2
L2
L1
②
L1
C0
Cn
① block found in remote core's private LLC
② block found in remote core's private L1
Dir
①
LLC
LLC
L2
L2
L1
②
L1
C0
Cn

(a) Shared LLC (b) Die-stacked private LLC

Figure 3. Accessing inter-core shared data with (a) shared LLC, (b) private LLC.

The bottom line is that accesses to shared data in an all-private cache hierarchy incur higher latency than in a shared hierarchy. This is less of an issue for read-only shared data, since the large capacity of NG-LLCs reduces the likelihood of capacity misses and eliminates inter-core cache contention. However, for read/write shared data, the high latency of core-to-core transfers in a private cache hierarchy may be frequently exposed.

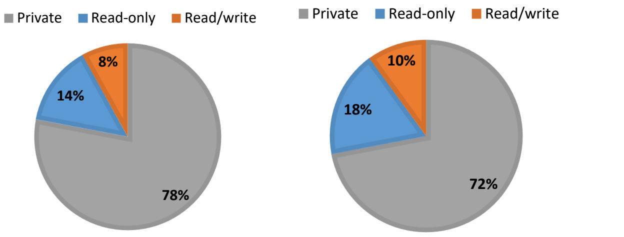
D. *Sharing Behavior Characterization*

Section II-B showed how a private LLC configuration is the superior choice in the context of large aggregate LLC capacities. However, as noted in Section II-C, private LLCs incur higher latency when it comes to inter-core data sharing. We next examine the sharing behavior of PARSEC-3.0 [12] and Splash-3 [16] workloads[1] to understand how it impacts performance of NG-LLCs. In this study, we use 512MB per-core NG-LLCs (Section V details all system parameters).

We classify all cache blocks into three categories: (1) Private: only one core accesses this block; (2) Shared read-only: the cache block is accessed by at least two cores in a read-only manner; and (3) Shared read/write: the cache block is accessed by two or more cores, and at least one write is performed on the block. We study the distribution of LLC access to these three categories.

Figure 4 shows the results. We observe that majority of LLC accesses fall into private cache blocks, which corroborates prior work [17]. On average, private data accounts for 78% of all cached blocks in PARSEC and 72% in Splash. The average proportion of LLC accesses to shared read-only data for PARSEC and Splash is 14% and 18%, respectively. LLC accesses to shared read/write data blocks constitute a non-trivial 8% for PARSEC and 10% for Splash. Shared

[1] For brevity, we refer to these as simply PARSEC and Splash.



(a) PARSEC (b) Splash

Figure 4. Classification of data cached in the LLC for PARSEC and Splash.

read-only data blocks may be replicated in each sharing core's private vault and all subsequent requests may be served locally. However, accesses to the shared read/write data blocks trigger a remote read whenever the reader first accesses the data after a different core performed a write.

To evaluate the performance impact of shared read-only and shared read/write data blocks in a system with NG-LLC, we artificially decrease the latency of accesses to these two types of data blocks served from remote vaults to the latency of a local vault access. The results are shown in Figure 5. We find that after such idealized reduction in the access latency of remote read/write shared data, the potential performance improvement across all PARSEC workloads ranges from 9% to just under 20% (14.5% geomean). In contrast, the potential performance improvement for reducing the access latency of remote read-only shared data is relatively small, ranging from 1.9% to 3.6%, with a geomean of 2.3%. Results on Splash benchmarks (not shown in the figure) show very similar trends.

Similarly, we measure the system performance impact of read/write shared data in a system with a shared LLC and the results are shown in Figure 13. We find that the potential performance improvement across all PARSEC workloads on a system with a shared LLC ranges from 1.5% to just under 8.0% (4.2% geomean). Compared to NG-LLC, the potential performance improvement is considerably smaller with a shared LLC, owing to the faster access latency as explained in Section II-C.

We conclude that modern multi-threaded workloads exhibit significant data sharing, both read-only and read/write. Slow accesses to shared read/write data are particularly detrimental to the performance of NG-LLC and are a promising optimization point from a coherence protocol's perspective. Meanwhile, potential performance gains of optimizing read-only sharing in NG-LLC and read/write sharing in shared LLC are relatively small.

III. Coherence for Read/Write Data

A. *Invalidating vs Updating Protocols*

Today's processors employ *invalidating* coherence protocols that guarantee cache coherence by first invalidating all sharers of a cache block prior to a write. This is facilitated through the use of a directory, which tracks the state of each block (e.g., shared or modified) and the set of cores that may have the block in their private cache hierarchy. Invalidating protocols allow multiple cores to share a *clean* copy of the block while enforcing the invariant that at most one core may hold the block in a modified state.

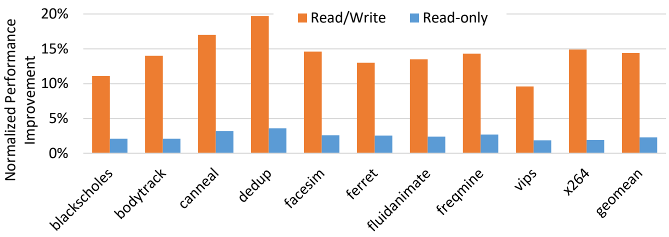


Figure 5. Performance improvement in a NG-LLC after eliminating remote access latency for read/write shared data.

As discussed in the previous section, invalidating protocols trigger a performance pathology for read/write shared data in systems with private caches, including in NG-LLCs, due to the long access latency on reads after a write. Meanwhile, our characterization (Section II-D) and prior work have observed that coherence misses caused by read/write data sharing account for a significant portion of overall cache misses [18]–[20]. Thus, reducing the access latency for read/write shared data in systems with an NG-LLC can improve performance of parallel workloads (Figure 5).

To eliminate the access latency for read/write shared data, prior work has proposed the use of *updating* coherence protocols, which can eliminate cache misses on reads after a write by eagerly pushing the data to the potential readers [4], [21]. The problem, however, is that a writer may update a cache block multiple times before it is consumed by any of the reader cores. As a result, an updating protocol may cause significant cache and interconnect traffic by propagating multiple updates for a given cache block, which, in fact, go unconsumed [4], [5], [22].

