View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems

Guowei Zhang Webb Horn Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{zhanggw, webbhorn, sanchez}@csail.mit.edu

ABSTRACT

We present Coup, a technique to lower the cost of updates to shared data in cache-coherent systems. Coup exploits the insight that many update operations, such as additions and bitwise logical operations, are *commutative*: they produce the same final result regardless of the order they are performed in. Coup allows multiple private caches to simultaneously hold *update-only* permission to the same cache line. Caches with update-only permission can locally buffer and coalesce updates to the line, but cannot satisfy read requests. Upon a read request, Coup *reduces* the partial updates buffered in private caches to produce the final value. Coup integrates seamlessly into existing coherence protocols, requires inexpensive hardware, and does not affect the memory consistency model.

We apply Coup to speed up single-word updates to shared data. On a simulated 128-core, 8-socket system, Coup accelerates state-of-the-art implementations of update-heavy algorithms by up to 2.4×.

Categories and Subject Descriptors

B.3.2 [**Memory structures**]: Shared memory; C.1.4 [**Processor architectures**]: Parallel architectures

Keywords

Cache coherence, coherence protocol, commutativity

1. INTRODUCTION

Cache coherence is pervasive in shared-memory systems. However, current coherence protocols cause significantly more traffic and serialization than needed, especially with *frequent updates to shared data*. For example, consider a shared counter that is updated by multiple cores. On each update, the updating core first fetches an exclusive copy of the counter's cache line into its private cache, invalidating all other copies, and modifies it locally using an atomic operation such as fetch-and-add, as shown in Fig. 1a. Each update incurs significant *traffic* and *serialization*: traffic to fetch the line and invalidate other copies, causing the line to ping-pong among updating cores; and serialization because only one core can perform an update at a time.

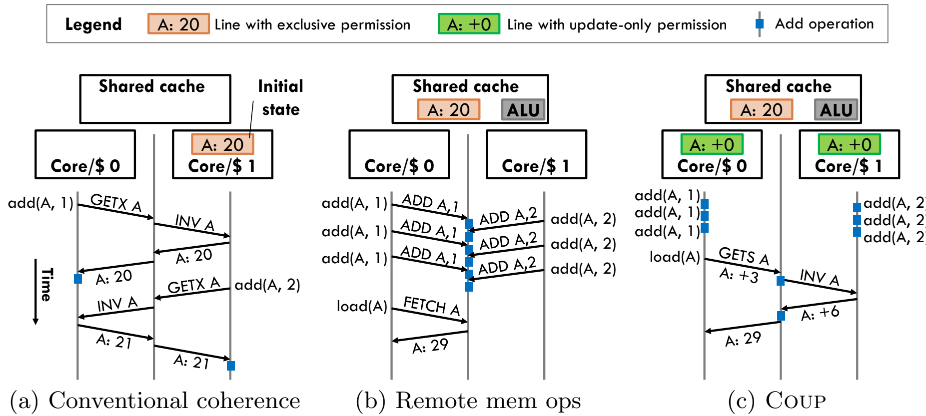
Prior work has proposed hardware and software techniques to reduce traffic and serialization of updates in parallel systems. In hardware, prior work has mainly focused on *remote memory operations* (RMOs) [29, 30, 57, 68]. RMO schemes send updates to a single memory controller or shared cache bank instead of having the line ping-pong among multiple private caches, as shown in Fig. 1b. Although RMOs reduce the cost of updates, they still cause significant global traffic and serialization, and often make reads slower, as remote reads may be needed to preserve consistency [39, 57].

In this work we leverage two key insights to reduce the cost of updates further. First, many update operations need not read the data they update. Second, update operations are often *commutative*, and can be performed in any order before the data is read. In our shared counter example, multiple additions from different cores can be buffered, coalesced, and delayed until the counter's line is next read. Commutative updates are common in other contexts beyond this simple example.

Two obstacles prevent these optimizations in current protocols. First, conventional coherence protocols support only two primitive operations, reads and writes, so commutative updates must be expressed as a read-modify-write sequence. Second, these protocols do not decouple read and write permissions. Instead, they enforce the *single-writer, multiple-reader invariant*: at a given point in time, a cache line may either have at most one sharer with *read-and-write* permission, or multiple sharers with read-only permission [3, 59].

We propose Coup (Sec. 3), a general technique that extends coherence protocols to allow *local and concurrent commutative updates*. Coup decouples read and write permissions, and introduces commutative-update primitive operations, in addition to reads and writes. With Coup, multiple caches can acquire a line with *update-only* permission, and satisfy commutative-update requests locally, buffering and coalescing updates. On a read request, the coherence protocol gathers all the local updates and *reduces* them to produce the correct value

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-48, December 05–09, 2015, Waikiki, HI, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4034-2/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830774



(a) Conventional coherence (b) Remote mem ops (c) Coup

Figure 1: Example comparing the cost of commutative updates under three schemes. Two cores add values to a single memory location, A. (a) With conventional coherence protocols, A's fetches and invalidations dominate the cost of updates. (b) With remote memory operations, cores send updates to a fixed location, the shared cache in this case. (c) With Coup, caches buffer and coalesce updates locally, and reads trigger a reduction of all local updates to produce the actual value.

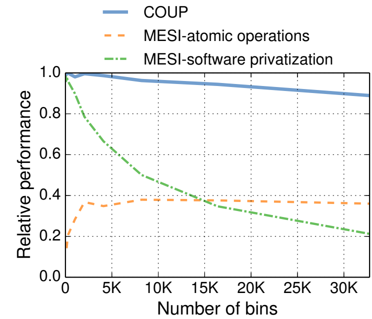


Figure 2: Performance of parallel histogram implementations using atomics, software privatization, and Coup. More bins reduce contention and increase privatization overheads, favoring atomics. Coup does not suffer these overheads, so it outperforms both software implementations.

before granting read permission. For example, multiple cores can concurrently add values to the same counter. Updates are held in their private caches as long as no core reads the current value of the counter. When a core reads the counter, all updates are added to produce the final value, as shown in Fig. 1c.

Coup confers significant benefits over RMOs, especially when data receives several consecutive updates before being read. Moreover, Coup maintains full cache coherence and does not affect the memory consistency model. This makes Coup easy to apply to current systems and applications. We demonstrate Coup's utility by applying it to improve the performance of *single-word update operations*, which are currently performed with expensive atomic read-modify-write instructions.

Coup also completes a symmetry between hardware and software schemes to reduce the cost of updates. Broadly, software techniques use either *delegation* or *privatization*. Delegation schemes send updates to a single thread [11, 12]. Privatization schemes lower the cost of commutative updates by using thread-local variables [8, 18, 46]: each thread updates its local variable, and reads require reducing the per-thread variables. Just as remote memory operations are the hardware counterpart to delegation, Coup *is the hardware counterpart to privatization*. Coup has two benefits over software privatization. First, transitions between read-only and update-only modes are much faster, so Coup remains practical in many scenarios where software privatization requires excessive synchronization. Second, privatization's thread-local copies increase memory footprint and add pressure to shared caches, while Coup does not.

In this work, we make the following contributions:

- We present Coup, a technique that extends coherence protocols to support concurrent commutative updates (Sec. 3). We show that Coup preserves coherence and consistency, and imposes small verification costs.
- We identify several update-heavy parallel applications where current techniques have clear shortcomings (Sec. 4), and discuss how Coup addresses them.
- We evaluate Coup under simulation, using single- and multi-socket systems (Sec. 5). At 128 cores, Coup improves the performance of update-heavy benchmarks by 4%–2.4×, and reduces traffic by up to 20×.

In summary, Coup shows that extending coherence protocols to leverage the semantics of commutative updates can substantially improve performance without sacrificing the simplicity of cache coherence.

2. BACKGROUND

We now discuss prior hardware and software techniques that reduce the cost of updates to shared data.

2.1 Hardware Techniques

Remote memory operations (RMOs) are the most closely related scheme to Coup. Rather than caching lines to be updated, update operations are sent to a fixed location. The NYU Ultracomputer [29] proposed implementing atomic fetch-and-add using adders in network switches, which could coalesce multiple requests on their way to memory. The Cray T3D [34], T3E [57], and SGI Origin [42] implemented RMOs at the memory controllers, while TilePro64 [30] and recent GPUs [63] implement RMOs in shared caches. Prior work has also proposed adding caches to memory controllers to accelerate RMOs [68] and data-parallel RMOs [5].

