Cache-Coherent CLAM (WIP)

Chen Ding
Benjamin Reber
University of Rochester
Rochester, New York, USA

Dorin Patru
Rochester Institute of Technology
Rochester, New York, USA

Abstract

Traditional caches are automatic and cannot be controlled directly by software. A recent design called CLAM manages a cache using leases and lets a program specify these leases. The lease cache is mostly controlled by software. This paper extends CLAM to support multiple cores with private caches. It presents the hardware extensions to support cache coherence for data-race free (DRF) programs. CLAM can use either inclusive or exclusive caching for shared data. Its performance can be improved by two programming techniques: cache draining and reference privatization.

CCS Concepts: **• Software and its engineering → Retargetable compilers**; **• Computer systems organization → Multicore architectures**.

Keywords: programmable cache, programmable cache coherence, lease cache

ACM Reference Format:
Chen Ding, Benjamin Reber, and Dorin Patru. 2022. Cache-Coherent CLAM (WIP). In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3519941.3535073

1 Introduction

A memory hierarchy cannot be fully optimized unless software and hardware can operate in concert. CLAM is a new cache design that is programmable. This work extends CLAM from a single cache to a set of caches and addresses the coherence problem among caches. It augments the design of programmable cache with programmable cache coherence.

We call the new design *CLAM coherence*. It adds a bit in each cache block to indicate share data and a new hardware primitive called *cache-wipe* for a thread to clear all shared data from its private cache. While conventional designs require inclusive caching for shared data, CLAM supports both inclusive and exclusive caching of shared data. However, CLAM coherence does not support all parallel programs but data-race free (DRF) programs only.

CLAM coherence is programmable and enables software and hardware to "cooperate" in two ways. The first is *cache draining*, where a thread can be programmed to evict shared data. In the best case, cache draining evicts all shared data ahead of time, the cache-wipe becomes a no-op, and the maintenance of cache coherence completely overlaps with computation and does not impede scalability. The second is *reference privatization*, which marks shared data as private and maintains their coherence entirely in software. The first technique hides the cost of cache-wipe but still evicts shared data upon synchronization. The second technique removes this requirement. Wang et al. [11] coined the term *cooperative caching*. CLAM may be called *cooperative cache coherence*.

2 CLAM Prototype

Prechtl et al. [8] built a prototype lease cache called *Compiler Lease of Accelerator Memory (CLAM)*. It has one RISC-V processor and one data cache.

A reference-lease table is loaded before executing each loop. At each access, the data block is given the lease (of its reference). The data block stays in the cache until it is reused or its lease expires, whichever comes first. If it is reused, its lease is extended (by the lease of the new access). It uses random eviction if the cache is full.

The lease cache is mostly controlled by software, but not always. At each access, a program instructs the cache to store a data block for the duration of the lease, but the hardware may evict it early because of lack of space. On the other hand, a program can always evict a data block by accessing it and assigning a zero lease.

3 CLAM Coherence

3.1 Machine and Program Model

A processor has a set of cores and a two-level cache hierarchy, shown in Figure 1. Each core has a first-level private cache, and all cores share the second-level cache. We may refer to each core or its cache as *self* and another one a *peer*. We will later extend the design to more than two levels of cache.

Our design supports data-race-free (DRF) programs. A program is DRF if in all its executions, if two threads access the same memory location, and at least one of them is a

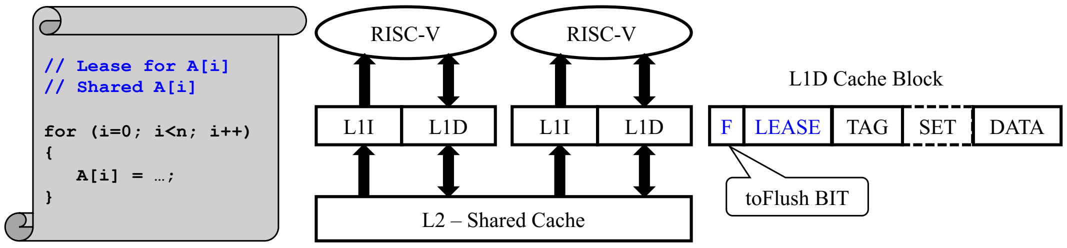

Figure 1. Overview of CLAM coherence. A program reference is annotated with the lease and reference type. An F (toFlush) bit is added to every L1D cache block.

write, the two accesses must be ordered by a happen-before relation [10, 3.4].

