TSOPER: Efficient Coherence-Based Strict Persistency

Per Ekemark*,Yuan Yao*,Alberto Ros†, Konstantinos Sagonas*‡, and Stefanos Kaxiras*

*Dept. of IT, Uppsala University, Sweden †DITEC, University of Murcia, Spain ‡School of ECE, NTUA, Greece

I. Motivation

In recent years, non-volatile memory (NVM) has attracted significant attention in the research community as it introduces new challenges in the hardware/software interface. In particular, a key question that arises is: what is the observed order in which memory writes are *persisted* in NVM?

Persistency models and their semantics, which roughly can be classified as *strict* and *relaxed* [7], give answers this question. Strict persistency semantics adhere to the underlying memory consistency model: the order of stores, as seen by observers in the consistency model, is preserved in the persist order. Conceptually, this allows enforcing persist order using the same mechanisms provided by the consistency model to enforce store order. However, this proves to be expensive. Relaxed persistency semantics, on the other hand, decouples persistency order from consistency order and allows the order of persist operations to deviate from the order in which the corresponding stores become visible in the consistency model.

Relaxed persistency models (epoch persistency, language-level persistency models, persistency for synchronization-free regions, etc.) are potentially a very good fit for relaxed consistency models as they require programmer involvement for correctness. In other words, the programmer has to fence both for the consistency model *and* for the persistency model, and the effort for the former may be leveraged for the latter. However, in architectures that implement a stronger consistency model, such as x86-TSO, we are faced with a discrepancy between the consistency model and the persistency model. This discrepancy exists today between x86-TSO and the relaxed persistency model introduced by Intel [4], formally described as Px86 [8]. The consistency model, x86-TSO, requires little or no involvement from the programmer. In contrast, the persistency model, Px86, requires significant involvement with insertion of `CLFLUSH` (program-ordered, buffered, persist operation), `CLFLUSHOPT` (unordered, buffered, persist operation), `CLWB` (unordered, buffered, persist operation), and `S/MFENCE` (persist barrier), in the proper places in the code. This involvement is comparable to the effort that would be needed for a relaxed consistency model. This largely negates the benefit of having TSO to begin with, as anyone who wishes to achieve correct persistency would have to fence the programs as if they were meant to run on relaxed consistency. This is problematic for all existing software that runs on x86-TSO but is not fenced for persistency.

Furthermore, relaxed models (either for consistency or persistency) rely on *data-race-free* (*DRF*) semantics for correctness. However, the burden falls on the programmer to provide DRF guarantees. Arguably, correct, well-behaved programs should be free from data races. Unfortunately, this reasoning has two issues: (1) it does not take into account legacy software that is written for TSO and which may have data races as optimizations to synchronization; and (2) it does not take into account data races at a *coarser granularity* than individual variables, i.e., data races that are due to *false sharing*.

In contrast, a *hardware-only* implementation of a strict persistency model that adheres to a consistency model such as TSO (or SC) is not plagued by such problems, because it can simply detect run-time conflicts at cacheline granularity and react accordingly. Thus, without any software involvement, a program compatible with TSO will also persist correctly on hardware-only strict TSO persistency. Furthermore, any DRF program with no additional annotations that runs correctly on TSO also persists correctly on this persistency architecture.

II. TSOPER: A Strict TSO Persistency Model

Our work provides a new solution for an efficient *hardware-only* implementation of a strict TSO persistency model, called *TSOPER*. TSOPER relies on a TSO persist buffer that sits in parallel to the LLC. Private caches persist directly to this buffer, bypassing the coherence serialization imposed by the shared LLC. This is our main differentiation point from the prior state-of-the-art, Buffered Strict Persistency (BSP) [6], and the driver for our design decisions.

Our insight is that we can use coherence to both automatically create the proper "epochs" per thread in the corresponding private caches *and* order these epochs according to the data dependencies (among threads) that arise at runtime. The term "epoch," however, refers to program (instruction) execution. We fully decouple persistency from program execution as our approach is *not* software-driven. Thus, in TSOPER, we talk about *atomic groups of cachelines*, rather than epochs, but we note that there is a relation between the two.

An atomic group (AG) is created and expands to include locally-modified cachelines in the private cache of a thread as long as no local modification is exposed to other threads. Inspired by BulkSC [1], our AGs preserve TSO, but in contrast to BulkSC, we do so *non-speculatively*. If the cachelines of an AG are persisted *atomically* (with no intervening conflicting persist), we maintain TSO persistency regardless of the order the individual cachelines of the group are persisted. Interactions with the outside world, concerning the cachelines of an AG, either create dependencies for the AG, when it sees the

This is an extended abstract of our HPCA 2021 paper with the same title. This work was supported in part from the EU Horizon 2020 research and innovation programme (grant No 801051), from the European Research Council (ERC grant No 819134), and from Vetenskapsrådet (grant No 2018-05254).

modifications of other AGs, or freeze the atomic group, when it is forced to expose its own modifications to other AGs. Freezing an atomic group automatically starts the process of persisting it; a new AG (in program order) starts forming in the private cache of the same thread, capturing subsequent stores for this thread. Similarly to other approaches (e.g., BSP [6] and SFR decoupled [7]), the dependencies among AGs (epochs in other approaches) must be respected in the persist order of the atomic groups. *The key insight of this work is that dependencies among atomic groups can be fully captured at the coherence level, and in particular entirely at the L1 caches.*

To demonstrate this capability, we have developed a sharing-list protocol, inspired by SCI [5], that naturally captures in a sharing list the order in which the coherence operations for a block are serialized by the directory. In such a protocol, different writers of a shared block queue up one after another in the list, and perform their persists in the order in which their stores were inserted in the memory order. Persistency is enforced belatedly, trailing coherence, but following the same order. A block's sharing list is dismantled only by the ordered persist of the locally-modified cachelines on the sharing list.