Coup has two key advantages over RMOs. First, while RMOs avoid ping-ponging cache lines, they still require sending every update to a shared, fixed location, causing global traffic. RMOs are also limited by the throughput of the single updater. For example, in Fig. 1b, frequent remote-add requests drive the shared cache's ALU near saturation. By contrast, Coup buffers and coalesces up-

dates in local caches, avoiding hotspots. Second, strong consistency models are challenging to implement with RMOs, as it is harder to constrain memory operation order. For example, TSO requires making stores globally visible in program order, which is feasible with local store buffers, but much more complicated when stores are also performed by remote updaters. As a result, most implementations provide weakly-consistent RMOs. Timestamp-based order validation [39, §5] allows strong consistency with RMOs, but it is involved. By contrast, COUP performs all memory operations locally, making consistency easy to maintain.

Note that COUP's advantages come at the cost of a more restricted set of operations: COUP is limited to commutative updates, while RMOs support non-commutative operations such as fetch-and-add and compare-and-swap. Also, COUP significantly outperforms RMOs only if data is reused (i.e., updated or read multiple times before switching between read- and update-only modes). This is often the case in real applications (Sec. 4).

2.2 Software Techniques

Conventional shared-memory programs update shared data using atomic operations for single-word updates, or normal reads and writes with synchronization (e.g., locks or transactions) for multi-word updates. Many software optimizations seek to reduce the cost of updates. Though often presented in the context of specific algorithms, we observe they are instances of two general techniques: *delegation* and *privatization*. We discuss these techniques here, and present specific instances in Sec. 4.

Delegation schemes divide shared data among threads and send updates to the corresponding thread, using shared-memory queues [11] or active messages [55, 61]. Delegation is common in architectures that combine shared memory and message passing [55, 64] and in NUMA-aware data structures [11, 12]. Delegation is the software counterpart to RMOs, and is subject to the same tradeoffs: it reduces data movement and synchronization, but incurs global traffic and serialization.

Privatization schemes exploit commutative updates. These schemes buffer updates in thread-private storage, and require reads to reduce these thread-private updates to produce the correct value. Privatization is most commonly used to implement reduction variables efficiently, often with language support (e.g., reducers in MapReduce [22], OpenMP pragmas, and Cilk Plus hyperobjects [28]). Privatization is generally used when updates are frequent and reads are rare.

Privatization is the software counterpart to COUP, and is subject to similar tradeoffs: it is limited to commutative updates, and works best when data goes through long update-only phases without intervening reads. Unlike COUP, privatization has two major sources of overhead. First, while COUP is about as fast as a conventional protocol if a line is updated only once before being read (Fig. 1c), software reductions are much slower, making finely-interleaved reads and updates inefficient. Second, with N threads, privatized variables increase memory footprint by a factor of N. This makes naive pri-

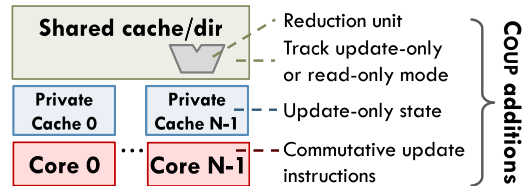


Figure 3: Summary of additions and modifications needed to support Coup.

vatization impractical in many contexts (e.g., reference counting). Dynamic privatization schemes [18, 46, 65] can lessen space overheads, but add time overheads.

These overheads often make privatization underperform conventional updates. For instance, Jung et al. [32] propose parallel histogram implementations using both atomic operations and privatization. These codes process a set of input values, and produce a histogram with a given number of bins. Jung et al. note that privatization is desirable with a few output bins, but works poorly with many bins, as the reduction phase dominates and hurts locality. Fig. 2 shows this tradeoff. It compares the performance of histogram implementations using atomic fetch-and-add, privatization, and COUP, when running on 64 cores (see Sec. 5 for methodology details). In this experiment, all schemes process a large, fixed number of input elements. Each line shows the performance of a given implementation as the number of output bins (x-axis) changes from 32 to 32 K. Performance is reported relative to COUP's at 32 bins (higher numbers are better). While the costs of privatization impose a delicate tradeoff between both software implementations, COUP robustly outperforms both.

3. EXTENDING CACHE COHERENCE TO SUPPORT COMMUTATIVE UPDATES

3.1 Coup Example: Extending MSI

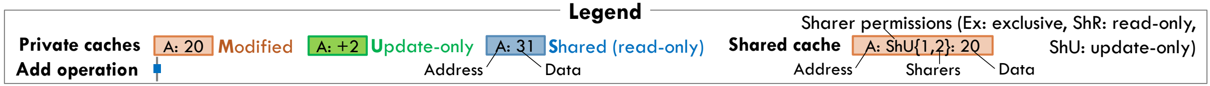
We first present the main concepts and operation of COUP through a concrete, simplified example. Consider a system with a single level of private caches, kept coherent with the MSI protocol. This system has a single shared last-level cache with an in-cache directory. It implements a single commutative-update operation, addition. Finally, we restrict this system to use single-word cache blocks. We will generalize COUP to other protocols, operations, and cache hierarchies in Sec. 3.2.

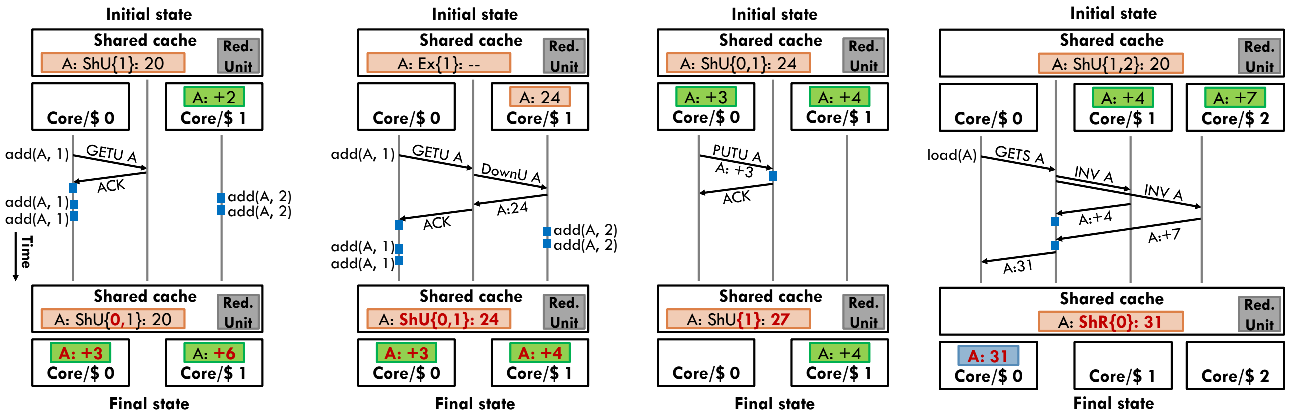
3.1.1 Structural changes

COUP requires modest changes to hardware structures, summarized in Fig. 3 and described below.

Commutative-update instructions: In most ISAs, COUP needs additional instructions that let programs convey commutative updates, as conventional atomic instructions (e.g., fetch-and-add) return the latest value of the data they update. In this case, we add a *commutative-addition* instruction, which takes an address and a single input value, and does not write to any register.

Some ISAs may not need additional instructions. For instance, the recent Heterogeneous System Architecture (HSA) includes atomic-no-return instructions that do not





(a) Upgrade to U caused by commutative-update request (b) Downgrade caused by commutative-update request (c) Partial reduction caused by eviction (d) Full reduction caused by read request

Figure 5: MUSI protocol operation: (a) granting update-only (U) state; (b) downgrade from M to U due to an update request from another core; (c) partial reduction caused by an eviction from a private cache; and (d) full reduction caused by a read request. Each diagram shows the initial and final states in the shared and private caches.

