Check for updates

Received: 11 May 2020 | Revised: 10 April 2021 | Accepted: 18 April 2021

DOI: 10.1002/cpe.6414

SPECIAL ISSUE PAPER

WILEY

Sharing non-cache-coherent memory with bounded incoherence

Yuxin Ren[1] | Gabriel Parmer[1] | Dejan Milojicic[2]

[1]The George Washington University, Washington, District of Columbia, USA

[2]Hewlett Packard Enterprise, Palo Alto, California, USA

Correspondence
Gabriel Parmer, The George Washington University, Washington, DC, USA.
Email: gparmer@gwu.edu

Summary

Cache coherence in modern computer architectures enables easier programming by sharing data across multiple processors. Unfortunately, it can also limit scalability due to cache coherency traffic initiated by competing memory accesses. Rack-scale systems introduce shared memory across a whole rack, but without inter-node cache coherence. This poses memory management and concurrency control challenges for applications that must explicitly manage cache-lines. To fully utilize rack-scale systems for low-latency and scalable computation, applications need to maintain cached memory accesses in spite of non-coherency. This paper introduces Bounded Incoherence, a memory consistency model that enables cached access to shared data-structures in non-cache-coherency memory. It ensures that updates to memory on one node are visible within at most a bounded amount of time on all other nodes. We evaluate this memory model on modified `PowerGraph` graph processing framework, and boost its performance by 30% with eight sockets by enabling cached-access to data-structures.

KEYWORDS

non-cache-coherent shared memory, rack-scale architectures, scalability

1 | INTRODUCTION

Recently, rack-scale systems have been gaining momentum. These include FireBox[1] from Berkeley, Rack-scale Architecture[2] from Intel, and The Machine from Hewlett Packard Enterprise.[3] These instantiations are comprised of tens of thousands of cores and petabytes of persistent byte-addressable memory. This pool of memory is accessible from any node in the system over fast photonic interconnects that enable load-store accesses at close to DRAM speeds. These systems promise to enable memory-centric computing in which many nodes that would traditionally be implemented as a distributed system can communicate and coordinate directly through memory. However, due to the scale of the system, there is no cache coherency support between the nodes, instead only among the cores on a single node. This complicates the use of the shared memory pool for inter-node collaboration via shared data-structures, as explicit software support is required to achieve synchronization and coherency among different nodes accessing memory. To meet the promise of the massive pool of shared memory for low-latency, high-throughput processing, new techniques to handle non-Cache Coherent (non-CC) memory are required in rack-scale systems.

To coordinate incoherent memory across nodes, we introduce a consistency model based around *Bounded Incoherence* (BI) for rack-scale architectures. This system enables multiple nodes that share only non-CC memory to have many of the benefits of typical multi-core shared memory processors. In non-CC architectures, BI enables controlled access to cache-lines that are incoherent with changes made by other nodes for at most a bounded window of time. Thus, lookups and loads in shared data-structures use efficient, cached access. BI trades time-to-consistency for this efficient, cache-based local access to data-structures. BI makes the observation that *access to stale cache-lines can be tracked similarly to the parallel references that are implicitly tracked by Scalable Memory Reclamation* (SMR) techniques.[4-8] The SMR in the BI runtime is based on logical clocks and efficient cache-line invalidation that together provide bounds on the staleness of cache contents. In a nutshell, BI tracks references to data-structures, invalidates possibly stale cache-lines and delays memory reuse.

 © 2021 John Wiley & Sons, Ltd.

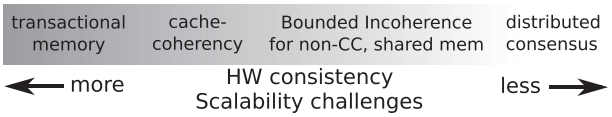


FIGURE 1 Varying hardware support for consistency

In many ways, the goal is to maintain the convenience of shared memory processing even across non-CC nodes. The lack of a cache-coherent memory fabric in rack-scale systems complicates the implementation of traditional shared-memory data-structures that span multiple nodes. Instead of using shared-memory, the hardware can be treated as a share-nothing distributed system, using message-passing-based distributed consensus to coordinate between nodes, often at the significant cost of network overheads compared to cache overheads. Figure 1 depicts a taxonomy of trade-offs between the hardware support for consistency via cache coherency and the scalability of distributed systems. Most prior research focuses on the right end of the spectrum,[9-11] leaving alternatives in the design space unexplored. As pointed out by Harris,[12] solely relying on message passing does not suit some workloads, and what programming models are appropriate for combining message passing with shared memory is still an open research question. BI makes an effort to answer this question, as it enables cached access to shared memory on the read-path, while it exploits message passing to ease synchronization on update-path.

We first have a general discussion about the challenge and overhead when sharing data-structures on top of incoherence caches in §2. We then introduce BI memory model and its impact on the semantics of data-structures in §3. After discussing the design and implementation of BI in §4 and §5, we apply BI to a modified `PowerGraph` (§6) and compare against distributed implementations. The evaluation in §7 validates BI's ability to maintain common case performance comparable to cache-coherent multi-processors for contemporary hardware, and to enable an infrastructure for managing the non-coherent memory. The contributions of this paper include:

1. the design and implementation of BI as a means for reasoning about shared memory in non-CC architectures,
2. the application of BI to the `PowerGraph` system, and to a RCU-driven balanced tree data-structure, and
3. the evaluation of BI versus distributed approaches to cache coherency management.

2 | MOTIVATION AND BACKGROUND

This section first discusses the general challenge imposed by incoherent cache when sharing data-structures (§2.1), and then investigates the overhead of typical cache operations (§2.2). §2.3 briefly introduces Read-Copy-Update[7] (RCU) and Scalable Memory Reclamation[5,6,8] (SMR), which inspires the implementation of BI.

2.1 | Challenges with cache-incoherence

To investigate the impact of incoherent memory accesses on the consistency properties of a data-structure, Figure 2 depicts a simple linked list data-structure undergoing modification with non-CC memory. We use the word "node" to refer to the computational elements in a coherency domain, and "objects" to reference the constituent allocations within a data-structure. A number of different data-structure inconsistencies arise due to the non-CC memory.

1. *Stale access:* in (b), the reference from the first to the second object can still remain in node's caches, despite the new object being linked after the first.
2. *Resolution failure:* in (b), the new object is not visible on nodes with stale cache-lines, thus attempting to resolve that object (i.e., do a lookup on it) fails.
3. *Dangling references:* in (c), stale cache-lines still reference a `removed` and `freed` object.
4. *Type inconsistency:* additionally, in (d), the previously removed node is reallocated as a different type of memory and used in another data-structure. The *type and size of the resource* can be different if accessed from a stale cache-line, and, even if the type is the same, the context is different.

Dangling references and type inconsistencies have a direct analogy in non-blocking data-structures concerning the ABA problem.[13] In the absence of a mutually exclusive abstraction (e.g., locks) over all data-structure modifications and access, the ABA problem stems from races between object access, and memory management operations on that object. An object can be both accessed by one node, and concurrently `freed` then `malloced` as an object of a *different* type on another node. Thus, non-blocking algorithms are often paired with SMR techniques[4,6,7,14] to avoid re-allocating objects until there are no parallel threads possibly accessing them.