3.2 Programming Interface

All data references, i.e. loads and stores, in a program belong to one and only one of the following three types:

- *Single copy* or *volatile*. Data accesses *bypass* private caches and use only the shared cache.
- *Non-shared* or *private*. Data accessed by these references are either thread local or read only.
- *Shared* or *default*. The last case is the default, where the data accessed may be accessed by multiple threads and are not read only.

Figure 1 shows an example loop where one reference is marked shared and given a lease (Section 2). By default, any data a thread accesses is shared. The shared data is the target of the cache system to maintain their coherence and consistency.

3.3 Hardware Support

The hardware support that is needed beyond a single-core system is as follows:

- Private cache: it stores a local copy of shared data when it is accessed by its core. Each cache block has a *toFLush* (F)[1] bit, as shown in Figure 1. The bit is set if a data block is accessed by a shared reference.
- Shared cache: reads and writes to shared cache are totally ordered. *It does not store special bits for coherence control.*
- cache-wipe instruction: This is a new hardware instruction. When a thread executes a cache-wipe, the private cache clears all shared data by evicting all cache blocks with a F bit.

A cache-wipe is a blocking operation and halts a thread execution until it finishes. It may be invoked concurrently by multiple threads, and multiple cache-wipes may overlap in execution. Writebacks are serialized at the shared cache.

[1]The term *toFlush (F)* is based on Ros and Jimborean [9] (see Section 4).

Exclusive Caching of Shared Data. A two-level cache may be inclusive, where L1 content is a subset of L2 content; exclusive, where their contents have no overlap; or a mix, where some data uses inclusive caching and others exclusive caching. Inclusion is most beneficial when threads share the working set that fits in L2 but not in L1. We may call a shared working set *actively* shared data (which can be measured precisely [6]). In other cases, inclusion causes low space utilization in the shared cache. As the number of cores increases, private caches must be small so the shared cache has space for data not already stored in private caches. Exclusive caching allows large private caches.

Of the three reference types (Section 3.2), only private and shared data use two-level caching, and both can use either inclusive or exclusive caching. Current hardware uses directory-based protocols [7]. For shared data, the share cache must store the directory information. In comparison, CLAM uses only the share bit in private caches and no coherence meta-data in the shared cache.

3.4 Program Synchronization

Single-copy references (Section 3.2) are used to implement synchronization. They by-pass private caches. Their updates are visible immediately, and their accesses serialized, both through the shared cache. When used to access a small amount of data, bypassing is simple and low cost.

Synchronization is necessary for a parallel program to remove data races. Single-copy loads and stores have a global ordering. Synchronization can use them to implement classical algorithms, including the 2-thread spin lock due to Peterson, n-thread mutual exclusion due to Lamport, the MCS queued lock, and barriers [10].

While a cache-wipe can be added to every volatile load or store, it is sufficient to have just one wipe for each successful synchronization. For example, if an atomic section is used, two cache-wipes are added, one at the entry and the other the exit, which ensure respectively the loading and writing back the newest values of shared data.

3.5 Cache Coherence and Memory Consistency

For CLAM, cache coherence is trivial for volatile accesses, which bypass private caches, and private accesses, which are either read only or have an effect only within a thread. We next consider only data accessed by shared references. CLAM design guarantees cache coherence for programs with no data races [10, Sec. 3.4.2]. As Scott [10, p. 46] explained, in a DRF program, "any region of code that contains no synchronization (and that does not interact with the 'outside world' via I/O or syscalls) can be thought of as atomic: it cannot–by construction–interact with other threads."

Cache coherence means that (1) changes to data will be made visible to all threads and (2) changes to the same location are seen in the same order [10, p. 12]. The first requirement is ensured by each thread calling cache-wipe which a thread does whenever it performs synchronization. The second is ensured for DRF programs, where accesses to the same shared location must be ordered by synchronization.

In CLAM cache, an eviction from a private cache to the shared cache makes a write visible. We call the order of evictions at the shared cache the *eviction order* and the program order of accesses at a private cache the *program order*. For a thread, its program order of writes may conflict with its eviction order, e.g. the shared cache may see an early change late, or more specifically, it sees a later change before it sees an earlier change. This is an *order conflict*. In memory consistency, sequential consistency means that concurrent accesses are interleaved in program order, and the same (interleaving) order is seen by all threads [7, Chap. 3]. If an order conflict is observed by a peer thread, we have inconsistency.