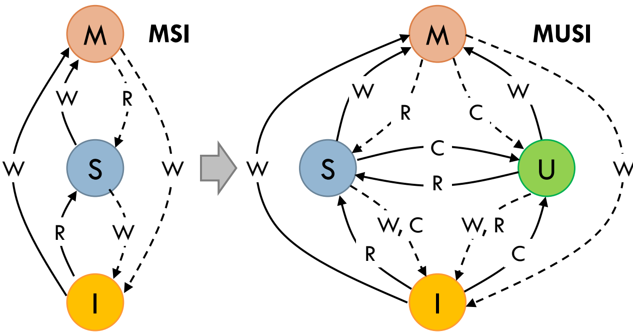


Figure 4: State-transition diagrams of MSI and MUSI protocols. For clarity, diagrams omit actions that do not cause a state transition (e.g., R requests in S).

return the updated value [2]. While these instructions were likely introduced to reduce the cost of RMOs, Coup could use them directly.

Update-only permission: Coup extends MSI with an additional state, *update-only (U)*, and a third type of request, *commutative update (C)*, in addition to conventional reads (R) and writes (W). We call the resulting protocol MUSI. Fig. 4 shows MUSI's state-transition diagram for private caches. MUSI allows multiple private caches to hold read-only permission to a line and satisfy read requests locally (S state); multiple private caches to hold update-only permission to a line and satisfy commutative-update requests locally (U state); or at most a single private cache to hold exclusive permission to a line and satisfy all types of requests locally (M state). By allowing M to satisfy commutative-update requests, interleaved updates and reads to *private* data are as cheap as in MSI.

MUSI's state-transition diagram shows a clear symmetry between S and U: all transitions caused by R/C requests in and out of S match those caused by C/R requests in and out of U. We will exploit this symmetry in Sec. 3.4 to simplify our implementation.

Directory state: Conventional directories must track both the sharers of each line (using a bit-vector or other techniques [13, 53, 66]), and, if there is a single sharer, whether it has exclusive or read-only permission. In Coup, the directory must track whether sharers have exclusive, read-only, or update-only permission. The sharers bit-vector can be used to track both multiple readers or multiple updaters, so MUSI requires only one extra bit per directory tag.

Reduction unit: Though cores can perform local updates, the memory system must be able to perform reductions. Thus, Coup adds a reduction unit to the shared cache, consisting of an adder in this case.

3.1.2 *Protocol operation*

Performing commutative updates: Both the M and U states provide enough permissions for private caches to satisfy update-only requests. In M, the private cache has the actual data value; in U, the cache has a partial update. In either case, the core can perform the update by atomically reading the data from the cache, modifying it (by adding the value specified by the commutative-add instruction) and storing the result in the cache. The

cache cannot allow any intervening operations to the same address between the read and the write. This scheme can reuse the existing core logic for atomic operations. We assume this scheme in our implementation, but note that alternative implementations could treat commutative updates like stores to improve performance (e.g., using update buffers similar to store buffers and performing updates with an ALU at the L1).

Entering the U state: When a cache has insufficient permissions to satisfy an update request (I or S states), it requests update-only permission from the directory. The directory invalidates any copies in S, or downgrades the single copy in M to U, and grants update-only permission to the requesting cache, which transitions to U. Thus, there are two ways a line can transition into the U state: by requesting update-only permission to satisfy a request from its own core, as shown in Fig. 5a; or by being downgraded from M, as shown in Fig. 5b.

When a line transitions into U, its contents are always initialized to the *identity element*, 0 for commutative addition. This is done even if the line had valid data. This avoids having to track which cache holds the original data when doing reductions. However, reductions require reading the original data from the shared cache.

Leaving the U state: Lines can transition out of U due to either evictions or read requests.

Evictions initiated by a private cache (to make space for a different line) trigger a *partial reduction*, shown in Fig. 5c: the evicting cache sends its partial update to the shared cache, which uses its reduction unit to aggregate it with its local copy.

The shared cache may also need to evict a line that private caches hold in U. This triggers a *full reduction*: all caches with update-only permission are sent invalidations, reply with their partial updates, and the shared cache uses its reduction unit to aggregate all partial updates and its local copy, producing the final value.

Finally, read requests from any core also trigger a full reduction, as shown in Fig. 5d. Depending on the latency and throughput of the reduction unit, satisfying a read request can take somewhat longer than in conventional protocols. Hierarchical reductions can rein in reduction overheads with large core counts (Sec. 3.2). In our evaluation, we observe that reduction overheads are small compared to communication latencies.

3.2 Generalizing Coup

We now show how to generalize Coup to support multiple operations, larger cache blocks, other protocols, and deeper cache hierarchies.

Multiple operations: Formally, Coup can be applied to any *commutative semigroup* $(G, \circ)$.[1] For example, G can be the set of 32-bit integers, and $\circ$ can be addition, multiplication, *and*, *or*, *xor*, *min*, or *max*.

Supporting multiple operations in the system requires minor changes. First, additional instructions are needed to convey each type of update. Second, reduction units must implement all supported operations. Third, the directory and private caches must track, for each line in U state, what type of operation is being performed. Fourth, Coup must serialize commutative updates of different types, because they do not commute in general (e.g., $+$ and $*$ do not commute with each other). This can be accomplished by performing a full reduction every time the private cache or directory receives an update request of a type different from the current one.

[1] $(G, \circ)$ is a commutative semigroup iff $\circ : G \times G \rightarrow G$ is a binary, associative, commutative operation over elements of set G, and G is closed under $\circ$.

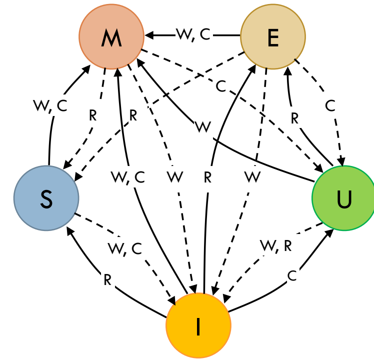


Figure 6: State-transition diagram of MEUSI. Just as MESI grants E to a read request if a line is unshared, MEUSI grants M to an update request if a line is unshared. For clarity, the diagram omits actions that do not cause a state transition (e.g., C requests in U).

Larger cache blocks: Supporting multi-word blocks is trivial if $(G, \circ)$ has an *identity element* (formally, this means $(G, \circ)$ is a commutative monoid). The identity element produces the same value when applied to any element in G. For example, the identity elements for addition, multiplication, *and*, and *min* are 0, 1, all-ones, and the maximum representable integer, respectively.

All the operations we implement in this work have an identity element. In this case, it is sufficient to initialize every word of the cache block to the identity element when transitioning to U. Reductions perform element-wise operations even on words that have received no updates. Note this holds *even if those words do not hold data of the same type*, because applying $\circ$ on the identity element produces the same output, so it does not change the word's bit pattern. Alternatively, reduction units could skip operating on words with the identity element.

In general, not all operations may have an identity element. In such cases, the protocol would require an extra bit per word to track uninitialized elements.

Finally, note we assume that data is properly aligned. Supporting commutative updates to unaligned data would require more involved mechanisms to buffer partial updates. If the ISA allows unaligned accesses, they can be performed as normal read-modify-writes.

Other protocols: Coup can extend protocols beyond MSI. Fig. 6 shows how MESI [48] is extended to MEUSI, which we use in our evaluation. Note that update requests enjoy the same optimization that E introduces for read-only requests: if a cache requests update-only permission for a line and no other cache has a valid copy, the directory grants the line directly in M.

Deeper cache hierarchies: COUP can operate with multiple intermediate levels of caches and directories. COUP simply requires a reduction unit at each intermediate level that has multiple children that can issue update requests. For instance, a system with private per-core L1s and L2s and a fully shared L3 needs reduction units only at L3 banks. However, if each L2 was shared by two or more L1Ds, a reduction unit would be required in the L2s as well.

Hierarchical organizations lower the latency of reductions in COUP, just as they lower the latency of sending and processing invalidations in conventional protocols: on a full reduction, each intermediate level aggregates all partial updates from its children before replying to its parent. For example, consider a 128-core system with a fully-shared L4 and 8 per-socket L3s, each shared by 16 cores. In this system, a full reduction of a line shared in U state by all cores has $8 + 16 = 24$ operations in the critical path—far fewer than the 128 operations that a flat organization would have, and not enough to dominate the cost of invalidations.

Other contexts: We focus on single-word atomic operations and hardware cache coherence, but note that COUP could apply to other contexts. For example, COUP could be used in software coherence protocols (e.g., in distributed shared memory), and can support more sophisticated operations such as insertion and deletion into sets. We leave such extensions to future work.

3.3 Coherence and Consistency

COUP maintains cache coherence and does not change the consistency model.