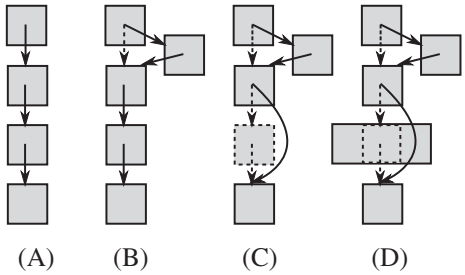
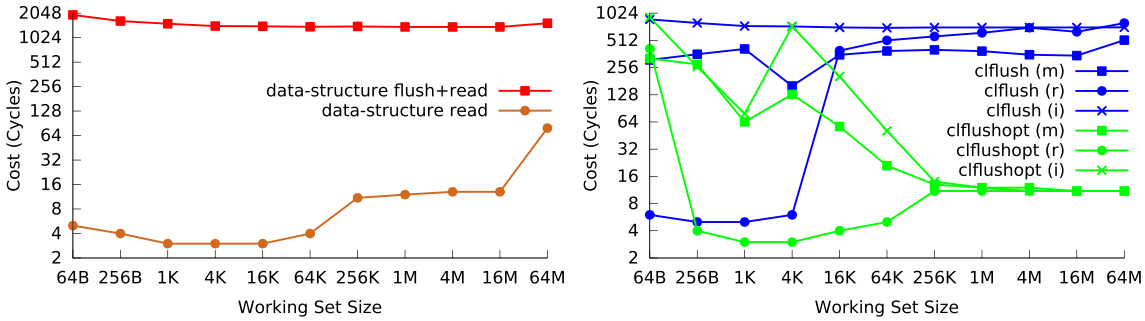


FIGURE 2 A singly-linked list going through a sequence of modifications with non-CC memory. Each list is a configuration after a modification. Dashed lines and boxes represent stale cache contents for that object, while dark continuous lines denote the state in memory. (A) is the initial configuration; (B) adds an object, leaving a stale link in some node's caches; (C) `removes` and `frees` the second to last object, but it remains in stale cache-lines on other nodes; and (D) the freed object is allocated as a different type (shape)



(A) Cache flush overhead on memory access.

(B) Cost of different cache-line states and instructions.

FIGURE 3 Cache operation per-cache-line overheads

Just as lock-free algorithms often use SMR to handle stale parallel access to read-mostly data-structures, comparable techniques can be used to handle stale cache references to shared data-structures between non-coherent nodes. While SMR prevents the re-use of memory while a parallel thread *can be accessing* the object, BI is designed to prevent the re-use of memory while any node *can have the object in its cache*. This is a key observation of this paper, and is (to the best of our knowledge) the first instance of SMR applied to non-coherence systems.

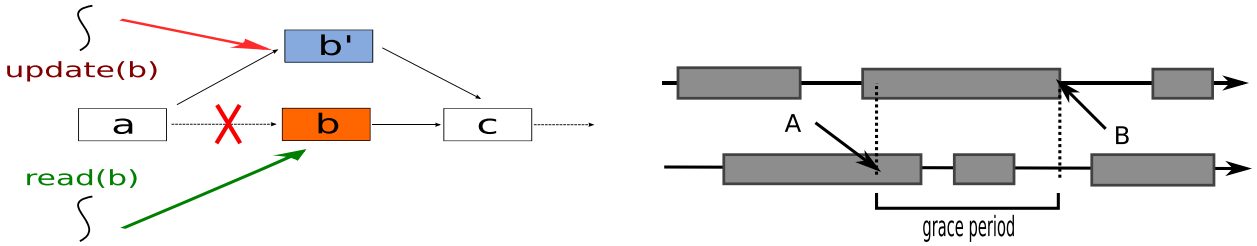
2.2 | Cache operation overheads

There are three typical cache operations for cache coherence management: (1) invalidate, which marks a cache line as invalid. (2) write-back, which only writes a dirty cache line back to memory, leaving the cache still valid, and (3) flush, which combines invalidate and write-back. To better understand the interactions between data-structure accesses, and cache operations in non-CC memory, Figure 3 reports the per-cache-line overhead for a number of different memory and cache operations. Results are from the HPE Superdome Flex server (more hardware details can be found in §7).

Overhead on memory access. When programming for non-CC memory, one option is to add cache instructions to flush and write-back cache-lines either explicitly, or through compiler extensions. However, the performance impact of these instructions and subsequent accesses to the cache-lines are significant, as they operate at memory-speed. The `data-structure read` and `flush+read` lines in Figure 3A represent random-accesses to cache-lines of a given working set. The working set is the total size of accessed memory, and should be evaluated relative to cache sizes on this platform – a 32KB L1cache, 1MB L2 cache, and 38.5MB L3 cache. A `clflush` will writeback and/or invalidate the cache-line in all levels of cache (even in non-shared caches such as the L2 of another core), and each cache-line is 64 bytes. This represents the cost of accessing a simple data-structure using only reads – mainly a function of the level of cache in which the load resolves, versus pre-pending each of those reads with a cache-line invalidation to ensure that the read accesses the most up-to-date value.

Conclusion #1: frequently executed data-structure operations should avoid cache operations. Requiring explicit flush operations on each access of shared memory data-structures has significant overhead. This motivates BI to enable cache-speed access to shared data-structures.

Cost of different cache-line states and instructions. In Figure 3B, the lines for `clflush` demonstrate the cost of flushing a random sequence of cache-lines after either reading the cache-line into cache (r), modifying it (m), or invalidating it (i). This shows the cost of flushing cache-lines in different states in the cache. `clflush` is a serializing instruction which effectively awaits previous `clfush`'s completion before completing the next.. Alternatively, the `clflushopt` instruction, available on modern x86 processors, enables micro-architectural pipelining and reordering of the flush instructions (to different cache-lines). To await the completion of all `clflushopt`instructions, a single memory barrier (via `mfence`) is executed. For a small numbers of flushes, the cost of this barrier dominates, but for larger numbers of flushes, its impact is marginalized. Manipulating invalidated cache-lines has the largest cost. Because the current core is not able to ascertain other cores' cache-line status, it has to broadcast the invalidation request to its cache coherency domain and waits for the responses from all the other cores. Due to cache write-backs, there is more



(A) RCU update operation. Writers perform modifications on a separate private copy, and then atomically unlink the old object while linking in the new one.

(B) Time advances to the right for two readers, and the bars represent read operations. At time A, data is freed and made unreachable, but remains referenced till B by the second reader. At that point, it can be reclaimed.

FIGURE 4 Read copy update and scalable memory reclamation

overhead for cache-lines that are modified, than if they are only read. The decreased pipeline serialization seen in `clflushopt` decreases the cost of flushing cache-lines for large buffers. These results show that, for this processor, it is beneficial to both leverage architectural support for pipelined flushes, and to batch those flushes to the largest extent possible.

Conclusion #2: systems should batch cache-line flush operations and use non-serializing instructions. The integration of `clflushopt`, and its use on large ranges of non-modified memory, minimizes cache operation overheads.

2.3 | Read copy update and scalable memory reclamation

RCU maximizes the performance of data-structure readers as it requires no explicit synchronization on the read path. However, data-structure writers have to perform more complex and slower update operations to avoid interference on concurrent readers. Figure 4A illustrates an RCU-style update operation in a shared linked list. A writer finds the object to be updated (`b`), and copies it into a newly allocated structure (`b'`). After modifying that object with the intended changes, the writer atomically unlinks the old object while linking in the new one. The unlinked object introduces a challenge of SMR, which reclaims freed memory at a safe point when no reader can potentially hold a reference to the old data.