Two additional factors exacerbate this problem. First, updating protocols tend to track the recent sharers of a block, and propagate updates to *all* the sharers. Thus, bandwidth waste is multiplicative in the number of sharers a block has had. Second, in strongly-consistent memory models, such as TSO, all copies of the modified cache block that are updated in remote caches are placed in a Shared state following the update, while the writer's copy is also downgraded to the Shared state. Consequently, a subsequent write by the same writer necessitates first invalidating all of the sharers, which requires a round of invalidation and acknowledgement messages whose count is proportional to the number of sharers. The combination of these factors amplifies the network traffic cost of updating protocols by many fold compared to an invalidating protocol, potentially compromising the latency benefit of an updating design.

B. *Adaptive Protocols*

Given that both invalidating and updating protocols have their advantages and drawbacks, researchers have proposed hybrid protocol designs that seek to capitalize on advantages of both. One way of implementing the hybrid scheme is through software support, whereby the programmer indicates the memory pages or data structures that are read/write shared and would benefit from updates [4]. While such software-directed schemes are simple to support from a system's perspective, they rely on the programmer to tag data structures and the corresponding memory pages with the coherence write action to employ. To avoid the need for burdensome software support, we focus on hardware-directed schemes.

Hardware methods track the sharing behavior at cache block granularity and decide (1) which blocks are actively shared, and (2) whether updating or merely invalidating a block is the preferred action in the face of a write. A state-of-the-art class of scheme, as highlighted by Culler and Singh in their seminal book [4], is the *competitive update (CU)* protocol, which limits the number of updates generated for a block without an interleaving read by a core other than the writer. The CU protocol, derived from the competing snooping protocol [23], and extended to a directory-based implementation [24], associates a counter value with each cache block. Upon every block read by the local core, the counter value is set to the maximum threshold t. This counter value is decremented upon every block update received from a remote core. While the counter value is greater than 0, the write action for the block is an update. Upon the counter counting down to 0, the block is invalidated in all remote caches and no further updates are triggered while the current writer continues to modify the block.[2] Alternatively, the block may be probabilistically invalidated, as in the Sun Sparc Center 2000 [25].

Another enhancement to the update-based protocol is a *write grouping (WG)*, also known as write combining, scheme [26], [27]. WG can combine multiple writes to the same cache block into a single update to reduce the coherence traffic. This scheme requires an additional bit in the write buffer to indicate if the block address of the incoming write matches that of the previous write. If the addresses match, then the additional bit is set to 1 indicating that the current write can be grouped with previous write(s). If the addresses do not match, the bit is set to 0 indicating a new write group. Write groups are delayed in the write buffer to utilize temporal locality, thus a delay counter is also required. The head entry in the writer buffer is sent to the cache line if (i) An incoming write cannot be grouped (ii) the write buffer is full, or (iii) the delay counter has reached its threshold. The delay counter is initialized to a given number of cycles and resets each time a new write is grouped.

C. Comparison of Existing Protocols

As discussed in the previous section, the CU protocol tries to balance the network traffic stemming from unnecessary updates with the latency reduction that updates may provide for read/write shared data. The update threshold t is the critical parameter that determines the efficacy of the CU approach. If t is set to a low value, then few updates are sent and the overhead from the extra interconnect and cache traffic is low. The downside of t being set to a low value is that it reduces the opportunity to hide the read latency in case the number of writes prior to a read by a different core is greater than t.

To evaluate the performance impact of different threshold values of CU protocol in a system with private LLCs, we vary the threshold in the range 1–4 and measure the system performance on the PARSEC benchmark. We simulate a 16-core system with an NG-LLC as discussed in previous sections; full details of the methodology can be found in Section V. We find that for the CU approach, a threshold $t = 3$ yields the highest performance across our workload set, and use this value in our studies.

We next compare the following five protocols in the studied NG-LLC setting: (1) MOESI invalidating protocol (WI); (2) pure updating protocol that extends MOESI (WU); (3) CU hybrid protocol with $t = 3$; (4) WG hybrid protocol with the delay counter set to 5 cycles as suggested in [26]; and (5) an ideal scheme that incurs zero latency to access read/write shared data from a writer core's cache hierarchy by a reader core. The WU protocol records the sharers of each cache block at the point of an invalidation by a writer core and sends updates only to these recent sharers. Each write is preceded by an invalidation to guarantee consistency (TSO, in our system), while each update places the sharers (and the writer) into the Shared state. The updated data will be attempted to be pushed to one core's private LLC. If the block has been previously evicted by the core, then no cache insertion is performed. The CU protocol extends the WU protocol by reverting to an invalidating protocol after t updates have been sent without an intervening read, as explained in the previous section.

Figure 6 summarizes the results of the study. We make two key observations from the data. First, the pure updating protocol hurts performance by greatly increasing cache and interconnect contention, corroborating prior studies [5]. That is, while lowering the access latency for the small fraction of the blocks that are read/write shared, the protocol increases the average access latency for other cache traffic to the detriment of the system. Second, both the WG and the CU adaptive protocols improve performance over WI with an average of 3.0% and 3.8% respectively, with CU performing slightly better then WG. The reason why CU outperforms WG is that CU reverts to an invalidating protocol after some

[2] As noted earlier, with a strongly-consistent memory model, each update must be preceded by an invalidation. Thus, in the CU protocol, once the write-update threshold reaches the value of 0, an update is sent to all sharers, then the writer invalidates all sharers to modify the block yet again, and after that no further invalidates (or updates) are sent.

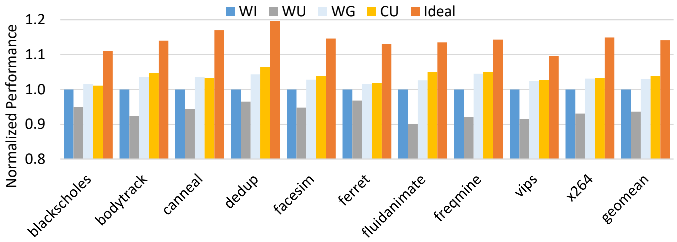


Figure 6. Comparison of coherence protocols for NG-LLCs in PARSEC.

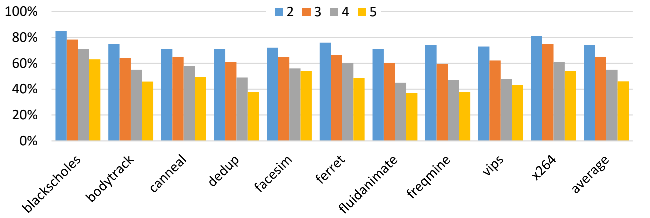


Figure 7. Write stability of PARSEC: number of consecutive write-read iterations in which the number of writes prior to a read by another core is constant.

number of updates, thus reducing on-chip traffic, while WG always sends an update, though sometimes is able to reduce the number of messages via grouping. However, although they are able to limit the excess traffic in the system, they fall considerably short when compared to the ideal improvement of 14.5%.