Coherence: A memory system is coherent if, for each memory location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and that obeys two invariants [20, §5.1.1]:[2]

1. Operations issued by each core occur in the order in which they were issued to the memory system by that core.
2. The value returned by each read operation is the value written to that location in the serial order.

In COUP, a location can be in exclusive, read-only, or update-only modes. The baseline protocol that COUP extends already enforces coherence in and between exclusive and read-only modes. In update-only mode, multiple cores can concurrently update the location, but because updates are commutative, *any serial order* we choose produces the same execution result. Thus, the first invariant is trivially satisfied. Moreover, transitions from update-only to read-only or exclusive modes propagate all partial updates and make them visible to the next reader. Thus, the next reader always observes the last value written to that location, satisfying the second property. Therefore, COUP maintains coherence.

Consistency: As long as the system restricts the order of memory operations as strictly for commutative updates as it does for stores, COUP does not affect the consistency model. In other words, it is sufficient for the memory system to consider commutative updates as being equivalent to stores. For instance, by having store-load, load-store, and store-store fences apply to commutative updates as well, systems with relaxed memory models need not introduce new fence instructions.

[2]Others reason about coherence using the *single-writer, multiple-reader* and the *data-value* invariants [59], which are sufficient but not necessary. COUP does not maintain the single-writer, multiple-reader invariant.

3.4 Implementation and Verification Costs

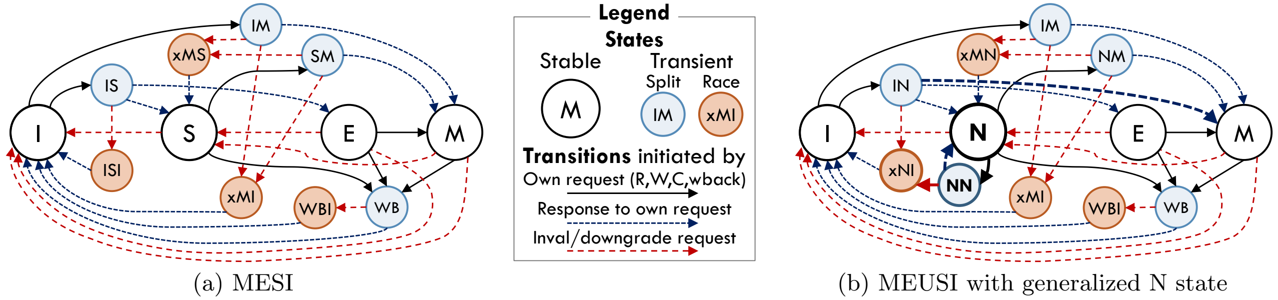
While we have presented COUP in terms of stable states, realistic protocols implement coherence transactions with additional transient states and are subject to races, which add complexity and hinder verification. By studying full implementations of MESI and MEUSI, we show that COUP requires a minimal number of transient states and adds modest verification costs.

We first implement MESI protocols for two- and three-level cache hierarchies. Our implementations work on networks with unordered point-to-point communication, and use two virtual networks without any message buffering at the endpoints. In the two-level protocol, the L1 coherence controller has 12 states (4 stable, 8 transient), and the L2 has 6 states (3 stable, 3 transient). Fig. 7a shows the state-transition diagram of the more complex L1 cache. In the three-level protocol, the L1 has 14 states (4 stable, 10 transient), the L2 has 38 (9 stable, 29 transient), and the L3 has 6 (3 stable, 3 transient).

Generalized non-exclusive state: While we have introduced U as an additional state separate from S, both have a strong symmetry and many similarities. In fact, reads are just another type of commutative operation. We leverage this insight to simplify COUP's implementation by integrating S and U under a single, generalized *non-exclusive* state, N. This state requires minor extensions over the machinery already described in Sec. 3.2 to support multiple commutative updates.

Multiple caches can have a copy of the line in N, but all copies must be under the same operation type, which can be read-only or one of the possible commutative updates. An additional field per line tracks its operation type when in N. Non-exclusive and downgrade requests are tagged with the desired operation type. E and M can satisfy all types of requests; commutative updates cause an E→M transition. N can satisfy non-exclusive requests of the same type, but requests of a different type trigger an invalidation (if starting from read-only) or a reduction (if starting from a commutative-update type) and cause a type switch. Invalidations and reductions involve the same request-reply sequence, so they can use the same transient states.

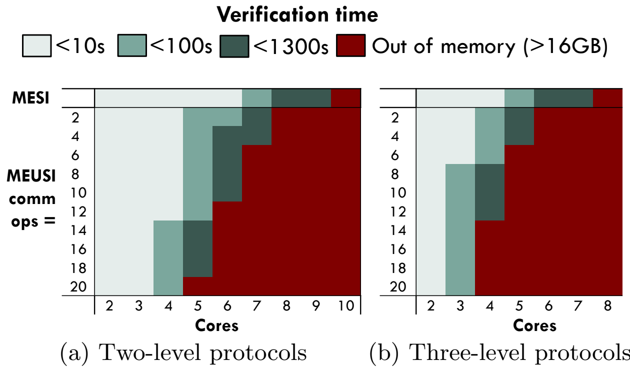
Implementing two-level MEUSI this way requires 13 states in the L1 and 6 states in the L2. Compared to two-level MESI, MEUSI introduces only one extra L1 transient state. Fig. 7b shows the L1's state-transition diagram, which is almost identical to MESI's. The new transient state, *NN*, is used when moving between operation types (e.g., from read-only to commutative-add or from commutative-and to commutative-or). Our three-level MEUSI protocol is also similar to three-level MESI:



(a) MESI

(b) MEUSI with generalized N state

Figure 7: Coup implementation: (a) full state-transition diagram for the L1 cache on the baseline two-level MESI protocol; (b) corresponding MEUSI state-transition diagram. The non-exclusive state, N, generalizes S and U, and requires only an extra transient state and four transitions over MESI.



(a) Two-level protocols (b) Three-level protocols

Figure 8: Coup exhaustive verification costs for two- and three-level protocols. Costs grow much more quickly with the number of cores and levels than the number of commutative updates.

the L1 has 15 states (one more transient than MESI, *NN*), the L2 has 43 (five more transient states than MESI, which, similarly to *NN*, implement transitions between operation types), and the L3 has 6.

Verification costs: We use Murphi [24] to verify MESI and MEUSI. We adopt common simplifications to limit the state space, modeling caches with a single 1-bit line; self-eviction rules model a limited capacity. In three-level protocols, we model systems with a single L2 and a single L3, and simulate traffic from other L2s with L3-issued invalidation and downgrade rules. Even then, Murphi can only verify systems of up to 4-8 cores, a well-known limitation of this approach [69, 70].

MEUSI's verification costs grow more quickly with the number of cores and levels than the number of commutative operations. Fig. 8 reports the verification times for two- and three-level MESI and MEUSI protocols supporting 2–20 commutative-update types. We run Murphi on a Xeon E5-2670, and limit it to 16 GB of memory. Murphi can exhaustively verify MESI up to 7-9 cores and MEUSI up to 3-7 cores depending on the number of levels and commutative updates. This shows that MEUSI can be effectively verified up to a large number of commutative updates. Moreover, just as protocol designers assume that modeling a few cores provide reasonable coverage, verifying up to a few commutative operations should be equally reasonable.

4. MOTIVATING APPLICATIONS

In this work, we apply Coup to accelerate single-word updates to shared data. To guide our design, we first study under what circumstances Coup is beneficial over state-of-the-art software techniques, and illustrate these circumstances with specific algorithms and applications.

As discussed in Sec. 2, Coup is the hardware counterpart to privatization. Privatization schemes create several replicas of variables to be updated. Each thread updates one of these replicas, and threads synchronize to reduce all partial updates into a single location before the variable is read.

In general, Coup outperforms prior software techniques if *either* of the following two conditions holds:

- Reads and updates to shared data are finely interleaved. In this case, software privatization has large overheads due to frequent reductions, while Coup can move a line from update-only mode to read-only mode at about the same cost as a conventional invalidation. Thus, privatization needs many updates per core and data value to amortize reduction overheads, while Coup yields benefits with as little as two updates per update-only epoch.
- A large amount of shared data is updated. In this case, privatization significantly increases memory footprint and puts more pressure on shared caches.