SMR implementations fundamentally exist to ascertain if a freed object that has been disconnected from a data-structure is still possibly accessed by a parallel thread. Instead of tracking individual references to specific objects within a data-structure, SMR techniques track when data-structure references could exist inside each reader. *Quiescence* of a particular reader is achieved when it no longer holds references to freed objects. A *grace-period*[7] is a period of time during which every reader goes through at least one quiescent state. The core principle of SMR is that a freed object can be safely reclaimed after a grace period. Figure 4B gives an example of grace-period calculation for two parallel readers. All existing SMR solutions rely on a fundamental assumption: a reader will no longer access freed objects after exiting an RCU-protected read operation. However, incoherent cache breaks such assumption, and this paper presents how BI fills this gap.

3 | BI MEMORY MODEL

This paper introduces the Bounded Incoherence (BI) memory consistency model that enables efficient, cache-based access for shared data-structures on non-cache-coherent architectures.

3.1 | BI consistency model

To provide the benefits of cached read-path access, and to avoid many of the consistency problems from §2.1, BI is designed to have a number of properties:

- **P1** Cached object access is used for all read-paths, thus eliding expensive cache-line invalidations and subsequent memory accesses.
- **P2** Cache-lines are stale only for at most a bounded amount of time.
- **P3** Similar to RCU quiescence that tracks when no possible references to freed data-structure objects can exist and memory can be reclaimed, BI tracks when there are no possible references in caches to freed data-structure objects. BI prevents those objects from being reused while their possibly stale cache-lines exist in any cache. This avoids dangling references and type inconsistencies, but also delays the reuse of memory thus increasing memory requirements.
- **P4** As accesses to stale data-structure cache-lines are allowed, *modifications* to that data-structure must be atomic with respect to reads.

Given these properties, BI is a memory consistency model with specific visibility constraints between loads and stores on different nodes. For example, sequential consistency[15] ensures that loads and stores on a specific node are visible in the same order, and not reordered with respect to loads and stores on another node. In contrast, BI is a relaxed consistency model in that it admits looser orderings between loads and stores across nodes:

1. *Stores* of one node are visible to other node's loads in at most a *bounded amount of time*. Due to cached-access to potentially stale object cache lines, loads don't immediately observe another node's store.
2. *Stores* are made directly to memory and those made to a single address are seen in a sequential order.
3. *Stores* to different addresses can be reordered on another node. Stores to addresses a_0, then a_1 might be seen by another socket as modifying memory at a_1 first, then a_0. This happens if a load from a_0 hits in the cache (finding cached, stale data), then a_1 is loaded, missing in cache, thus seeing the store's updated value. After a_0's cache-line is evicted from the cache (from capacity, contention, or explicit eviction), a further load will miss in cache and find the updated value. As such, a socket can see a_1's store before a_0's, reordering stores.

3.2 | Data-structure semantics for BI

To understand which data-structures can benefit from BI, we discuss the interplay between the semantics of data-structures and BI properties. Here we'll separately consider data-structure lookups (read-paths) and modifications (update-paths).

Data-structure lookups with BI. Lookups exclusively use loads on objects. Modifications to these objects on other nodes do not have guaranteed immediate visibility (**P1**). Thus it is necessary that even stale versions of objects result in fault-free lookups. Different data-structure semantics admit trade-offs here.

Tolerable delayed freshness on lookups. The most lenient data-structure consistency requirements allow lookups to access stale objects. This has the potential to violate causality. For example, a hash-table shared between nodes can have a `put` operation add a key/value to the data-structure, while a `get`serviced by another node will only be guaranteed to return that key/value after a bounded latency. This might be acceptable if the hash-table is simply a cache for web objects, and failure to find a key after it's addition is compensated by logic to retrieve data from a backing data-base.

Lazy invalidation on lookup resolution failure. As part of BI, we investigate another option that has stronger consistency properties between additions and subsequent accesses. If a lookup fails to find the object it is looking for, then the lookup is retried while invalidating all cache-lines along the lookup path before loading them. This enables *additions* to the data-structure to be immediately visible to parallel lookups. We call this *lazily invalidating* cache-lines. Using the same hash-table example, additions of keys on one node are immediately visible on another. However, *modification* and *removal* of *existing* keys will be visible after at most a bounded amount of time (**P2**).

Data-structure modifications with BI. When multiple nodes can modify the same objects in a data-structure, they require consistency with concurrent lookups. In linked data-structures, (e.g., a simple linked list), an object might be added after an existing object while a parallel modification has already removed the existing object from the list. This has the effect of adding a node without actually making it visible within the structure. Synchronizing between concurrent modifications is generally a difficult problem, even for parallel systems using SMR.[16-18] To cope with such difficulty, BI supports two mechanisms (**P4**):

1. *Mutually exclusive writers.* Mutual exclusion alone is not sufficient, and must be paired with explicit cache-line write-back and invalidation operations.
2. *Partitioned writers.*

   To avoid the overhead of flushing possibly stale cache-lines in objects to be modified, data-structures can be *partitioned* across nodes, thus avoiding synchronization of cache-lines between writers. Given the partitioning of the data-structure modifications to specific nodes, message passing is required to steer the modification request.

Summary. BI *trades time-to-coherency for increased locality of data access* and the ability to avoid explicit cache operations on read-paths. It complicates update-paths as they must be atomic with respect to parallel lookups and other modifications (**P4**). This way we are optimizing the common case and moving complexity to less frequent case. These limitations are similar, but more restrictive than those around non-blocking data-structures that use SMR techniques. BI is similar to RCU in that it uses quiescence as a fundamental mechanism. However, BI is the first system to associate quiescence with data-structure coherency on a non-coherent system. This requires new mechanisms to provide quiescence on non-coherent systems (**P3**). The wide-spread use of RCU in the Linux kernel[7,16] demonstrates that there are data-structures with relaxed consistency requirements. An important question is if the additional constraints of BI including delayed visibility for modifications, and modification synchronization using either expensive mutual exclusion, or partitioning, prohibit interesting applications.

4 | BI DESIGN

This paper focuses on creating abstractions to handle shared data-structures on non-CC memory. The primary goals of BI are (1) allow common-case read-only operations to proceed without any synchronization nor cache operations; and (2) guarantee consistency between concurrent readers and writers.

BI focuses on relatively general applicability and adheres to the classic RCU API (§4.1), which is used broadly across the Linux kernel, and in applications via a user-level library. BI extends RCU with a set of enhancements to both RCU readers and writers. On the reader side, BI provides cache invalidation focusing on batching cache-line invalidation operations (motivated by Figure 3B), and integrates this cache invalidation into the quiescence mechanism (§4.2). On the other hand, a BI writer is responsible for (1) stale cache tracking, which coordinates with BI readers for cache invalidation; and (2) memory management which provides safe memory reclamation (§4.3).

4.1 | BI API

The BI API mainly inherits from RCU, but extends it to explicitly manage cache coherence. On the reader side, BI use the same API as RCU to access shared data-structures.

1. `bi_enter()` declares the start of a code section in which references to objects can exist. Its usage is the same as `rcu_read_lock()` in RCU.
2. `bi_exit()` declares the end of that section, same as the RCU counterpart, `rcu_read_unlock()`. No thread-local references to objects can remain after this.
3. `bi_dereference(void **)` fetches a shared pointer, referenced by a shared pointer, which can be safely dereferenced.