IV. 1-Update Protocol

A. Basic Idea

As explained in the previous section, an updating protocol has the potential to eliminate the latency of a read for shared data following a write by a different core; however, capitalizing on this opportunity requires avoiding excess cache and NOC traffic due to unneeded updates. To address the latter, we introduce the *1-Update protocol*, which seeks to deliver the latency benefits of an updating design while minimizing the potential for excess traffic. The key idea behind 1-Update is to precisely identify *when* to update the sharers, and to send the update only then and at no other times. The key question is how to determine *when* to send the update.

We make a critical insight that the number of consecutive writes before a read by any of the sharers is likely to be stable across write/read iterations. Such stability arises, for instance, when the writer updates a synchronization variable (e.g., a counter) a recurring number of times and/or updates a shared data structure that has multiple fields in the same cache block.

Figure 7 shows the number of consecutive write/read iterations in which the number of writes prior to a read from a different core is unchanged in the PARSEC suite. For instance, in the best case, we see that over 60% of the time, the number of consecutive writes by the writer core prior to a read by a sharer is unchanged after five write/read iterations. Overall, the figure shows that consecutive write/read iterations are very likely to have the same number of writes without an intervening read by a sharer. On average, the very next write/read iteration has a likelihood of 70% to have the same number of writes as the previous iteration and 50% after four iterations.

The 1-Update protocol leverages write stability by updating sharers only after the previously observed number of writes to the block. To achieve this, 1-Update records the number of times a shared block is modified by one core prior to a read by a different core. The next time a core (any core) writes to the block, 1-Update tracks the number of writes without an intervening read from another core, and once the previously-observed number of writes has been reached, the writer updates all of the sharers. No further updates are sent until a read is detected and another write iteration begins.

The 1-Update protocol guarantees that *at most* one update is sent per each write/read iteration. More precisely, the protocol behaves as an invalidating protocol at all times other than when the write count has reached the previously observed write count for a shared block. If the read happens exactly after the expected number of writes, and if one or more of the previous sharers are doing the read, than the update is effective at hiding the read latency. If a read arrives before the expected number of writes, the protocol acts as a normal invalidating protocol, and no bandwidth is wasted on updates. Only if the number of writes prior to a read is greater than in the previous write/read iteration, or if all of the readers are different than before, is bandwidth entirely wasted on the update.

B. Protocol Details

Starting with an updating protocol that extends an invalidating protocol as described in Section II and Section III, supporting 1-Update involves only a small set of changes. Specifically, 1-Update necessitates a small number of metadata bits per cache block in all of the caches and minimal extensions to the updating coherence protocol. 1-Update requires *no* new coherence protocol messages, and *no* changes to the directory structure but only some extra bits to store the previous sharing information in the directory. Moreover, the amount of per-cache-block metadata necessitated by 1-Update is independent of the number of cores and cache levels in the system, which makes the protocol trivial to deploy across a family of products.

The following metadata must be maintained in all cache blocks in the processor:

Read flag (F): Indicates whether a core has read the block. Multiple cores may set this flag. A read that encounters the block with F set at 0 transitions it to 1. Meanwhile a write that encounters the opposite (i.e., F set at 1) resets it to 0. This latter transition (a write-after-read to the block) establishes the beginning of a new write/read iteration.

Write count (W): A saturating counter that tracks the number of consecutive writes performed since the last read of the block in a non-exclusive state and (i.e., in S or O). The counter stops when a read from a remote core is detected.

Writes until an update (U): When a read terminates a sequence of writes, this counter is initialized with the value of *W*, denoting when an update will take place. U is decremented after each write; once it reaches 1, an update is sent to all sharers prior to the beginning of the current write/read iteration, at which point the coherence state is changed from `Modified` to `Owner`. Figure 8 presents a flowchart detailing the workings of the threshold **U**, when to invalidate and when to push an update.

We find that three bits are sufficient for both **W** and **U**, and one bit for **F**, for a total of 7 bits per cache block. This information is exchanged across cores without introducing new coherence messages as follows:

- When a cache-to-cache block transfer happens, **F**, **W**, and **U** simply travel with the data on a cache-to-cache block transfer, thus transferring any state changes occurred in an exclusive state.
- While in non-exclusive state, only the read flag **F** of a local block may change upon a read. Rather than eagerly communicating **F** with the sharers or the directory, which would mandate extra protocol messages and a latency penalty in the critical path, we communicate this directly with the writer lazily. Remember that **F** is needed only before a write to detect if a write-after-read occurred, denoting the beginning of a new write/read iteration. Therefore, we simply piggyback **F** (1 extra bit) on the **ACK** response to a writer's invalidation request for exclusive access.

Note that with this lazy propagation of **F**, we risk that **F** might have been set on a block which is subsequently evicted and thus not conveyed to the writer. However, this does not affect the correctness, it may only lead to a misprediction. Moreover, our empirical results show that the large capacity of NG-LLCs makes this an unlikely problem for actively shared data.

We now detail the write-update operation, illustrated in Figure 9. In the MOESI protocol, for each cache block, the directory holds the state of the block, an owner field and a sharer list. When the block is in the Modified state, the owner field indicates the writer. Thus, when the block is being modified, the sharer list reflects *previous* sharers. This allows 1-Update to track previous readers with no additional metadata in the directory. When an update message must be sent, the owner of the block (the writer) contacts the directory, which provides the list of previous sharers. Subsequently, the writer performs the update on its local block, which also transitions from `Modified` state to `Owner` state, and uses the obtained sharing information to update the cores that are on the sharing list with the up-to-date data.

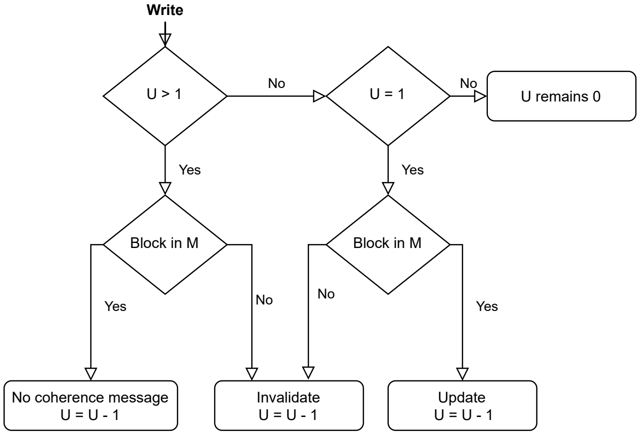


Figure 8. Flowchart of decision on invalidate or update.