We now discuss several parallel patterns and applications that have these properties.

4.1 Separate Update- and Read-Only Phases

Several parallel algorithms feature long phases where shared data is either only updated or only read. Privatization techniques naturally apply to these algorithms.

Reduction variables: Reduction variables are objects that are updated by multiple iterations of a loop using a binary, commutative operator (a reduction operator) [49, 50], and their intermediate state is not read. Reduction variables are natively supported in parallel programming languages and libraries such as HPF [37], MapReduce [22], OpenMP [25], TBB [51], and Cilk Plus [28]. Prior work in parallelizing compilers has developed a wide array of techniques to detect and exploit reduction variables [31,49,50]. Reductions are commonly implemented using parallel reduction trees, a form of

privatization. Each thread executes a subset of loop iterations independently, and updates a local copy of the object. Then, in the reduction phase, threads aggregate these copies to produce a single output variable.

Reduction variables can be small, for example when computing the mean or maximum value of an array. In these cases, the reduction variable is a single scalar, the reduction phase takes negligible time, and COUP would not improve performance much over software reductions.

Reduction variables are often larger structures, such as arrays or matrices. For example, consider a loop that processes a set of input values (e.g., image pixels) and produces a histogram of these values with a given number of bins. In this case, the reduction variable is the whole histogram array, and the reduction phase can dominate execution time [32], as shown in Fig. 2. Yu and Rauchwerger [65] propose several adaptive techniques to lower the cost of reductions, such as using per-thread hash tables to buffer updates, avoiding full copies of the reduction variable. However, these techniques add time overheads and must be applied selectively [65]. Instead, COUP achieves significant speedups by maintaining a single copy of the reduction variable in memory, and overlapping the loop and reduction phases.

Reduction variables and other update-only operations often use floating-point data. For example, depending on the format of the sparse matrix, sparse matrix-vector multiplication can require multiple threads to update overlapping elements of the output vector [5]. However, floating-point operations are not associative or commutative, and the order of operations can affect the final result in some cases [60]. Common parallel reduction implementations are non-deterministic, so we choose to support floating-point addition in COUP. Implementations desiring reproducibility can use slower deterministic reductions in software [23].

Ghost cells: In iterative algorithms that operate on regular data, such as structured grids, threads often work on disjoint chunks of data and only need to communicate updates to threads working on neighboring chunks. A common technique is to buffer updates to boundary cells using ghost or halo cells [35], private copies of boundary cells updated by each thread during the iteration and read by neighboring threads in the next iteration. Ghost cells are another form of privatization, different from reductions in that they capture point-to-point communication. COUP avoids the overheads of ghost cells by letting multiple threads update boundary cells directly.

The ghost cell pattern is harder to apply to iterative algorithms that operate on irregular data, such as PageRank [47, 56]. In these cases, partitioning work among threads to minimize communication can be expensive, and is rarely done on shared-memory machines [56]. By reducing the cost of concurrent updates to shared data, COUP helps irregular iterative algorithms as well.

4.2 Interleaved Updates and Reads

Several parallel algorithms read and update shared data within the same phase. Unlike the applications in Sec. 4.1, software privatization is rarely used in these cases, as software would need to detect data in update-only mode and perform a reduction before each read. By contrast, COUP transparently switches cache lines between read-only and update-only modes in response to accesses, improving performance even with a few consecutive updates or reads.

Graph traversals: High-performance implementations of graph traversal algorithms such as breadth-first search (BFS) encode the set of visited nodes in a bitmap that fits in cache to reduce memory bandwidth [4, 15]. The first thread that visits a node sets its bit, and threads visiting neighbors of the node read its bit to find whether the node needs to be visited.

Existing implementations use atomic-*or* operations to update the bitmap [4], or use non-atomic load-*or*-store sequences, which reduce overheads but miss updates, causing some nodes to be visited multiple times [15]. In both cases, updates from multiple threads are serialized. In contrast, COUP allows multiple concurrent updates to bits in the same cache line.

Besides graph traversals, commutative updates to bitmaps are common in other contexts, such as recently-used bits in page replacement policies [19], buddy memory allocation [36], and other graph algorithms [40].

Reference counting: Reference counting is a common automatic memory management technique. Each object has a counter to track the number of active references. Threads increment the object's counter when they create a reference, and decrement and read the counter when they destroy a reference, When the reference count reaches zero, the object is garbage-collected.

Using software techniques to reduce reference-counting overheads is a well-studied problem [17, 18, 26, 45]. Scalable Non-Zero Indicators (SNZIs) [26] reduce the cost of non-zero checks. SNZIs keep the global count using a tree of counters. Threads increment and decrement different nodes in the tree, and may propagate updates to parent nodes. Readers just need to check the root node to determine whether the count is zero. SNZIs make non-zero checks fast and allow some concurrency in increments and decrements, but add significant space and time overheads, and need to be carefully tuned.

Refcache [17] delays and batches reads to reference counts, which allows it to use privatization. Threads maintain a software cache of reference counter deltas, which are periodically flushed to the global counter. When the global counter stays at zero for a sufficiently long time, the true count is known to be zero and the object is deallocated. This approach reduces reference-counting overheads, but delayed deallocation hurts memory footprint and locality.

COUP enables shared reference counters with no space overheads and less coherence traffic than shared counters. COUP also allows delayed reference counting as in Refcache without a software cache (Sec. 5.4).

5. EVALUATION

5.1 Methodology

Modeled systems: We perform microarchitectural, execution-driven simulation using zsim [54]. We eval-

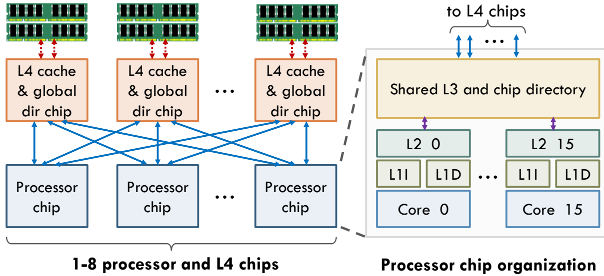


Figure 9: Architecture of the simulated system.

Processor chip	**Cores**	1–128 cores, 16 cores/processor chip, x86-64 ISA, 2.4 GHz, Nehalem-like OOO [54]
	L1 caches	32 KB, 8-way set-associative, split D/I, 4-cycle latency
	L2 caches	256 KB private per-core, 8-way set-associative, inclusive, 7-cycle latency
	L3 caches	32 MB, 8 banks, 16-way set-associative, inclusive, 27-cycle latency, in-cache directory
	Off-chip network	Dancehall topology, 40-cycle point-to-point links between each processor and L4 chip
	L4 & dir chip	128 MB, 8 banks/chip, 16-way set-associative, inclusive, 35-cycle latency, in-cache directory
	Coherence	MESI/MEUSI, 64 B lines, no silent drops
	Main memory	4 DDR3-1600-CL10 channels per L4 chip, 64-bit bus, 2 ranks/channel

Table 1: Configuration of the simulated system.

uate single- and multi-socket systems with up to 128 cores and a four-level cache hierarchy, shown in Fig. 9. Table 1 details the configuration of these systems. Each processor chip has 16 cores. Each core has private L1s and a private L2, and all cores in the chip share a banked L3 cache with an in-cache directory. The system supports up to 8 processor chips, connected in a dancehall topology to the same number of L4 chips. Each of these chips contains a slice of the L4 cache and global in-cache directory, and connects to a fraction of main memory. This organization is similar to the IBM z13 [62].

We compare MESI and MEUSI (Fig. 6). With MEUSI, each L3 and L4 bank has a reduction unit. We perform hierarchical reductions as described in Sec. 3.2: on a full reduction, each L3 bank invalidates all its children, aggregates their partial updates, and sends a single response to the L4 controller.

Coup operations and data types: We add support for eight commutative-update types:

- Addition of 16, 32, and 64-bit integers, and 32 and 64-bit floating-point values.
- AND, OR, and XOR bitwise logical operations on 64-bit words.

We observe multiplication update-only operations are rare, so we do not support multiplication. We also observe *min* and *max* are often used with scalar reduction variables (e.g., to find the extreme values of an array). Coup would provide a negligible benefit for scalar reductions, as discussed in Sec. 4.1. Thus, we do not support *min* or *max*. Finally, we support a single word size for bitwise operations, because this suffices to express updates to bitmaps of any size (smaller or larger).