BI introduces two additional APIs for readers to achieve cache coherence.

1. `bi_stale_object_quiescent(callback_fn)` invalidates stale cache lines to get their updated value. For instance, in Figure 4A, readers use this to invalidate the cache containing object `a`. `callback_fn` is discussed in §4.3.
2. `bi_free_object_quiescent()` flushes the local cache to drop any references to freed objects. As an example, in Figure 4A, readers call this to flush object `b`'s cache lines.

§5 details how BI invokes the above APIs to achieve cache quiescence inside each reader.
On the writer side, BI needs more APIs to manage cache and coordinate concurrent readers.

1. `bi_assign_pointer(void **, value)` assigns a new value to a shared pointer. It also writes back the new value to memory, and records the modified object if BI modification tracking is enabled.
2. `synchronize_bi()` detects an elapsed grace period. It differs from `synchronize_rcu()` in two ways. First, it does not block waiting for quiescence in the future. Instead it calculates the most recent time in the past when quiescence was achieved. Second, in addition to checking if readers exit the read-side section, it also checks if necessary cache flushes are performed.

To integrate memory and cache coherency management, BI provides an extra set of memory operations.

1. `bi_alloc(size, flag)` allocates memory for data-structure objects. §4.3 discusses `flag`.
2. `bi_free(void *)` deallocates the object without actually yet freeing up its memory.
3. `bi_reclaim()` reclaims and frees previously deallocated objects. It uses `synchronize_bi()` to make sure all reclaimed objects can be safely reused.

Figure 5 depicts example pseudocode illustrating the use of BI to operate a shared data-structure. A reader (writer) finds the object of interest and processes (modifies) it.

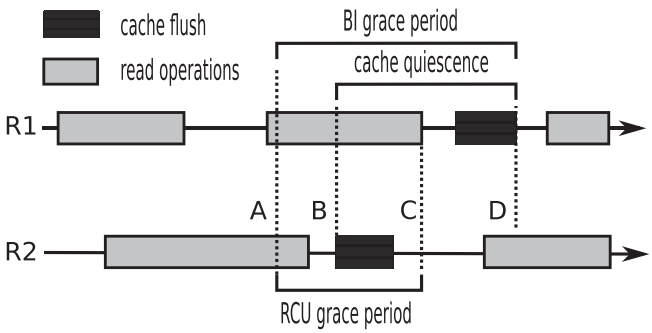
All such read operations are delimited by `bi_enter` and `bi_exit`. To perform a modification, the writer first allocates a new object (line 2), then performs the update by copying (a part of) the old object (line 5), updating the copy (line 6), and replacing the old version with the new one (line 7). The old object is marked to be freed later (line 8). The reclamation of freed objects (line 12) happens after quiescence detection (line 11), which contains the logic to ensure a grace period has elapsed. On the reader side, extra care needs to be taken to deal with incoherent caches. After `bi_exit` (line 19), while readers do not logically hold any references to freed object, references can still be contained in stale cache lines. Thus the reader has to flush the stale lines of freed objects (line 22). Furthermore, as the reference itself is modified by a writer, the reader needs to invalidate that cache

```
1  void writer(D, identifier) {
2      n = bi_alloc(size, flag);      // allocate a new object
3      bi_enter();
4      p = lookup(D, identifier);     // find existing object
5      copy(n, bi_dereference(p));    // copy p to n
6      modify(n);                     // update the contents
7      bi_assign_pointer(p, n);       // add n and remove p
8      bi_free(p);                    // add to quiescence queue
9      bi_exit();
10     ...
11     synchronize_bi();              // detect quiescence
12     bi_reclaim();                  // free after quiescence
13 }
```

```
14 void reader(D, identifier) {
15     bi_enter();
16     p = lookup(D, identifier);
17     e = bi_dereference(p);
18     process(e);
19     bi_exit();
20     ...
21     // drop reference to free objects (e)
22     bi_free_object_quiescent();
23     // drop stale cache of modified memory (p)
24     bi_stale_object_quiescent();
25 }
26
```

FIGURE 5 Typical usage of BI API with a shared data-structure D, in this case a lookup-based structure

FIGURE 6 Time advances to the right for two readers. AC is RCU grace period. BD is cache quiescence, and AD is BI global grace period. Hence, an object freed at A can be reclaimed after C in RCU, but can only be reclaimed after D with BI



line (line 24) as well in order to see the updated reference. Those cache invalidation can be delayed to a later point to batch more cache flushes, but doing so introduces staleness. Therefore, it is essential for BI to guarantee the staleness is sustained for at most a bounded amount of time.

4.2 | Cache quiescence

Non-coherent caches break existing RCU implementations as references to freed objects can still exist in a stale cache. As shown in Figure 6, with a coherent cache, R1 is unable to access a freed object after it completes a read operation (at time point C). However, without coherency, R1 can access freed objects via stale references. Thus we cannot achieve quiescence at C to reclaim freed memory. To drop these stale references, BI invalidates stale cache-lines on each reader node, and determines a time period at which *every* reader has done so. This time period is defined as the time at which BI achieves cache quiescence, thus ensuring that no stale cache lines persist after quiescence. Global quiescence is achieved by combining cache quiescence and existing time-based quiescence. Figure 6 compares RCU quiescence against BI quiescence. Any resources or objects deactivated *before* the last quiescence point may then be reused.

Achieving quiescence. As §2.2 suggests, cache invalidation overhead is minimized when batching cache-line flush operations and using non-serializing instructions. Therefore, BI delays cache invalidation to batch more stale cache lines, instead of executing it immediately after every read operation.

Invalidating stale cache-lines is performed in one of three ways: (1) invalidate all data-structure cache-lines, (2) invalidate all *accessed* cache-lines since last quiescing, or (3) invalidate all *modified* objects on all nodes. Each option entails trade-offs depending on the dimensions of data-structure size, working set, and fraction of data-structure objects modified, respectively. Invalidating the whole data-structure saves extra tracking overhead, but only works with small data-structures. When the working set is not large, invalidating only accessed cache lines efficiently avoids unnecessary flushs of unaccessed memory. Invalidating modified objects works best in the case of read-heavy workloads.

Quiescence policy. Currently BI provides built-in support for logging stale cache lines and invalidating them. The log is stored in a shared ring buffer, which is populated by writers when modifying objects (§4.3). Inside the cache quiescence routine, a reader first invalidates the shared log to get its latest contents. Then the reader iterates the log and flushes every logged object's cache lines. In the default setting, BI periodically invokes the cache quiescence routine for each reader in the background. This period forms the upper bound of cache quiescence as well as cache staleness. The frequency of cache quiescence on each reader represents a system trade-off between time-to-consistency and cache operation overheads. High frequencies will incur more overhead due to the activation of the thread to perform quiescence, and due to more frequent cache misses for data-structures, but will achieve quiescence at a finer granularity, thus enabling the reuse of data-structure objects sooner. Conversely, low

frequencies decrease the cache quiescence thread's activation overheads, and more effectively batch cache flush operations, but provide a coarser granularity of quiescence. Determining the best frequency is beyond the scope for this work.