Our current design of 1-Update performs an in-place update of a recently invalidated block in the sharer's LLC. We find that, given the large capacity of NG-LLC, the likelihood of an invalidated block being evicted before an expected update is low. If the block update is successful, the sharing core sends an ACK to the directory. If the block update is not successful because the block has been evicted before the update's arrival, then a NACK is sent to the directory and the update is suppressed. Finally, the directory updates its block's state and sharers.

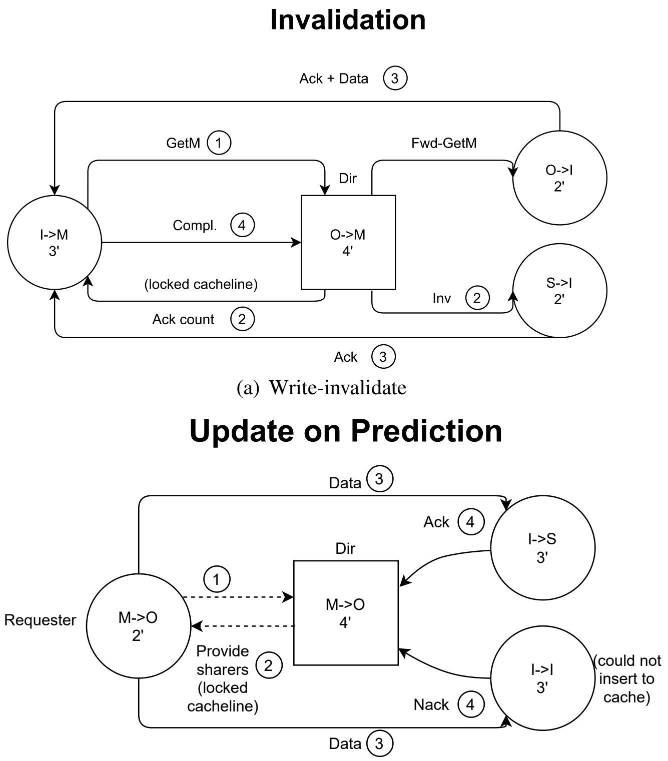
Overall, there are three prediction outcomes based on the number of total consecutive writes in the current write/read iteration and the number of writes in the previous iteration:

1) **Accurate prediction:** The current number of writes is exactly the same as previous one. After the writer pushes the data to previous sharers, the potential readers hit on the data and **F** is set to **1**. The next time the writer writes to the block, it triggers a round of invalidations and **F** is sent back to the writer with at least one of the ACKs. This triggers a new write/read iteration.
2) **Read before predicted update:** This occurs when a core reads the block before the predicted update (i.e., before **U** has been decremented to **1**). This behaves similarly to a read miss to modified data in a pure invalidate-based protocol. The core gets the data from the writer but note that the writer will set the **F** to **1** after receiving an ACK in a subsequent invalidation, thus terminating this write/read iteration. The writer then prepares for a new write/read iteration by resetting **W** and **U**.
3) **No read after predicted update:** This occurs if, after the writer's update message, no other core reads the block. In this case, **U** remains at **0**, while **W** continues increasing until a read takes place or until the counter (W) saturates.

C. Operational Example

This section provides a detailed example to illustrate how the 1-Update protocol works.

Assume one cache block is shared by three cores: **C1**, **C2** and **C4**. The last write count (without intervening reads)



(a) Write-invalidate

(b) Write-update

Figure 9. Coherence actions for write-invalidate and write-update.

W is **4**, which was followed by reads by both **C2** and **C4**. Consequently, **U** is set to **4** (i.e., same as **W**). **C1** performs a number of writes on this block. The first write, which invalidates the copies in **C2** and **C4**, finds the read flag **F** equal to **1** (indirectly through the ACKs that contain **F**) and resets F back to **0**. Right after that first write, **W** also resets to **1** while **U** is decremented to **3**.

Consecutive writes from **C1** will trigger the increment of **W** and decrement of **U**. Note that because the block is already in modified state, no extra coherence messages are sent – exactly as in a typical pure invalidating protocol.

When **U** is decremented to **1**, an update round is triggered. **C1** first contacts the directory to obtain the stored sharing information.

Subsequently, the directory puts the block into an internal state indicating that an update is in progress (the same state used when an invalidation is in progress) and responds to **C1** with the last sharer list (**C2** and **C4** in this example). Once **C1** receives the directory response, its local block state transitions from `Modified` state to `Owner` and an *update* (i.e., containing the local block's data and metadata) is issued to **C2** and **C4**. Upon the reception of the update, **C2** and **C4** try to update the block in their private LLCs. If successful (or not successful), they respond back to the directory with an ACK (or NACK). The directory finalizes the write by updating the sharers and state of the block.

D. *Verification*

We have specified the 1-Update protocol in TLA$^+$ and verified it through model checking for safety and the absence of deadlocks. For safety, we verified several invariants; we elaborate on the two most important ones. First, the single-writer-multiple-reader invariant (SWMR) [28]: at any given time, there can either be at most one writer that can update a cache block *or* one or more readers that can safely read the cache block. Second, the data value invariant: if a cache block is in a state that can be read, it must reflect the most recent value update to that cache block.

Our TLA$^+$ model allows for the number of processors, the predicate on switching update policy (based on the number of writes), and the total number of writes to be configured. We have verified up to six processors, six writes, and an updated policy triggered at three writes. The complete list of invariants and the TLA$^+$ specification are available online.[3]

V. Methodology

A. *Evaluated Systems*

We model a 16-core CMP with 3-way out-of-order (OoO) cores running at 2.0 GHz. Table I details the system parameters. We consider a TSO memory consistency model and a 3-level inclusive cache hierarchy, which is representative of state-of-the-art systems. The L1 and L2 are SRAM-based and are private to each core. While our evaluation mainly focuses on NG-LLCs (die-stacked private LLCs [2]), we also consider a conventional on-die shared LLC.