	Input set	Comm ops	Seq run-time
hist	GRiN [1], 512 bins	32b int add	2720 Mcycles
spmv	rma10 [21]	64b FP add	94 Mcycles
fldanim	simlarge [9]	32b FP add	5930 Mcycles
pgrank	Wikipedia (2007) [21]	64b int add	2850 Mcycles
bfs	cage15 [21, 44]	64b OR	5764 Mcycles

Table 2: Benchmark characteristics.

Commutative-update instructions: We add an instruction for each supported operation and data type. Each instruction takes two register inputs, with the address to be updated and the value to apply, and produces no register output. We encode these instructions using x86-64 no-ops that are never emitted by the compiler.

The x86 (TSO) memory model specifies that atomic instructions have an implicit store-load fence [58]; for consistency, we also add an implicit fence to commutative-update instructions. We implement conventional atomic operations and commutative updates using a four-μop sequence: load-linked, execute (in one of the appropriate execution ports), store-conditional, and store-load fence.

Reduction unit organization: Since functional units for the required operations are relatively simple, we assume a 2-stage pipelined, 256-bit ALU (4× 64-bit lanes). This ALU has a throughput of one full 64-byte cache line per two clock cycles, and a latency of three clock cycles per line. We explore the sensitivity to reduction unit throughput in Sec. 5.5.

Hardware overheads: In summary, our Coup implementation introduces modest overheads:

1. Eight additional commutative-update instructions.
2. Four bits per line to encode the non-exclusive operation type, either read-only or one of eight commutative-update types (Sec. 3.4).
3. One reduction unit per L3 and L4 bank.

Workloads: We use a set of five multithreaded benchmarks that cover the cases described in Sec. 4:

- **hist** is the TBB-based OpenCV [10] histogramming program (version 2.4.11).
- **spmv** is a sparse matrix-vector multiplication kernel, where the matrix is encoded in compressed sparse column (CSC) format. CSC requires multiple threads to perform scattered additions to the output vector. Other input formats, such as EBE, also cause scattered adds in matrix-vector multiplication [5].
- **fluidanimate**, from the PARSEC suite [9], is a regular iterative algorithm (Sec. 4.1). We optimize the default implementation, which uses locks to guard updates to shared cells, to use atomic operations instead.
- **pgrank** is a PageRank implementation similar to the shared-memory optimized version of Satish et al. [56].
- **bfs** is a parallel breadth-first search algorithm. Our implementation extends PBFS [44] with a visited bitvector to reduce memory traffic (Sec. 4.2), similar to state-of-the-art approaches [4, 15].

Table 2 details the input sets, commutative-update operations used, and sequential run-time of each benchmark.

All the baseline benchmark implementations use atomic operations. We also compare against a privatization-based variant of `hist` (implemented using TBB reduc-

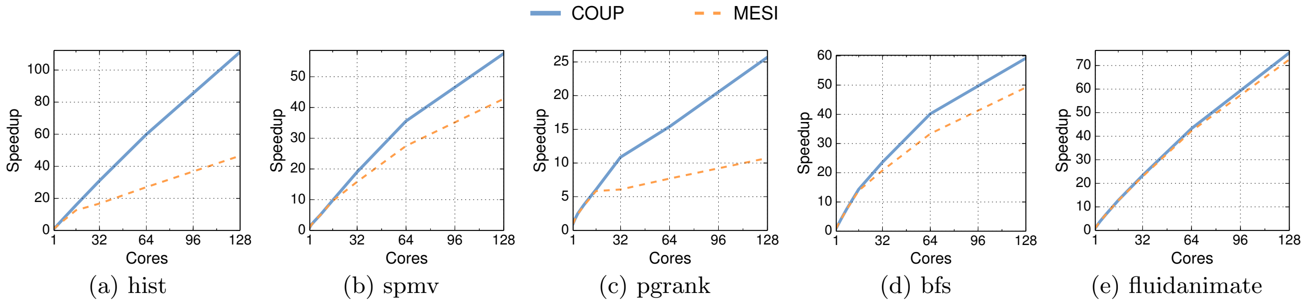


Figure 10: Per-application speedups of Coup and MESI on 1–128 cores (higher is better).

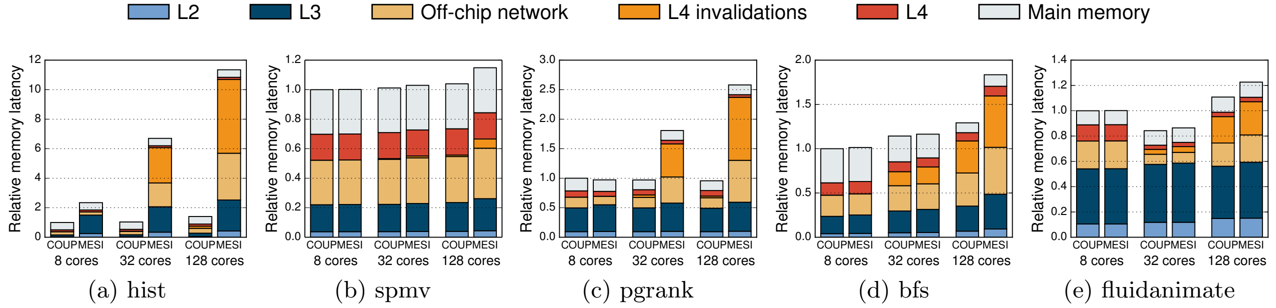


Figure 11: Breakdown of average memory access latency (AMAT) of Coup and MESI on 8, 32, and 128-core systems. AMAT is normalized to Coup's at 8 cores (lower is better).

tions) in Sec. 5.3, and develop reference-counting microbenchmarks to compare Coup against SNZI and Refcache in Sec. 5.4.

We report results on 1–128 cores. We scale the number of processor and L4 chips on runs with more cores (e.g., 1-core runs use a single processor and L4 chip, 32-core runs use two of each, and so on), which also scales the bandwidth of the memory system and L4 capacity. To achieve statistically significant results, we introduce small amounts of non-determinism as proposed by Alameldeen and Wood [6], and perform enough runs to achieve 95% confidence intervals ≤1%.

5.2 Comparison Against Atomic Operations

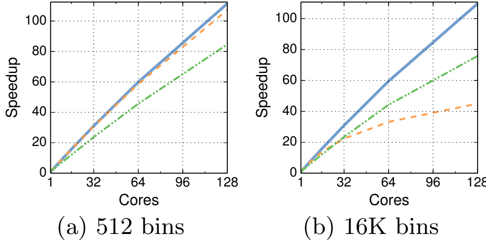
Fig. 10 compares the performance and scalability of Coup and a conventional MESI protocol. Each graph shows results for a single application, and each line in the graph shows how performance scales for a particular scheme (MESI or Coup) as the number of cores grows from 1 to 128 (x-axis). All speedup numbers are relative to the run-time of the application on a single core under MESI. Higher numbers are better.

Fig. 10 shows that Coup always outperforms MESI, often substantially. At 128 cores, Coup outperforms MESI by 2.4× on `hist`, 34% on `spmv`, 2.4× on `pgrank`, 20% on `bfs`, and 4.0% on `fluidanimate`. Moreover, the gap between MESI and Coup often widens as the number of cores grows, showing that Coup has better scalability than MESI.

Coup is especially beneficial for applications where shared data goes through long update-only phases. This is the case with `hist`, `spmv`, and `pgrank`. In `bfs`, where cache lines are constantly moving between U and S states as cores update and check the visited bit-vector (Sec. 4.2), Coup's advantage is lower but still significant. Finally, shared cells in `fluidanimate` experience long read-only and update-only phases, but only a fraction of cells are shared, and shared cells see few updates from neighboring threads on each update-only phase, so Coup provides a small speedup over MESI.