Despite this periodic cache quiescence, BI also provides on-demand cache quiescence to allow applications to employ their own specific object tracking and cache quiescence policy. Applications explicitly call cache quiescence APIs (e.g., `bi_free_object_quiescent`, `bi_stale_object_quiescent`) to invalidate their stale cache. BI tracks their invocation history to aid in the calculation of quiescence. To guarantee bounded incoherence, applications are in charge of invoking cache quiescence in a timely manner. §4.3 covers more details about on-demand cache quiescence. §6presents how to utilize both periodic and on-demand BI cache quiescence in a graph processing framework.

4.3 | BI writers

BI writers are more complicated due to several requirements: (1) Atomic update which prevents readers accessing incomplete data. (2) Exclusive modification requires coordination between updating nodes. Currently BI does not support parallel writers to the same cache line. Hardware atomic instructions are insufficient for mutual exclusion as their atomicity guarantees don't extend to memory, and our current hardware does not support in-memory atomic operations (e.g., `cas`), thus data-structures are partitioned across nodes. Each partition is exclusively accessed by one writer, and message passing is used to coordinate updates across nodes. (3) Cache line write-back must be done explicitly to modified data. Thus, modified cache lines are written back to memory immediately. This avoids readers reading stale data even after cache invalidation. BI writers also require both (4) stale cache line tracking to invalidate only the required cache lines, and (5) memory reclamation.

Stale cache line tracking. BI tracks two types of stale objects to be invalidated by readers. First, BI logs objects modified through `bi_assign_pointer` (e.g., object `a` in Figure 4A). This guarantees that a BI reader can retrieve the memory for updated objects with a bounded delay. Second, BI logs memory freed via `bi_free` (e.g., object `b` in Figure 4A). This enables the system to safely reuse this memory after achieving quiescence. These two kinds of objects are logged into two separate ring buffers, and object headers are annotated with the time that they were logged. The timestamp used for the memory reclamation is discussed next. On the other hand, readers invalidate modified and freed objects by invoking `bi_stale_object_quiescent` and `bi_free_object_quiescent`, respectively.

However, as discussed in §4.2, invalidating all modified objects is not the only way to deal with stale cache-lines. Thus BI allows users to flush any necessary cache lines in application-specific ways. When calling `bi_alloc`, applications set the `flag` to indicate whether BI should track modifications to the allocated object or not. When performing cache invalidation via `bi_stale_object_quiescent`, applications pass in a callback function (`call_back_fn`), which iterates through and flushes their managed objects.

Memory reclamation. Writers regularly call `bi_reclaim` to reclaim freed memory to avoid unbounded memory consumption. Before reclaiming an object, `synchronize_bi` is invoked to check if a grace period has elapsed since its deallocation. Formally, `synchronize_bi` returns a time point such that objects freed before that point can be safely reclaimed. `synchronize_bi`calculates a grace period by combining time-based and cache quiescence. Similar to existing work,[4] every reader records the times when they begin and end accessing the shared structure inside `bi_enter` and `bi_exit`. Writers retrieve this timing information to determine the time-based quiescence point (`Q`). However, with stale cache values, an object freed before `Q` is still visible and accessible by other sockets , unless its stale reference has been invalidated before `Q`. As an example, consider the object freed at `A` in Figure 6. Without cache invalidation at `B`, reader `R2`can still access that object at `D`. Therefore, from `Q`, `synchronize_bi` minus the length of cache quiescence, and returns that as the global quiescence point. `bi_reclaim` finally iterates the freed memory log, and reclaims any objects deallocated before the quiescence point.

4.4 | BI correctness

The key for BI correctness is to ensure following invariants on consistency, freshness, and memory utilization. (1) No objects should be modified while concurrent readers can potentially hold any type of references to them. This guarantees that readers always see consistent data. (2) All modifications should be visible to readers within at most a bounded amount of time, which is the BI grace period. This ensures that readers will access fresh data after a grace period. (3) All memory of freed objects should be eventually reclaimed. This prevents memory exhaustion due to delayed memory reclamation.

Consistent read. The reasoning about consistent reads and visibility of objects is similar to existing RCU techniques. Writers never modify objects in-place, but on a private copy instead and atomically modify a single pointer to replace the old object with the new. Memory reuse and its subsequent modification is delayed by the quiescence mechanism (§4.2). BI quiescence ensures that memory is reallocated and reused only after every reader both completes its read operation and flushes its stale cache.

Bounded staleness. This is guaranteed by coordination of readers and writers. Writers always commit changes back to memory immediately, and all modified objects are logged (§4.3). On the other hand, readers invalidate these logged objects periodically and are able to see the updated

data after cache invalidation (§4.2). In case of on-demand cache quiescence, it is the application's responsibility to provide this guarantee at their chosen granularity.

No memory leakage. As proved by Ren et al.,[8] an SMR implementation has bounded memory utilization as long as it has the following property: during every memory reclamation cycle, it collects all freed objects which have past the quiescence point. BI identifies and collects all such objects using two mechanisms (§4.3): (1) the quiescence calculations return the latest global quiescence point, and (2) BI always invalidates its internal logging structures to retrieve the up-to-date logged objects.

5 | BI IMPLEMENTATION

5.1 | BI runtime

We implement BI prototype as a run-time library addition to ParSec[4] which provides a slab memory allocator, SMR, to track possible parallel accesses to a non-blocking data-structure, and delays the re-use of `free`d memory until no such accesses can exist. BI extends ParSec to implement RCU-style APIs in §4.1.

Maintaining and using logical time. BI needs to compare different node's potential object accesses with when memory was freed, to determine if it can be reused. ParSec uses invariant TSC[19] support to determine this ordering. Unfortunately, in rack-scale systems, such architectural support cannot be assumed as nodes are more loosely coupled than cores on sockets. A simple implementation atomically increments a single logical clock each time an object is deallocated. The logical time is the value of that counter at any point in time, and tracks deallocations. BI records the logical time when each node invalidates its data-structure cache-lines. Unfortunately, this requires frequent modifications to the shared logical clock which not only involves memory-scale latency, but also contends the in-fabric atomic operation units.

Instead, the BI runtime uses a time-based implementation in which the logical counter is incremented periodically at some granularity related to the timer tick of each node. The current prototype uses the cycle counter (via `rdtsc`) on each node to calculate the logical time. While the absolute value of cycle counter may vary on different nodes, they are all incremented at the same fixed frequency. Thus, different nodes, though not tightly coupled, can maintain logical times within an error margin of a single logical tick. Any comparisons between clocks on different sockets must take into account the maximum difference between each node's logical time, $D = \max_{\forall i}(Q_i) - \min_{\forall i}(Q_i)$. An example: a data-structure, freed at time t has its `logged_time`=t. When a node attempts to reclaim the data-structure, the `enter` and `exit`access times to the data-structure are referenced on each node, compared against $t + D$ to compensate for time offsets on each node. Though it adds some pessimism in when objects can be reclaimed, this design enables the efficient, cached access to all node's logical clocks. Updating a node's own logical clock requires only writing the cache-line back to memory, and does not require atomic memory operations (e.g., a write-back via `clwb` is sufficient).