We use CACTI (details in Section V-B) to obtain DRAM and SRAM access latencies. For the DRAM cache we use a design similar to Intel's Xeon Phi Knights Landing [11], which is placed on-package and shared by all processor cores. The on-package DRAM cache is hardware-managed and uses a page-based organization. Although conventional DRAM caches have been shown to have access times similar to or higher than that of main memory [29], we optimistically assume the access latency to be 20% faster than main memory. We use a closed-page policy for DRAM in all three settings (die-stacked LLC, on-package cache, and main memory), which outperforms the open-page policy on server workloads [30]. We assume main memory access latency of 50ns [2].

We focus on the following cache coherence protocols:

Baseline (WI): A directory-based write-invalidate MOESI cache coherence protocol.

Competitive update (CU): Combines an updating protocol, which extends WI as described in Section III. Recall that the CU protocol switches from an updating to an invalidating one after t updates have been sent. We use a threshold of $t = 3$, which yields the highest performance for the evaluated workloads.

[3] https://github.com/ease-lab/1Update

Processor	16-core, 2GHz, 3-way OoO, 128 ROB, ISA: UltraSPARC v9
L1-I/D	64KB, 8-way, 64B line, 3-cycle, private, stride data prefetcher
L2	512KB, 8-way, 64B line, 5-cycle, private, stride data prefetcher
Interconnect	4x4 2D mesh, 3-cycles/hop
On-chip SRAM shared LLC	32 MB shared NUCA, 2MB per core, 7-cycle, 16-way, 64B line, non-inclusive MESI, LRU
On-chip SRAM private LLC	32 MB in total, 2MB per core, 7-cycle, 16-way, 64B line, non-inclusive MESI, LRU
Die-stacked DRAM shared LLC	Direct-mapped, 64B line, 512B row, inclusive MOESI 512MB vault/core, NUCA, 8GB in total, 50 cycles average round trip
Die-stacked DRAM private LLC	Direct-mapped, 64B line, 512B row, inclusive MOESI 512MB vault/core, 8GB in total, 32 cycles
On-package DRAM cache	8GB, page-based, direct-mapped, 40ns
Main memory	Access latency 50ns

Table I
MICROARCHITECTURAL PARAMETERS OF THE SIMULATED SYSTEMS.

1-Update: The 1-Update cache coherence protocol, as described in Section IV.

B. Latency and Energy Modeling

We model SRAM and DRAM access latencies in CACTI-3DD [31] at 22nm. Our SRAM LLC model uses the low-standby-power (LSTP) cell type and accounts for advanced latency reduction techniques [32].

We measure the energy and power consumed by the die-stacked LLC using a hybrid energy modeling framework that employs both technology-specific parameters and statistics obtained from cycle-accurate simulations. We use CACTI-3DD for the technology-specific energy parameters for die-stacked DRAM [31]. In our evaluation, we take into account energy dissipated by the additional traffic on the on-chip network. For the on-chip network, we use energy parameters published in recent literature [33]. The router energy for 64-bit flit is 2pJ and wire/link energy for 64-bit flit is 4.5pJ. For die-stacked DRAM, the static power is 120mW per vault and the dynamic energy is 0.4nJ per access.

C. Simulation Infrastructure

We use Flexus [34], a full-system simics-based multiprocessor simulator, which implements the SPARC v9 ISA. Flexus extends simics with out-of-order cores, memory hierarchy, and NOC. We use the SMARTS [35] sampled execution methodology to reduce simulation time. Each sample uses a warmed up architectural and microarchitectural state from which cycle-accurate simulation are run to measure performance. As the performance metric, we use the number of application instructions executed per cycle (including time spent executing the operating system code), which reflects system throughput [34].

D. Workloads

For our evaluation, we primarily use contemporary parallel applications from the PARSEC-3.0 benchmark suite [12]. In our 16-core setup, the number of threads equals the number of cores. We use the native input sets and only simulate the *Region of Interest (ROI)*. Compilation and runtime issues prevented us from being able to run two workloads: `streamcluster` and `swaptions`.

We also evaluate the following workloads from Splash-3 [16]: `barnes`, `fft`, `lu_cb`, `lu_ncb`, `ocean_cp` and `ocean_ncp`. The rest of Splash workloads are unavailable because of compilation, runtime and measurement issues.

Similar to PARSEC, we use the native input sets and only simulate the ROI.

We generate samples over 80 billion instruction (5 billion per core) for each workload. We run cycle-accurate simulation for each sample using checkpoints comprising full architectural and partial microarchitectural state, including caches and branch prediction structures. For each sample, we simulate 100K cycles to warm-up and achieve steady state, we then use the following 700K cycles for measurement.

VI. EVALUATION

We evaluate 1-Update protocol against competing protocols based on NG-LLC on PARSEC and Splash workloads. Additionally, we evaluate all baseline protocols and 1-Update in the context of a conventional shared LLC.

A. Performance on PARSEC

Figure 10 plots the performance of the system with the evaluated protocols. All results are normalized to the baseline: NG-LLCs with directory-based WI protocol (see Section V for detailed description). We observe that the CU protocol delivers a geomean performance improvement of only 3.8%, which is much lower than the ideal possible improvement of 14.5%. In comparison, 1-Update delivers a performance improvement of 6.1%–10.2%, with a geomean performance improvement of 8.0% and closer to the ideal improvement of 14.5%. This is expected because 1-Update is based on a write-invalidate protocol, which is more friendly to network traffic; whereas CU protocol is based on a write-update protocol, which tends to generate more traffic and hence affecting the system's performance.

B. Coherence Miss Comparison

The 1-Update protocol is designed to reduce coherence miss traffic caused by typical write-invalidate protocols. We evaluate the coherence misses of the baseline, the CU protocol, and the 1-Update protocol. Figure 11 presents

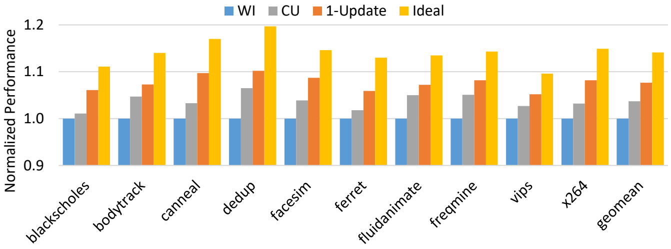


Figure 10. Performance with NG-LLCs.