In a DRF execution, write-write and read-write to the same location accesses must be synchronized [10, 3.4]. CLAM synchronization ensures a cache-wipe by the first thread, which writes back all its changes, followed by a cache-wipe at the second thread, which invalidates all its local copy.

In any thread, the eviction order between synchronization cannot be observed by peer threads. When writes are visible to a peer thread, they have all been evicted. To a peer thread, it appears that the writer thread created all these changes at once. Hence, an order conflict, if it happens, cannot be observed by a peer thread. This hides any inconsistency between the access order and the eviction order and therefore guarantees memory consistency.

Block Granularity. A data block is shared if any part is accessed by a shared reference. This design will tag all shared data but may introduce "false sharing" where some portion of a shared cache block may store private data. At a cache-wipe, it may write back non-shared data unnecessarily because of false sharing. The inefficiency may be addressed at program level by padding.

Thread Migration. If a thread is moved from one core to another, it needs to call cache-wipe to evict all share data before the move.

3.6 Cooperative Cache Coherence

Coherence can be improved by programming in two ways:

Cache Draining. In the worst case, a cache-wipe operation may incur a cost proportional to the size of the private cache. It initiates bulk operations for the shared cache and may cause contention when multiple caches sync at the same time. Both are threats to parallel performance as well as robustness, predictability, and quality of service.

In CLAM, a program can always evict a cache block by accessing it and assigning it a zero lease. The cost of a cache-wipe can be reduced by *cache draining*, where a program systematically evicts shared data early and ahead of a cache-wipe. In the best case, a cache-wipe is a no-op and can return immediately.

Reference Privatization. If a program can identify shared data whose value remains up to date, it can mark their references as private, so they may stay in a private cache beyond a synchronization point and across multiple synchronization-free regions. We call this *reference privatization*. Scalar or array privatization is a well-known compiler technique that replicates data to improve parallelism [1]. Here privatization is marked for references not data, and the purpose is cache coherence, not parallelization. In reference privatization, a compiler may be developed to automatically ensure correctness.

Reference privatization opens the door for manual control. Bennett et al. [2] characterized common patterns of data sharing. A programmer may write code to support specific patterns, for example, to mark only "boundary" data as shared and to support migration by first using private references and then shared references.

Example CLAM Programming. Programmable cache works well for regular loop nests. Consider the scientific kernel Alternate Direction Integration (ADI). It is an iterative computation using mainly two arrays. Each step has two sub-steps in each of which one array is read, and the other modified. The role of the two arrays alternate in each sub-step. In basic programming for CLAM, references to both arrays are shared. Each sub-step is parallelized at the outermost loop. A barrier is added after each sub-step.

For optimization, reference leases are assigned to drain the cache before each barrier, i.e. cache draining. At the data reference where a shared array element is last accessed, zero-lease is assigned for that reference.

In addition in each loop, the read references can be marked private, i.e. reference privatization. The values in that array will be over written in the next sub-step, so evicting them is unnecessary. Moreover, keeping them in cache saves the cost of cache misses and re-loading them later for writing in the following sub-step.

3.7 Hardware Extensions

Detecting Write-write Races. A program may have errors such as unintended data races. A form of shared-cache "directory" may be added to detect write-write data races by keeping a directory in the shared cache to detect writes by more than one threads. Here, adding a directory is for debugging.

Coherence Hierarchy. A general hierarchy is a tree, where each leaf is a private cache, and each non-leaf node is a shared cache. The root of the tree is a cache shared by all. CLAM coherence can be applied to ensure coherence at the last level. At the root level, we treat each direct child of the root and its sub-tree as one private cache. All shared data blocks are tagged. Any cache-wipe would pause the operation of the entire sub-tree. All reads and writes of data at the root are totally ordered. A more scalable solution is nested coherence. A program is written with nested parallelism with nesting levels matching the cache hierarchy. Synchronization at a given level uses cache-wipe at the corresponding level of the hierarchy. Only the global synchronization incurs the full cost of synchronizing at the root.

4 Related Work

Software support may be used to implement cache coherence directly or reduce the cost of hardware cache coherence.

Software Cache Coherence. Treadmarks was widely used for distributed shared memory [5]. A program allocates shared data through a special interface, and the shared data is managed by the Treadmarks run-time system. It uses acquire and release for synchronization. The private cache in CLAM corresponds to shared data in local memory, and its local cache-wipe is both an acquire and release.