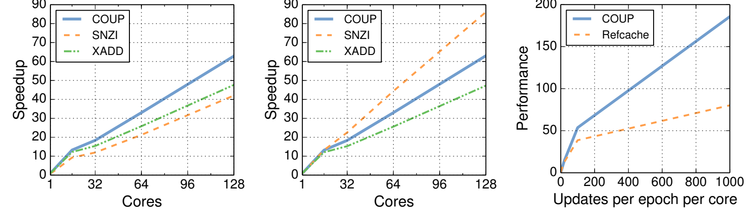
Fig. 11 gives more insight into these results by showing the breakdown of average memory access latency (AMAT). Each graph shows results for a single application. Each set of two bars shows results for Coup and MESI for a given system size (8, 32, or 128 cores). The height of each bar is the average memory access latency of all loads, stores, and instruction fetches issued from the L1s, normalized to the AMAT that Coup achieves at 8 cores. Each bar is broken down into time spent at the L2, L3, off-chip network, L4, coherence invalidations from the L4, and main memory. This breakdown shows critical-path delays only (e.g., the time spent on invalidations is not the time spent on every invalidation, but the critical-path delay that L4 requests suffer because other sharers need to be invalidated or downgraded).

Fig. 11 shows that Coup substantially reduces AMAT over MESI. At 128 cores, Coup's AMAT is lower than MESI's by 12.6× on `hist`, 10% on `spmv`, 3.0× on `pgrank`, 24% on `bfs`, and 12% on `fluidanimate`. Coup mainly does this by reducing invalidations and serialization. The effect of this reduction on the overall AMAT depends on how the application uses the memory system. For instance, Coup nearly eliminates invalidation traffic in `hist`, `spmv`, and `pgrank`. In `hist` and `pgrank`, invalidations are the dominant contributor to AMAT, so eliminating them has the largest impact. But AMAT in `spmv` is dominated by L4 and main memory accesses, so the overall impact of eliminating invalidations is smaller.



(a) 512 bins (b) 16K bins

Figure 12: Speedups of `hist` with Coup and both core- and socket-level privatization, using small (512) and large (16 K) numbers of bins.



(a) Immediate dealloc, low count (b) Immediate dealloc, high count (c) Delayed dealloc

Figure 13: Performance of Coup on reference counting microbenchmarks: (a, b) immediate deallocation and (c) delayed deallocation.

Beyond reducing AMAT, Coup also lowers traffic: at 128 cores, Coup incurs lower off-chip traffic than MESI by a factor of 20.2× on `hist`, 18% on `spmv`, 4.9× on `pgrank`, 20% on `bfs`, and 18% on `fluidanimate`.

Finally, even though Coup's benefits are significant, these benchmarks execute a relatively small fraction of commutative-update instructions: at 128 cores, commutative-update instructions are 1.0% of all executed instructions on `hist`, 2.4% on `spmv`, 4.9% on `pgrank`, 0.40% on `bfs`, and 0.96% on `fluidanimate`. Their impact is significant because, at large core counts, each atomic read-modify-write to a contended memory location can take several hundred cycles.

5.3 Case Study: Reduction Variables

All baseline benchmarks use atomic operations instead of privatization. To compare Coup with software privatization, we modify `hist` to make the histogram a reduction variable, and vary the number of bins (elements) in the histogram. We evaluate both core-level privatization, where each thread has its own variable, and socket-level privatization, where each socket has its own variable, shared and updated by all threads running in that socket using atomic operations. Socket-level privatization seeks to balance the overheads of the fully-shared and fully-privatized implementations.

Fig. 12 compares the performance and scalability of Coup with core-level and socket-level privatization on `hist`. Fig. 12a shows that, with a small number of bins, Coup outperforms core-level privatization by 3% and socket-level privatization by 38%. Core-level privatization works well in this case because each thread performs many updates to each histogram bin (128 on average), so reduction overheads are highly amortized.

In contrast, Fig. 12b shows that, with a large number of bins, Coup outperforms core-level privatization by 2.5× and socket-level privatization by 51%. In this case, core-level privatization is dominated by the cost of reductions, as each thread performs a small number of updates to each histogram bin (2 on average).

Finally, privatization also increases footprint and adds pressure to shared caches. If we grow both the number of bins and the image size (so the number of updates per bin and thread, and thus reduction overheads, stay constant), we see an additional performance degradation of 9% in the core-level privatized version when the aggregate size of all privatized histograms overflow the L3 caches, while Coup does not suffer this degradation.

5.4 Case Study: Reference Counting

We use two microbenchmarks to compare Coup's performance on reference counting against the software techniques described in Sec. 4.2. The first microbenchmark models immediate-deallocation schemes, and we use it to compare against a conventional atomic-based implementation and SNZI [26]. The second microbenchmark models delayed-deallocation schemes, and we use it to compare against Refcache [17].

Immediate deallocation: In this microbenchmark, each thread performs a fixed number of increment, decrement, and read operations over a fixed number of shared reference counters. We use 1 to 128 threads, 1 million updates per thread, and 1024 shared counters. On each iteration, a thread selects a random counter and performs either an increment or a decrement and read.

SNZI uses binary trees with as many leaves as threads. The performance of SNZI depends on the number of references per object—a higher number of references causes higher surpluses in leaves and intermediate nodes, and less contention on updates. To capture this effect, we run two variants of this benchmark. In the first variant (low count), each thread keeps only 0 or 1 references per object, while in the second mode (high count), each thread keeps up to five references per object.

To achieve this, in low-count mode, when a thread randomly selects an object, it will always increment its counter if it holds no references to that object, and it will always decrement its counter if it holds one reference. In high-count mode, threads will increment with probability 1.0, 0.7, 0.5, 0.5, 0.3, and 0.0 if they hold 0, 1, 2, 3, 4, and 5 local references to that counter, respectively.

For updates, Coup and XADD use commutative-add and atomic fetch-and-add instructions, respectively.

Fig. 13a and Fig. 13b show the results for these experiments. In the low-count variant (Fig. 13a), SNZI incurs high overhead when counts drop to zero, so both Coup and XADD outperform SNZI (by 50% and 17% at 128 cores, respectively). By contrast, in the high-count variant (Fig. 13b), SNZI enjoys lower contention and outperforms Coup (by 35% at 128 cores). Coup

outperforms XADD in both cases.

We conclude that, in high-contention scenarios, COUP provides the highest performance, but in specific scenarios, software optimizations that exploit application-specific knowledge to avoid contention among reads and updates can outperform COUP. We also note that it may be possible to modify SNZI to take advantage of COUP and combine the advantages of both techniques.

Delayed deallocation: In the delayed-deallocation microbenchmark, 128 threads perform increments and decrements (but not reads) on 100,000 counters. We divide the benchmark into epochs, each with a given number of updates per thread. When they finish an epoch, threads check whether counters are zero, simulating delayed-deallocation periods as in Refcache [17].

Our COUP implementation updates counters with commutative-add instructions and maintains a bitmap with a "modified" bit for each counter. The bitmap is updated with commutative-or instructions. Between epochs, cores use ordinary loads to read the value of marked counters and check whether the counters are zero. Refcache uses a per-thread software cache (a hash table) to maintain the deltas to each modified counter. Threads flush this cache when they finish each epoch.

Fig. 13c shows the performance COUP and Refcache on the delayed deallocation microbenchmark as the number of updates per epoch (x-axis) grows from 1 to 1000 updates per thread and epoch. COUP outperforms refcache across the range, by up to 2.3×.

We conclude that COUP primarily helps delayed-deallocation reference counting by allowing a simpler, lower-overhead implementation to capture the low communication costs of prior software approaches (in this case, using counters and bitmaps instead of hash tables).

5.5 Sensitivity to Reduction Unit Throughput

COUP is barely sensitive to reduction unit throughput. We compare the default 256-bit ALU, which has a throughput of one cache line per 2 cycles, with a simpler, unpipelined 64-bit ALU, which has a throughput of one line per 16 cycles. The maximum performance degradation incurred with the slower ALU is 0.88% at 128 cores on `bfs`. Smaller systems incur somewhat lower worst-case degradations (e.g., 0.76% at 64 cores).

6. ADDITIONAL RELATED WORK

Loosely consistent memory (LCM) [41] is a software-controlled coherence protocol built on top of Tempest [52] that allows multiple caches to hold writable copies of the same line. These copies can become incoherent, and software must explicitly reconcile them in a later merge phase. Unlike LCM, COUP preserves cache coherence and transparently merges partial updates, requiring no software intervention.

Several cache-coherence optimizations reduce the cost of updates, though that is not their primary purpose: self-invalidations, done with either hardware predictors [43] or software protocols [16, 33], remove invalidations from the critical path; adaptive-granularity coherence schemes [38, 67, 71] reduce both false sharing and the amount of dirty data sent on invalidations; and speculation and fast networks can reduce the cost of atomic operations [27]. These schemes are orthogonal to COUP, which could be used in conjunction with them to improve performance.