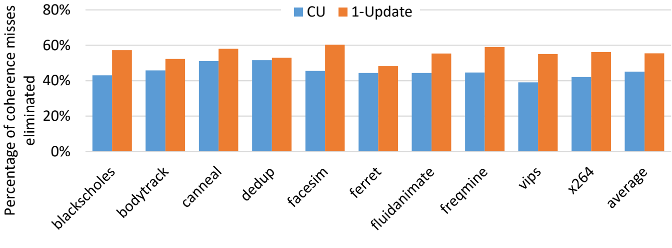


Figure 11. Coherence misses.

the results, normalized to the baseline. We observe that both CU and 1-Update protocols reduce the coherence misses compared to the baseline write-invalidate protocol by 45% and 55% respectively. Workloads that observe large coherence miss reduction (e.g., `canneal, facesim`) also observe higher performance improvement. Overall, 1-Update covers a larger fraction of coherence misses than CU while creating considerably less excess traffic than CU (as detailed in Section VI-C), hence delivering higher performance.

C. Traffic and Power

We evaluate the network traffic of the baseline, the CU, and the 1-Update protocols. The system parameters are covered in Section V. Figure 12 shows the results. The traffic is measured in number of flits sent through the network and is normalized to the baseline. We observe that the CU protocol tends to generate 30% more traffic than the baseline since it is a variant of a pure write-update protocol, thus generating considerable excess traffic. In contrast, 1-Update generates only 9.5% more traffic compared to the baseline.

We also study the impact of the extra NOC traffic and LLC accesses (including directory accesses) on power. We find that total dynamic power of LLC and NOC combined in the target 16-core system increases by 3% due to 1-Update.

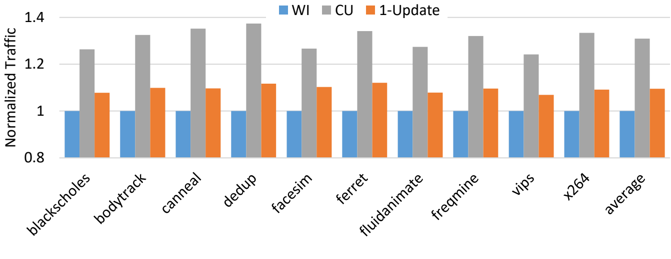


Figure 12. NOC traffic.

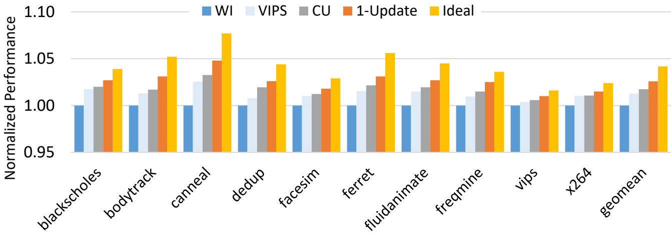


Figure 13. Performance with a shared LLC.

D. Performance in a Shared LLC

In addition to our evaluation of 1-Update on NG-LLC with an all-private cache organization, we also evaluate the performance of the 1-Update in a shared LLC setting, which is typical of current processors. The system employs an on-chip SRAM 32MB shared LLC. We evaluate four directory-based coherence protocol variants: (1) WI which is representative of current systems, (2) CU (with $t = 3$), (3) VIPS [36] which is the state-of-the-art optimized coherence protocol for shared LLCs, and (4) 1-Update. The system parameters are detailed in Section VI.

Figure 13 plots the performance of PARSEC workloads with results normalized to the system with write-invalidate. Compared to write-invalidate, all three alternatives (CU, VIPS, and 1-Update) provide a marginal performance improvement, with the geomean improvement of 1.7%, 1.3%, and 2.6%, respectively. The marginal performance improvement can be explained by the fact that shared LLCs naturally benefit read/write data sharing since a core can normally obtain the up-to-date modified data from the shared LLC, without requiring a long-latency remote access.

Of the three alternatives to the baseline invalidating protocol, VIPS yields the lowest performance improvement. The modest speed-up for VIPS is consistent with the results presented in the original paper [36]. VIPS targets reducing coherence state to simplify implementation in hardware, and therefore does not yield performance benefits. While the performance gains are marginal in the private setting, 1-Update still outperforms the CU protocol in all workloads. Thus, we can conclude that 1-Update is superior to CU regardless of the LLC configuration (private or shared).

E. Effect of History Length

Up until now, we have considered 1-Update with a history of 1, i.e, update prediction is based on the number of writes in just the last write-read iteration. This policy works well when the write count without interleaving reads does not fluctuate often. From Figure 7 we observe that while PARSEC workloads demonstrate a fair degree of write count stability, there is clearly some degree of variance as well. Thus, we study whether a confidence mechanism can help mitigate the impact of transient fluctuations in write count.

We assume a simple implementation of a confidence mechanism that maintains a count of writes observed during

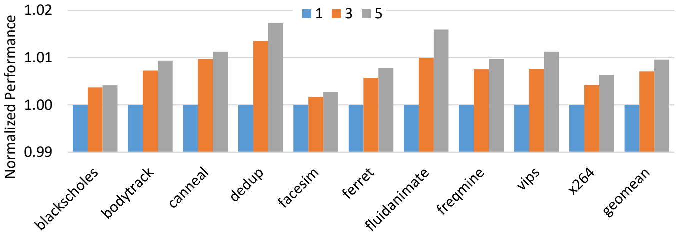


Figure 14. Sensitivity to history length.

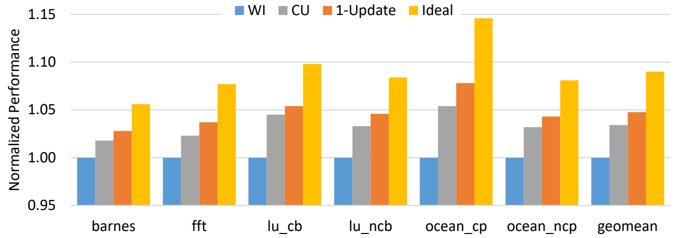


Figure 15. Performance on Splash.

the previous N iterations (N=1 for baseline 1-Update protocol). If the write-count in the N iterations differs, a majority vote is used to decide the write count to be used as the update trigger in the current iteration. If a majority does not exist, the write count from the most recent iteration is used.