By ensuring that it is impossible to create dependence cycles among AGs, we guarantee *deadlock-free* TSO persistency with enough flexibility to coalesce multiple stores in cachelines and re-order individual cacheline persists (within an atomic group) to match the level of performance of the relaxed persistency models. We do not rely on any kind of speculation or transactional approach that would detect conflicts, roll-back, and retry. Our approach is strictly *non-speculative*, rendering any cost related to out-of-core speculation (checkpoints, maintaining speculative state, commit overhead, etc.) unnecessary.

We envision a unified mechanism that enforces *coherence* to support the consistency model (TSO) *and* enforces the *persistency* model. In our HPCA'21 paper [2], for simplicity, we use the *same* sharing-list protocol for both coherence and persistency, but, in principle, a different protocol could be used for coherence. Using the same sharing-list protocol for both coherence and persistency has the advantage of easily demonstrating the decoupling of coherence and persistency: coherence happens at the head of the sharing lists (the young memory order end) while persistency happens at the tails of the sharing lists (the old memory order end).

As in other PM models, including Px86, SFR persistency, and PTSO, we also employ *persist buffering* to decouple *program execution* from the actual persist operations that eventually reach the NVM. In our model, a TSO persist buffer, similarly to Intel's *Write Pending Queue*, guarantees that AGs, persisted directly from the L1 caches, will be made durable in NVM even in the event of a crash. Our TSO persist buffer, called *Atomic Group Buffer (AGB)*, accommodates multiple versions of same-address cachelines in different atomic groups awaiting their writing to NVM. This is a fundamental differentiation point from both BSP and LAD [3] that persist through LLC and impose a single-version restriction: *visibility* and writing to NVM must be interlocked on the same version.

III. Main Innovations and a Taste of Performance

In a nutshell, TSOPER eliminates two major serializations in strict persistence via the following two innnovations:

1. The use of a sharing list protocol allows multiple writers to co-exist in the sharing list, eliminating the need to *block in the L1 cache, waiting for values written in different caches to persist in order*. Different writers of a shared block queue up one after another in the sharing list, and perform their persists in the order in which their stores were inserted in memory.

2. In TSOPER, atomic groups (AGs) are directly persisted from L1s to the persist buffer, AGB, in parallel to being written in the LLC. In other words, we do not need to use the LLC as a staging area before writing to NVM as in BSP. This nearly eliminates the need to wait for an LLC value to be persisted before overwriting it in the consistency domain. Similarly to Intel's *Write Pending Queue*, our AGB guarantees that atomic groups, persisted directly from the L1 caches, will be made durable in NVM even in the event of a crash (e.g., power failure). To reduce the high write latency to NVM, AGB banks are implemented with faster technology such as battery-backed SRAM of modest size (10 KB) that is feasible with existing technology. In contrast, whole-system persistence would require power backup for all private/shared caches and, most importantly, it must be equipped to handle the worst case scenario of potentially having to write tens of MB of dirty cached data back to NVM on a power failure.

In our full paper [2], we show through detailed simulation that our approach decouples coherence and persistence further than BSP, and achieves performance levels close to relaxed persistency (3% performance overhead on average).

References

[1] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk enforcement of sequential consistency," in *Proceedings of the 34th ISCA*. ACM, Jun. 2007, pp. 278–289. [Online]. Available: https://doi.org/10.1145/1250662.1250697

[2] P. Ekemark, Y. Yao, A. Ros, K. Sagonas, and S. Kaxiras, "TSOPER: Efficient coherence-based strict persistency," in *IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA 2021. IEEE Computer Society, Mar. 2021.

[3] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd MICRO*. ACM, Oct. 2019, pp. 466–478. [Online]. Available: https://doi.org/10.1145/3352460.3358321

[4] "Intel® 64 and IA-32 architectures software developer's manual (combined volumes)," Oct. 2019, Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[5] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi, "Distributed-directory scheme: Scalable coherent interface," *IEEE Computer*, vol. 23, no. 6, pp. 74–77, Jun. 1990. [Online]. Available: https://doi.org/10.1109/2.55503

[6] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th MICRO*. ACM, Dec. 2015, pp. 660–671. [Online]. Available: https://doi.org/10.1145/2830772.2830805

[7] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the 41st ISCA*. IEEE Computer Society, Jun. 2014, pp. 265–276. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853222

[8] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the Intel-x86 architecture," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 11:1–11:31, 2020. [Online]. Available: https://doi.org/10.1145/3371079