5.2 | BI pseudo-code

BI **Metadata.** Figure 7 presents the BI metadata. BI augments each object with a header (line 1-7), including the time it was logged, a flag and a mark to indicate if BI tracks its modifications and if it was freed, respectively (§4.3). BI tracks the time each reader enters and exits a read operation in a per-reader structure (line 11-14). This timing information is retrieved by writers when calculating quiescence (§4.3). Readers only update timing records, while writers only read them. Every writer maintains two ring buffers, as logs to track freed and modified objects (line 16-28). Readers

```
1  // object header prepend to each data-structure object
2  struct object_header {
3      time_t logged_time;
4      size_t sz;
5      bool log_flag;
6      bool free_mark;
7  };
8  // functions to convert between an object and its header
9  void *obj2header(void *);
10 void *header2obj(void *);
11 // per-reader timing information of read operations
12 struct timing_info {
13     time_t enter, exit;
14 };
15 struct timing_info reader_time[NUM_READERS];
16 // stale cache tracking logs for freed and modified objects
17 struct log_record {
18     void *addr;
19     size_t sz;
20 };
21 struct log_buffer {
22     int head, tail;
23     struct log_record records[MAX_LOG_SIZE];
24 };
25 struct log_buffer freed[NUM_WRITERS];
26 struct log_buffer modified[NUM_WRITERS];
27 // functions to get, add and remove objects from logs
28 struct log_record *peek(struct log_buffer *log);
29 void enqueue(struct log_buffer *log, void *p, size_t sz);
30 void dequeue(struct log_buffer *log);
```

FIGURE 7 BI metadata

```
1  void bi_enter(void) {
2      struct timing_info *t = &reader_time[thdid()];
3      t->enter = cur_time();
4      mem_barrier();
5  }
6  void bi_exit(void) {
7      struct timing_info *t = &reader_time[thdid()];
8      t->exit = cur_time();
9  }
10 void *bi_dereference(void **p) {
11     if (*p == NULL) clflushopt(p, sizeof(void **));
12     if (*p == NULL) return NULL;
13     struct object_header *h = obj2header(*p);
14     if (h->free_mark) clflushopt(*p, h->sz);
15     return *p;
16 }
17 void invalidate_stale_obj(struct log_buffer *buf) {
18     for(i = buf->head; i < buf->tail; i++) {
19         struct log_record *rec = &(buf->records[i]);
20         prefetch(rec + PREFETCH_OFFSET);
21         clflushopt(rec->addr, rec->sz);
22     }
23 }
24 void invalidate_log_buffer(struct log_buffer *buf) {
25     for(i = 0; i < NUM_WRITERS; i++) {
26         clflushopt(&buf[i], sizeof(struct log_buffer));
27     }
28     mem_barrier();
29     for(i = 0; i < NUM_WRITERS; i++) {
30         invalidate_stale_obj(&buf[i]);
31     }
32     mem_barrier();
33 }
34 void bi_stale_object_quiescent(callback_fn) {
35     if (callback_fn) callback_fn();
36     invalidate_log_buffer(modified);
37 }
38 void bi_free_object_quiescent(void) {
39     invalidate_log_buffer(freed);
40 }
41 void reader_engine(callback_fn) {
42     writeback(&reader_time[thdid()]);
43     bi_free_object_quiescent();
44     bi_stale_object_quiescent(callback_fn);
45 }
46
```

FIGURE 8 BI reader–related APIs. `thdid` returns the current thread id, `cur_time` returns the current logical time. `clflushopt` and `writeback` uses the `clflushopt` and `clwb` instructions, respectively

iterate over these logs and invalidate the cache of all logged objects to achieve cache quiescence (§4.2). Thus, these logs are manipulated by writers, and are read-only to readers.

Figure 8 presents the pseudo-code for the reader-related APIs.

Reader section. The start and end of a read operation is marked by `bi_enter` and `bi_exit`. They simply record the current logical time into that per-reader timing record (line 1-9). We use the typical memory barrier here to ensure that the store of the current time into `enter` is visible in cache (the store-buffer is flushed) before accessing the data-structure. If the `enter` time is larger than `exit`, it means a reader is currently executing a read operation. Otherwise, the reader has been finished.

In our first unoptimized implementation, these records are written back to memory immediately, which assures accurate quiescence detection. However the write-back is expensive, and significantly slows down read operations in the fast-path. Fortunately, immediate write-back is not necessary, and could be performed periodically in a later point. This may cause a problem only if the quiescence detection has wrongly ascertained that a reader has completed computation on the shared data-structure. By case analysis, such a mistake only occurs when `exit` is up-to-date, while `enter` has stale value. Hence, the quiescence detection always retrieves the value of `exit` first. In this way, the value of `exit`can never be newer than `enter`. §7.2 demonstrates the performance improvement of this optimization.

Read shared data. `bi_dereference` fetches a BI-protected pointer (line 10-16). In case of a resolution failure, it implements lazy invalidation (§3.2), which accesses the memory again after invalidating the pointer. Line 13-15 prevents a bug introduced by the hardware prefetcher. The hardware prefetcher can possibly prefetch a freed object into the cache, even after a reader achieves quiescence, thus the memory is no longer in the process of being freed. This breaks the BI invariant that a reader should no longer have reference to freed objects once achieving quiescence. To detect such a prefetch-after-invalidate race condition, the object is marked by `bi_free`. When a marked object is found, we invalidate the cache to get its latest value. Note that a stale pointer and/or header is allowed in cache as it might have been freed on another node, and this is a normal, (bounded) incoherent access. Instead, this logic protects against caches holding freed memory after it has quiesced, been reused, and relinked back into the structure.

Cache quiescence. Cache quiescence is achieved by invalidating both stale and freed objects stored in log buffers (line 34-40). For each type of objects, BI first iterates through each writer's log buffer and invalidates the buffer itself. We batch the invalidation of all logs (line 25-27), an optimization informed by Figure 3B. This guarantees access of the latest log contents. Then BI iterates the buffer again to invalidate each object's cache-lines (line 18-22). Explicit prefetch instructions are used to pre-load the recently flushed logged objects (line 20), largely avoiding memory latencies.

Read/writer synchronization. Interleavings between the quiescence code and `free`ing or modifying nodes can result in log items being added while the reader node is flushing. This can result in freed or modified memory being added to the log, but not being visible to the quiescence. All such recent additions to the log are for memory that has not yet quiesced (as it was just freed/modified), thus allowed to be inconsistent in caches as it must have been added after lines 25-27. In short, these cases are identical to the case where the `free`s or modifications are made directly following the quiescence operations. Log modifications are trivially synchronized as each log is only modified by a single node, and written back immediately when logged.

```
1  void bi_assign_pointer(void **p, void *v) {
2      struct object_header *h = obj2header(*p);
3      if (h->log_flag) {
4          h->logged_time = cur_time();
5          enqueue(&modified[thdid()], p, h->sz);
6      }
7      *p = v;
8      writeback(p);
9      mem_barrier();
10 }
11 time_t time_quiescence(void) {
12     time_t enter, exit, q = cur_time();
13     clflushopt(reader_time, sizeof(reader_time));
14     mem_barrier();
15     for(i = 0; i < NUM_READERS; i++) {
16         exit = reader_time[i].exit;
17         enter = reader_time[i].enter;
18         if (exit < enter) q = min(q, enter);
19     }
20     return q;
21 }
22 time_t cache_quiescence(time_t t) {
23     return t - CACHE_QUIESCENCE_PERIOD;
24 }
25 time_t synchronize_bi(void) {
26     time_t q = time_quiescence();
27     return cache_quiescence(q);
28 }
29 void *bi_alloc(size_t sz, bool flag) {
30     struct object_header *h;
31     h = malloc(sz + sizeof(struct object_header));
32     h->sz = sz;
33     h->log_flag = flag;
34     writeback(h);
35     return header2obj(h);
36 }
37 void bi_free(void *p) {
38     struct object_header *h = obj2header(p);
39     h->free_mark = true;
40     h->logged_time = cur_time();
41     writeback(h);
42     enqueue(&freed[thdid()], p, h->sz);
43 }
44 void bi_reclaim(void) {
45     struct object_header *h;
46     struct log_buffer *buf = &freed[thdid()];
47     time_t q = synchronize_bi();
48     clflushopt(buf, sizeof(struct log_buffer));
49     for(i = buf->head; i < buf->tail; i++) {
50         struct log_record *rec = peek(buf);
51         h = obj2header(rec->addr);
52         if (h->logged_time > q) break;
53         free(h);
54         dequeue(buf);
55     }
56 }
```

FIGURE 9 BI writer–related APIs

Reader engine. Putting it together, readers need to regularly perform some BI tasks, which are encapsulated in `reader_engine` (line 41-45). They write back timing information and perform cache quiescence. As discussed in §4.2, it can be periodically invoked by BI or explicitly called from applications.