Figure 14 shows the performance results of 1-Update with a history length of 1 (baseline 1-Update), 3, and 5 across PARSEC workloads, normalized to the 1-Update. We observe that history length of 3 and history length of 5 improve performance slightly compared to 1-Update, with a geomean improvement of 0.7% and 1%, respectively. We observe that history lengths of 3 and 5 performs better when write count stability is low, such as on `dedup` and `fluidanimate`. However, a longer history exacerbates storage requirements and adds complexity to the prediction mechanism in terms of both history maintenance and decision-making, resulting in a tradeoff for system designers.

F. Performance on Splash

We study the performance of the coherence protocols on the Splash workloads. Figure 15 plots the performance, with results normalized to the baseline: NG-LLCs with directory-based write-invalidate protocol. The results show that 1-Update delivers a performance improvement ranging from 3% to 8% with a geomean improvement of 5%, which is better than the CU protocol with a geomean improvement of 3.5%. Similar to PARSEC workloads, the 1-Update protocol always outperforms CU on Splash workloads.

VII. Related work

As previously discussed, implementations focusing solely on write-invalidate protocols [37] or write-update protocols [3] have their drawbacks. To mitigate these shortcomings, hybrid protocols, such as competitive snooping [23] and competitive update [5] have been proposed. These have been further optimized using additional write buffers and schemes that group writes to the same cache block [38], [39] bearing similarities with the implementation of Alpha 21064 [40]. These techniques amortize the cost of an update over a group of writes to same block by triggering just one update; thus further reducing the messages of competitive update protocols. However, all these works assume a baseline of write-update protocols. Thus, cannot be easily adopted by today's invalidating-based world of coherence and come with their own consistency challenges. For instance, mandating a two-phase implementation [41] to guarantee correctness in strongly-consistent memory models.

Several works target to reduce the overhead of coherence building on a baseline of write-invalidate protocols. Most of them focus on detecting sharing patterns, including producer-consumer [42], [43], migratory sharing [44], [45] and pairwise sharing [46]. Each scheme targets a specific sharing pattern, which may not yield benefits when applied to applications with different sharing behavior. Besides, they are all orthogonal and complementary to 1-Update.

Finally, coherence overhead can be reduced through coherence prediction. Cosmos coherence message predictor [47] is derived from the two-level branch prediction of Yeh and Patt [48]. Acacio et al. [49] propose a mechanism about sharers prediction which reduces 3-hop misses to 2-hop misses. A series of works by Lai and Falsafi focus on increasing the prediction accuracy as well as reducing the overheads of sharing predictors [50], [51]. Instruction-based predictors [52], [53] are proposed as an alternative to normal address-based predictors. There are also some coherence predictors based on perceptron [54]. All of these optimizations are costly and may require some major changes to a processor design. In contrast, the 1-Update protocol is effective yet it requires minimal hardware overheads and keeps the design simple.

VIII. Conclusion

Recent work has introduced NG-LLCs based on die-stacked DRAM organized as per-core private caches. While NG-LLCs offer a number of advantages over today's shared LLC designs, they are prone to slow inter-core reads for read/write shared data. This work has introduced the *1-Update* protocol that anticipates when a read would occur, and updates the caches of likely readers ahead of time, thus eliminating the long-latency cache-to-cache transfers. The prediction mechanism for triggering an update is powered by a new observation regarding the stability of writes across write-read iterations. The 1-Update protocol achieves high prediction accuracy and low excess traffic, which sets it apart from prior updating and hybrid coherence protocols.

Acknowledgments

This work was supported by Arm PhD Scholarship Program and the EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh.

References

[1] *Why Is the Evolving HBM3 Such a Revolutionary Technology and How Can You Be Ready for It?*, https://community.cadence.com/cadence_blogs_8/b/fv/posts/why-is-the-evolving-hbm3-such-a-revolutionary-technology-and-how-can-you-be-ready-for-it.

[2] A. Shahab, M. Zhu, A. Margaritov, and B. Grot, "Farewell my shared llc! a case for private die-stacked dram caches for servers," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[3] C. Thacker, L. Stewart, and E. Satterthwaite, "Firefly: a multiprocessor workstation," *IEEE Transactions on Computers*, vol. 37, 1988.

[4] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[5] H. Grahn, P. Stenström, and M. Dubois, "Implementation and evaluation of update-based cache protocols under relaxed memory consistency models," *Future Generation Computer Systems*, vol. 11, 1995.

[6] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541944

[7] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485974

[8] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner, "Picoserver: Using 3d stacking technology to enable a compact energy efficient chip multiprocessor," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168873

[9] D. H. Woo, N. H. Seong, D. L. Lewis, and H. H. S. Lee, "An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan 2010.

[10] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, Aug 2011.

[11] A. Sodani, "Knights landing (knl): 2nd generation intel; xeon phi processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015.

[12] X. Zhan, Y. Bao, C. Bienia, and K. Li, "Parsec3.0: A multicore benchmark suite with network stacks and splash-2x," *ACM SIGARCH Computer Architecture News*, vol. 44, 2017.

[13] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Washington, DC, USA: IEEE Computer Society Press, 1986.

[14] "Arm cortex-a57 mpcore processor technical reference manual," Tech. Rep., https://developer.arm.com/documentation/ddi0488/d/level-2-memory-system/cache-coherency.

[15] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, 2003.

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

[17] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[18] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, "A novel approach to reduce l2 miss latency in shared-memory multiprocessors," in *Proceedings 16th International Parallel and Distributed Processing Symposium*, 2002.

[19] Liqun Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter, "Interconnect-aware coherence protocols for chip multiprocessors," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006.

[20] N. Eisley, L. Peh, and L. Shang, "In-network cache coherence," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.

[21] E. McCreight, "The dragon computer system: An early overview. xerox corp." Tech. Rep., 1984.

[22] L. Cheng and J. B. Carter, "Extending cc-numa systems to support write update optimizations," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[23] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," *Algorithmica*, vol. 3, 1988.

[24] H. Grahn, P. Stenström, and M. Dubois, "Implementation and evaluation of update-based cache protocols under relaxed memory consistency models," *Future Generation Computer Systems*, vol. 11, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167739X9400067O

[25] B. Catazaro, "Multiprocessor system architectures: A technical survey of multiprocessor/multithreaded systems using sparc, multilevel bus architectures and solaris (sunos)," 1994.