While we have focused on shared-memory systems, reductions are also common with message passing. The BlueGene/L and BlueGene/Q supercomputers feature specialized collective networks that perform these reductions completely in hardware, using ALUs embedded in network routers [7, 14]. In contrast to COUP, their main advantage is minimizing the latency of scalar or short reductions across a very large number of nodes.

7. CONCLUSION

We have presented COUP, a technique that exploits commutativity to reduce the cost of updates in cache-coherent systems. COUP extends conventional coherence protocols to allow multiple caches to simultaneously hold update-only permission to data. We have introduced an implementation of COUP that uses this support to accelerate single-word commutative updates. This implementation requires minor hardware changes and, in return, substantially improves the performance of update-heavy applications. Beyond this specific implementation, a key contribution of our work is to recognize that it is possible to allow multiple concurrent updates without sacrificing cache coherence or relaxing the consistency model. Thus, COUP attains performance gains without complicating parallel programming. Finally, COUP can apply to other contexts. For example, with limited programmability in the cache controller, it may be possible to support multi-word commutative updates, such as insertions and removals from sets and other unordered data structures. We leave this and other applications of COUP to future work.

8. ACKNOWLEDGMENTS

We thank Nathan Beckmann, Christina Delimitrou, Joel Emer, Mark Jeffrey, Harshad Kasture, Anurag Mukkara, Suvinay Subramanian, Po-An Tsai, and the anonymous reviewers for their helpful feedback. This work was supported in part by C-FAR, one of six SRC STARnet centers by MARCO and DARPA, and by NSF grant CAREER-1452994. Guowei Zhang was partially supported by a MIT EECS Grier Presidential Fellowship.

9. REFERENCES

[1] GReat Images in NASA (GRiN), http://grin.hq.nasa.gov.

[2] "HSA Platform System Architecture Specification," HSA Foundation, Tech. Rep., 2015.

[3] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, 1996.

[4] V. Agarwal *et al.*, "Scalable graph exploration on multicore processors," in *SC10*, 2010.

[5] J. H. Ahn, M. Erez, and W. Dally, "Scatter-add in data parallel architectures," in *HPCA-11*, 2005.

[6] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[7] G. Almási *et al.*, "Optimization of MPI collective communication on BlueGene/L systems," in *ICS'05*, 2005.

[8] P. Bailis *et al.*, "Coordination avoidance in database systems," *VLDB*, vol. 8, no. 3, 2014.

[9] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT-17*, 2008.

[10] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library.* O'Reilly, 2008.

[11] I. Calciu *et al.*, "Message passing or shared memory: Evaluating the delegation abstraction for multicores," in *PODC*, 2013.

[12] I. Calciu, J. Gottschlich, and M. Herlihy, "Using elimination and delegation to implement a scalable NUMA-friendly stack," in *HotPar*, 2013.

[13] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS directories: A scalable cache coherence scheme," in *ASPLOS-IV*, 1991.

[14] D. Chen *et al.*, "The IBM Blue Gene/Q interconnection network and message unit," in *SC*, 2011.

[15] J. Chhugani *et al.*, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *IPDPS*, 2012.

[16] B. Choi *et al.*, "Denovo: Rethinking the memory hierarchy for disciplined parallelism," in *PACT-20*, 2011.

[17] A. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *EuroSys*, 2013.

[18] A. Clements *et al.*, "The scalable commutativity rule: Designing scalable software for multicore processors," in *SOSP-24*, 2013.

[19] F. J. Corbato, "A Paging Experiment with the Multics System," in *MIT Project MAC Report MAC-M-384*, 1968.

[20] D. Culler, J. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach.* Morgan Kaufmann, 1999.

[21] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM TOMS*, vol. 38, no. 1, 2011.

[22] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI-6*, 2004.

[23] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *ARITH*, 2013.

[24] D. Dill *et al.*, "Protocol verification as a hardware design aid," in *ICCD*, 1992.

[25] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *IWOMP-4*, 2008.

[26] F. Ellen *et al.*, "SNZI: Scalable nonzero indicators," in *PODC*, 2007.

[27] S. Franey and M. Lipasti, "Accelerating atomic operations on GPGPUs," in *NOCS-7*, 2013.

[28] M. Frigo *et al.*, "Reducers and other Cilk++ hyperobjects," in *SPAA*, 2009.

[29] A. Gottlieb *et al.*, "The NYU Ultracomputer: Designing a MIMD Shared Memory Parallel Computer," *IEEE Trans. Comput.*, vol. 100, no. 2, 1983.

[30] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *HiPEAC*, 2010.

[31] N. Johnson *et al.*, "Speculative separation for privatization and reductions," in *PLDI*, 2012.

[32] W. Jung, J. Park, and J. Lee, "Versatile and scalable parallel histogram construction," in *PACT-23*, 2014.

[33] J. Kelm *et al.*, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *ISCA-36*, 2009.

[34] R. Kessler and J. Schwarzmeier, "CRAY T3D: A new dimension for Cray Research," in *COMPCON*, 1993.

[35] F. Kjolstad and M. Snir, "Ghost cell pattern," in *Workshop on Parallel Programming Patterns*, 2010.

[36] K. Knowlton, "A fast storage allocator," *CACM*, no. 8, 1965.

[37] C. Koelbel, *HPF handbook.* MIT Press, 1994.

[38] S. Kumar *et al.*, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *MICRO-45*, 2012.

[39] G. Kurian, "Locality-aware Cache Hierarchy Management for Multicore Processors," Ph.D. dissertation, MIT, 2014.

[40] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC." in *OSDI-10*, 2012.

[41] J. Larus, B. Richards, and G. Viswanathan, "LCM: Memory system support for parallel language implementation," in *ASPLOS-VI*, 1994.

[42] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," in *ISCA-24*, 1997.

[43] A. Lebeck and D. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *ISCA-22*, 1995.

[44] C. Leiserson and T. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *SPAA*, 2010.

[45] M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE TPDS*, vol. 15, no. 6, 2004.

[46] N. Narula *et al.*, "Phase reconciliation for contended in-memory transactions," in *OSDI-11*, 2014.

[47] L. Page *et al.*, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[48] M. Papamarcos and J. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ISCA-11*, 1984.

[49] L. Rauchwerger and D. Padua, "The privatizing doall test: A run-time technique for doall loop identification and array privatization," in *ICS'94*, 1994.

[50] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE TPDS*, vol. 10, no. 2, 1999.

[51] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[52] S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: User-level shared memory," in *ISCA-21*, 1994.

[53] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *HPCA-18*, 2012.

[54] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[55] D. Sanchez, R. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *ASPLOS-XV*, 2010.

[56] N. Satish *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *SIGMOD*, 2014.

[57] S. Scott, "Synchronization and communication in the T3E multiprocessor," in *ASPLOS-VII*, 1996.

[58] P. Sewell *et al.*, "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, 2010.

[59] D. Sorin, M. Hill, and D. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, 2011.

[60] O. Villa *et al.*, "Effects of floating-point non-associativity on numerical computations on massively multithreaded systems," *Cray User Group*, 2009.

[61] T. Von Eicken *et al.*, "Active messages: a mechanism for integrated communication and computation," in *ISCA-19*, 1992.

[62] J. Warnock *et al.*, "22nm next-generation IBM System z microprocessor," in *ISSCC*, 2015.

[63] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[64] H. Wong *et al.*, "Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor," in *PACT-17*, 2008.

[65] H. Yu and L. Rauchwerger, "Adaptive reduction parallelization techniques," in *ICS'00*, 2000.

[66] J. Zebchuk *et al.*, "A tagless coherence directory," in *MICRO-42*, 2009.

[67] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *MICRO-40*, 2007.

[68] L. Zhang, Z. Fang, and J. Carter, "Highly efficient synchronization based on active memory operations," in *IPDPS*, 2004.

[69] M. Zhang *et al.*, "PVCoherence: Designing flat coherence protocols for scalable verification," in *HPCA-20*, 2014.

[70] M. Zhang, A. Lebeck, and D. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *MICRO-43*, 2010.

[71] H. Zhao *et al.*, "Protozoa: Adaptive granularity cache coherence," in *ISCA-40*, 2013.