The pseudo-code for the writer–related APIs appears in Figure 9.

Modify shared data. Writers use `bi_assign_pointer` to assign a new value to a BI-protected pointer (line 1-10), which effectively replaces an old object with a new one. If logging is enabled, the address of the changed pointer is saved, and the modification is written back to memory.

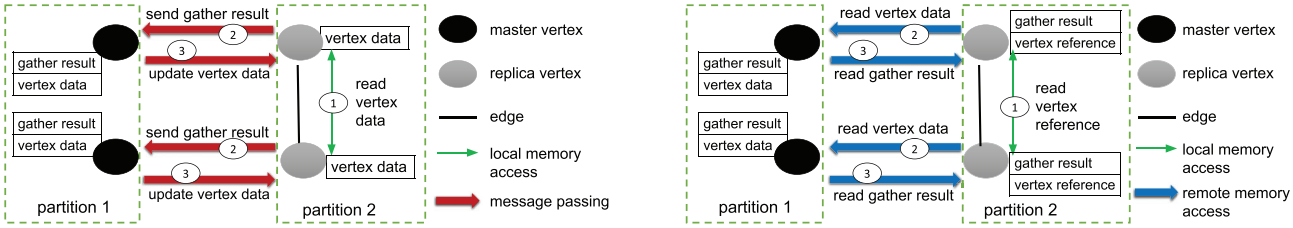
Quiescence calculation. Quiescence detection in `synchronize_bi` includes checking the reader's states based on their timing information, and validating cache quiescence (line 25-28). `time_quiescence` returns a time point `q` at which point every reader has exited the read section at least once. This function first invalidates the cache of timing records to fetch their latest value (line 12-14). Next, line 15-19 iterates through all reader's timing records to determine the time furthest in the past that all readers could not hold data-structure references (either now, if a reader has exited the data-structure, or the time it entered the data-structure). The minimal value of all of these times for all readers is returned as `q`, identical to the logic in Ren et al.[8]. With `q`, `cache_quiescence` further calculates the global grace period, when every reader has completed cache invalidation. For instance, with the periodic cache quiescence policy, the most recent cache quiescence point is simply `q` minus the cache invalidation period (line 22-24).

Memory management. Object allocation and release are done though `bi_alloc` and `bi_free` (line 29-43). `bi_alloc` allocates memory and prepares the object header. `bi_free` does not actually free the memory, but marks the object as freed and puts it into the log. The actual memory reclamation happens in `bi_reclaim`, which frees the memory of all objects that were deallocated before a grace period (line 44-55).

6 | BI USE CASE: POWERGRAPH

6.1 | `PowerGraph` background

`PowerGraph`[20] is a high performance graph computation framework. It supports both shared memory multi-processors and distributed clusters, and we investigate extending this support to non-CC memory. `PowerGraph` introduces a vertex cut to partition power-law graphs and a programming abstraction that supports parallel execution within a vertex computation. As a result, `PowerGraph` can scale to graphs with billions of vertices and edges.



(A) The communication pattern in the original PowerGraph. During the gather phase, a vertex reads adjacent vertices data locally (1). Then each replica sends gather result to the master via message passing (2). The master sends message with the updated vertex data to its all replicas (3).

(B) The communication pattern in the BI enabled PowerGraph. For every replica, BI replaces the actual vertex data with a reference to its master, and saves the gather result locally. During the gather phase, a vertex first gets the reference to adjacent vertices master (1), then fetches the actual data from remote master (2). After replicas calculate and save gather results locally, the master collects gather results from its remote replicas (3), and updates the vertex data accordingly. The master no longer needs to update remote replicas thenceforth.

FIGURE 10 The communication pattern in the original (A) and BI enabled (B) PowerGraph

Vertex Cut. PowerGraph evenly assigns edges to computation nodes and allows vertices to span multiple nodes. For each vertex that spans multiple nodes, it has a replica on each spanning node. One of the replicas is randomly assigned the master role, and the rest are read-only replicas. While vertex data can be retrieved locally from the local read-only replica, changes to vertex must be broadcast to all its replicas by the master. Such communication is implemented by MPI-based message passing. Since each edge is stored exactly once on the node it is assigned to, changes to edge data do not need any communication or synchronization across nodes.

GAS Vertex-Programs. Computation in PowerGraph is encoded as a state-less vertex-program, which implements the GAS model and explicitly factors into three conceptual phases: gather, apply, and scatter. The gather phase is applied to all replicas of vertices in parallel. During the gather phase, a vertex (maybe a replica) collects information about adjacent vertices and edges locally through a user-defined `sum` function. The `sum` function is required to be commutative and associative. Every replica sends its local result to its master replica, which combines all results using the same `sum` function. The final combined result is passed to the apply phase. After the gather phase has completed, the apply phase is invoked only on the master replica. Each master replica uses the gather result to update the vertex data via a user-defined `apply` function. The updated vertex data is then copied to all replicas by message passing. The scatter phase runs in parallel on all adjacent edges of updated vertices. It updates the edge data according to the new vertex data. Figure 10A shows the communication among replicas and masters within each phase.

6.2 | Challenges with non-CC memory

With non-CC memory, there are a number of complications to the PowerGraph design. The communication across replicas makes no use of shared-memory and it exposes message passing overhead. At scale, this message passing prohibits the effective use of an increasing number of cores. Message passing's overheads on non-CC systems are due to (1) the sporadic, short bursts of cache operations for message passing (write-back on the sender, and invalidation on the receiver) which don't leverage the pipelining of many cache operations required for acceptable overhead (Figure 3B), and (2) the polling at memory speed on a number of message queues equal to the number of communicating cores/replicas. The potential large amount of memory used by PowerGraph also complicates cache quiescence, which might require a huge number of cache-line invalidation operations. Worse still, the apply phase can be update-intensive, where traditional SMR and RCU techniques offer very little benefit. Care is taken to minimize modifications to remote cache lines in other nodes. Fortunately, some parts of PowerGraph abstractions do have some appealing characteristics for non-CC memory systems. Primarily, the whole graph is well partitioned, requiring no concurrent or atomic modifications. The edge data is totally local, therefore we only need to focus on the maintenance of the vertex data.