[26] D. Glasco, B. Delagi, and M. Flynn, "Update-based cache coherence protocols for scalable shared-memory multiprocessors," in *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, vol. 1, 1994.

[27] R. Bianchini and L. Kontothanassis, "Algorithms for categorizing multiprocessor communication under invalidate and update-based coherence protocols," in *Proceedings of Simulation Symposium*, 1995.

[28] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence, second edition," *Synthesis Lectures on Computer Architecture*, vol. 15, 2020. [Online]. Available: https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[29] C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C^{3}d: Mitigating the NUMA bottleneck via coherent DRAM caches," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783739

[30] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "Bump: Bulk memory access prediction and streaming," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, 2014. [Online]. Available: https://doi.org/10.1109/MICRO.2014.44

[31] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '12. San Jose, CA, USA: EDA Consortium, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2492708.2492719

[32] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.30

[33] S. Werner, J. Navaridas, and M. Luján, "Designing low-power, low-latency networks-on-chip by optimally combining electrical and optical links," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[34] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, 2006. [Online]. Available: https://doi.org/10.1109/MM.2006.79

[35] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03. New York, NY, USA: Association for Computing Machinery, 2003. [Online]. Available: https://doi.org/10.1145/859618.859629

[36] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012.

[37] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990. [Online]. Available: https://doi.org/10.1145/325164.325132

[38] F. Dahlgren, J. Skeppstedt, and P. Stenström, "An evaluation of hardware-based and compiler-controlled optimizations of snooping cache protocols," *Future Generation Computer Systems*, vol. 13, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X98000028

[39] F. Dahlgren, "Techniques for improving performance of hybrid snooping cache protocols," *Journal of Parallel and Distributed Computing*, vol. 59, 1999.

[40] D. DECChip, "21064 risc microprocessor preliminary data sheet," Technical report, Tech. Rep., 1992.

[41] A. W. W. Jr. and R. P. L. Jr., "Hiding shared memory reference latency on the galactica net distributed shared memory architecture," *J. Parallel Distributed Comput.*, vol. 15, 1992. [Online]. Available: https://doi.org/10.1016/0743-7315(92)90049-S

[42] L. Cheng, J. B. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[43] L. Cheng and J. B. Carter, "Extending cc-numa systems to support write update optimizations," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[44] A. Cox and R. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[45] P. Stenstrom, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[46] D. B. Gustavson, "The scalable coherent interface and related standards projects," *IEEE Micro*, vol. 12, 1992.

[47] S. S. Mukherjee and M. D. Hill, "Using prediction to accelerate coherence protocols," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98. USA: IEEE Computer Society, 1998. [Online]. Available: https://doi.org/10.1145/279358.279386

[48] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: Association for Computing Machinery, 1992. [Online]. Available: https://doi.org/10.1145/139669.139709

[49] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, "The use of prediction for accelerating upgrade misses in cc-numa multiprocessors," in *Proceedings International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[50] A.-C. Lai and B. Falsafi, “Memory sharing predictor: the key to a speculative coherent dsm,” in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999.

[51] An-Chow Lai and B. Falsafi, “Selective, accurate, and timely self-invalidation using last-touch prediction,” in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 2000.

[52] S. Kaxiras and J. R. Goodman, “Improving cc-numa performance using instruction-based prediction,” in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, 1999.

[53] S. Kaxiras and C. Young, “Coherence communication prediction in shared-memory multiprocessors,” in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, 2000.

[54] S. Leventhal and M. Franklin, “Perceptron based consumer prediction in shared-memory multiprocessors,” in *2006 International Conference on Computer Design*, 2006.

A. Artifact Appendix

A.1 Abstract

1-Update is a new cache coherence protocol that is based on traditional invalidate protocols and also adopts the advantages of updating protocols. It tracks the number of writes to a shared block, and sends at most one update to the reader cores after the observed number of writes. A brief description follows and more details can be found in the paper.

This is the publicly available artifact repository supporting 1-Update, which contains the formal protocol specification. The specification is written in TLA+ and can be used to verify 1-Update's correctness via model-checking.

A.2 Artifact check-list (meta-information)

- **Program:** Java, TLA+ Toolbox and formal protocol specification
- **How much disk space required (approximately)?:** 400MB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 10 minutes
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** `https://doi.org/10.6084/m9.figshare.15112932.v1`

A.3 Description

A.3.1 How to access

The formal specification of 1-Update protocol is available on:

- Github: `https://github.com/ease-lab/1Update`
- FigShare: `https://doi.org/10.6084/m9.figshare.15112932.v1`

A.3.2 Hardware dependencies

No specific requirements.

A.3.3 Software dependencies

Any OS with Java 1.8 or later, to accommodate the TLA+ Toolbox.

A.4 Installation

A.4.1 Install Java

Choose the Operating System for instructions from `https://java.com/en/download/help/download_options.html` to install Java.

A.4.2 Install TLA+ toolbox

Follow the instructions in `https://lamport.azurewebsites.net/tla/toolbox.html` to install TLA+ toolbox.

A.4.3 Obtain source code

Clone or download the git repository from `https://github.com/ease-lab/1Update`.

A.5 Experiment workflow

1. **Launch the TLA+ Toolbox**
2. **Create a spec:** *File* → *Open Spec* → *Add New Spec...*; Browse and use *1Update/OneUpdate.tla* as root module to finish.
3. **Create a new Model:** Navigate to *TLC Model Checker* → *New model...*; and create a model with the name "one-update-model".
4. **Setup Constants:** Then specify the values of declared constants (under *"What is the model?"* section). You may use low values for constants to check correctness without exploding the state space. An example configuration would be four cores, maximum writes of seven and an update prediction of five. To accomplish that, you would need to click on each constant and select the *"ordinary assignment"* option. Then fill the box for write related constants (i.e., *MAX_WRITES* and *WRITE_TO_UPDATE*) with the desired number (e.g., with *"7"* and *"5"*) and the core related constant (i.e., *CORES*) with a set of cores (e.g., *"1,2,3,4"* – for four cores). Then set the *ENABLE_DIR_ACKS* to *FALSE* if writer collects the acknowledgments. Finally, to model check the variant of the paper where the acknowledgments are gathered by the directory instead of the writer, set the *ENABLE_DIR_ACKS* to *TRUE*.

A.6 Evaluation and expected results

After following the steps in A.5, simply click *"Run TLC Model Checker"* and then the model checking results are available in TLA+ Toolbox.

A.7 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`