CLAM has a global data store, while Treadmarks distributes shared data. Treadmarks local memory is not managed as a cache. In CLAM private caches, shared data blocks are evicted regularly as part of cache management.

Software Support. A compiler can help by letting hardware know which data is definitely private. Li et al. [4] developed compiler analysis to identify private data and inform hardware through custom memory allocation, which is based on page-granularity classification at the time of a TLB miss [3]. CLAM differs in programming, i.e. reference rather than page based classification, and in cache design, i.e. lease based eviction. These earlier studies target distributed shared cache, which we do not consider in this paper.

Ros and Jimborean [9] developed compiler support of sequential consistency for data-race-free programs (SC for DRF). A compiler marks extended data-race-free code regions called *xDRF*, each delimited between special instructions sdrf (set SC-for-DRF coherence) and sdrf.flush. An xDRF is run with a SC-for-DRF coherence protocol, in which "every memory block sets a *toFlush* (F) bit", is "not tracked by the directory and remain(s) invisible to the (MOSEI) coherence protocol." At the end of an xDRF, all *modified* F blocks are written back by sdrf.flush. CLAM uses F bits to mark data accessed by shared references. Its cache-wipe not just writes back modified data but also invalidates all F blocks, which xDRF.flush does not.

Ros and Jimborean solved the SC-for-DRF problem for xDRF, assuming baseline support of cache coherence for the rest of the code. CLAM solves the problem for entire DRF programs, not just for xDRF regions and not using a baseline protocol. SC-for-DRF protocol does not support synchronization, whose operations are inherently "racy" and must lie outside xDRFs. CLAM provides complete cache coherence, including synchronization, which it handles with single-copy or volatile references. xDRF programming is based on regions, while CLAM is based on references.

Hardware Cache Coherence. Existing coherence-cache designs are generic and do not make assumptions about the eviction policies at private caches [7]. Hence, they can support the lease cache just as another cache policy. However, a generic design must track concurrent accesses and maintain access information for all cached data in the shared cache. CLAM supports only DRF programs. It does not track concurrent accesses and can use exclusive caching. CLAM is programmable for both caching and cache coherence. For cache eviction, CLFLUSH and CLWB exist on x86 processors, but they evict data from the entire cache hierarchy. In CLAM, leases are used to evict data only from a private cache, and the eviction is programmed at the last access, instead of using a separate flush instruction.

5 Summary

This paper presents the design of CLAM for programmable cache coherence. A program is annotated where each reference is of one of the three types: single-copy, private, or shared. The hardware maintains a share bit for data in private caches and adds a cache-wipe instruction. The software implements synchronization and cache coherence optimization. For DRF programs, the design ensures coherence and sequential consistency. The design supports both inclusive and exclusive caching of shared data and enables program optimization including cache draining, which hides the cost of cache wipes, and reference privatization, which maintains data coherence in software rather than hardware.

Acknowledgments

A LCTES reviewer informed us important related work. We thank also other reviewers and our colleagues at University of Rochester and Rochester Institute of Technology especially John Criswell, Michael Scott, Sandhya Dwarkadas, Sreepathi Pai, Alex Kneipp, and Divya Ojha for their help for us to present our work accurately and clearly. This work was supported in part by the National Science Foundation (Contract No. CCF-2217395, CCF-2114319, CNS-1909099).

References

[1] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. 1990. Adaptive Software Cache Management for Distributed Shared Memory Architectures.. In *Proceedings of the International Symposium on Computer Architecture*. 125–134.

[3] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the International Symposium on Computer Architecture*.

[4] Yong Li, Rami Melhem, and Alex K. Jones. 2012. Practically Private: Enabling High Performance CMPs through Compiler-Assisted Data Classification. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

[5] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. 1997. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 48–56.

[6] Hao Luo, Pengcheng Li, and Chen Ding. 2017. Thread Data Sharing in Cache: Theory and Measurement. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 103–115. http://dl.acm.org/citation.cfm?id=3018759

[7] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Morgan & Claypool Publishers.

[8] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. CLAM: Compiler Lease of Cache Memory. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*.

[9] Alberto Ros and Alexandra Jimborean. 2015. A Dual-Consistency Cache Coherence Protocol. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 1119–1128.

[10] Michael L. Scott. 2013. *Shared-Memory Synchronization.* Morgan & Claypool Publishers.

[11] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. Charlottesville, Virginia.