6.3 | BI PowerGraph implementation

We port PowerGraph to BI based on the open source GraphLab C++ implementation*. It provides different options to configure the PowerGraph engine, and we use the `synchronous` option. With the `synchronous` engine, PowerGraph employs the bulk synchronous parallel (BSP) model, and executes the gather, apply, and scatter phases in order with a barrier at the end of each phase. The porting process involves replacing the memory management facilities with the BI memory allocator with non-CC memory as backend. All message passing of vertex replica maintenance is replaced by using global shared memory. Access to the shared memory in gather and apply phase is managed by the BI runtime, which manages

*https://github.com/jegonzal/PowerGraph

the non-coherent cache and limits stale data to at most a bounded amount of time. The rest of the system, such as the partition strategy, vertex scheduling and scatter phase is left unchanged.

API usage in `PowerGraph`. `PowerGraph` uses statically allocated sets of vertices and edges. As there is no dynamic allocation for the graph, we use only the subset of the BI API focused around tracking modifications, accessing the shared memory, and providing cache quiescence. This means that the focus on the `PowerGraph` adaptation to BI is on the functions `bi_assign_pointer`, `bi_dereference`, and `reader_engine`. Together, these enable the tracking of modified vertices, properly accessing updated vertices, and performing quiescence based on those modifications.

Gather Phase. During the gather phase, all replicas send their local gather result to their masters. For BI, we augment the vertex data-structure with a new field to save the gather result locally. At the end of the gather phase, instead of each replica sending its own result to the master, the master replica directly accesses all its replica's data-structures to read their results and combines them to get the final result. Consequently, all modifications in this phase are purely local. Only masters need to read remote cache lines, which are made visible by the BI runtime.

Apply Phase. The apply phase updates all replicas with the updated vertex data. To avoid remote modifications, BI changes the replica structure by saving a reference to its corresponding master replica, instead of saving the actual data. Hence, after the master updates the vertex data, it no longer needs to send a message to notify its replicas. On the contrary, whenever a replica requires its vertex data (e.g., in the gather or scatter phase), it reads the data from its master via the reference. The BI runtime guarantees that the master's up-to-date data will become visible to replicas within at most a bounded amount of time. This totally eliminates message passing and remote modifications. Figure 10B shows the details of the communication among replicas and masters in BI enabled `PowerGraph`. Note that if quiescence is not aligned to per-pass BSP synchronizations, replicas can see stale data here. The impact of this staleness is discussed below in the "Staleness Analysis".

Cache Quiescence. Cache quiescence is necessary to provide cache coherence in two cases. First, in the gather phase, local gather results are required to be visible to the master. Second, in the apply phase, vertex replicas need to see the updated vertex data. In both cases, we choose to invalidate only *modified* cache lines as `PowerGraph` potentially accesses a huge amount of unmodified memory.

Cache quiescence is implemented in two ways. First, we mark the gather results and the replica structure as BI-managed. This enables BI built-in support to track modified objects and flush them periodically in the background. The cache flush is carried out by one core per socket, as we observe that invalidation on one core will flush all other cores within the same socket, even in non-shared caches (L1 and L2). In this way, the application is totally freed from reasoning about cache coherence. In the second implementation, we utilize the fact that `PowerGraph` already has information about active and modified vertices. Therefore, we extend the BSP barrier to invoke the BI cache quiescence on-demand. This iterates `PowerGraph` internal vertex set structure to identify all modified objects and invalidates their cache lines. This prevents stale cache lines across phases, as discussed below.

Staleness Analysis. Staleness is introduced by the BI's periodic cache flushes. If a core reads a modified remote cache line before BI invalidates that cache line, it will see the stale value instead of the updated one. This happens in both gather and apply phase, where a master may use an old gather result or a replica can see vertex data from a previous iteration. However, such staleness is bounded by the BI grace period. As studied in previous research,[21,22] a large class of iterative graph and machine learning algorithms are proved to converge even in the face of staleness between iterations, as long as such staleness is restricted within a limited amount of time. It is the ability of many graph algorithms to converge in the face of stale information that motivated our investigation of them with BI.

On the other hand, on-demand cache quiescence gives applications full control of data consistency. When `PowerGraph` invokes BI cache quiescence inside the BSP barrier, it guarantees that all changes made in the current phase will be seen by the next phase. As a consequence, no stale data is generated in such implementation, thus enabling us to study the impact of staleness on application performance. While not implemented in this work, we can also trigger cache quiescence only within specific iterations to achieve the A-BSP or SSP model;[21]

7 | EVALUATION

Experiment Platform. All experiments are run on HPE Superdome Flex servers[†]. We deploy two enclosures, with four 28-core sockets per enclosure. The Intel(R) Xeon(R) Platinum 8180 CPU is used, which is clocked at 2.5GHz. Each core has a 32KB L1 cache and 1MB L2 cache, and each socket has 38.5MB L3 cache. Each enclosure has 1.568TB local memory. There are 3.008TB global memory shared between two enclosures via the NUMALink fabric. Custom firmware is installed to configure cache coherency on top of global shared memory. The cache coherency domain is at the socket level. That is to say, when cache coherency is disabled, only cores inside the same socket are coherent, and caches between different sockets (even within the same enclosure) are non-coherent. A `clflush` on a core will write-back and invalidate that cache-line in all caches (including non-shared L1 & L2) on the socket. Each enclosure runs an independent copy of the SLES-15 operating system, with a Linux 4.12.14 kernel.

[†]https://www.hpe.com/us/en/servers/superdome.html

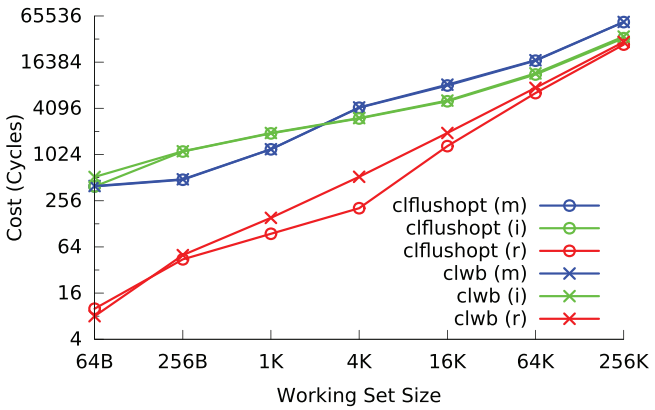


FIGURE 11 `clwb` and `clflushopt` overhead

7.1 | Micro-benchmarks

Cache operation overheads. BI uses `clwb` and `clflushopt` instructions to achieve bounded cache coherency. `clwb` is used by writers to commit modifications to memory, and `clflushopt` is used by readers to invalidate stale cache lines. To investigate the overhead of those instructions, Figure 11 depicts their cost of operating on an increasing amount of continuous non-coherent memory. This experiment runs on a single core and measures the cost with cache lines in different states (read, written to, or not in cache).

Those overheads are linear with the working set size. With a large working set, the memory latency dominates the cache overhead, effectively requiring memory latency for all operations since the targeted cache line is never in cache. In this case, these cache operations make a negligible difference over the memory latency, thus we only show results from working sets smaller than 256KB. With a small working set, the cost of both instructions are observable but not prohibitive. More importantly, on the non-coherent architecture, such overhead does not increase with the scale of the machine, as their impact is limited only to their own socket. In general, `clwb` has a little less overhead than `clflushopt`, as it does not invalidate operated cache lines. Furthermore, this avoids cache misses on the following memory accesses. The cache line status has a bigger impact on the cache operation as expected. Operations on read-only cache lines have the least overhead, as no memory access is triggered. On the other hand, operations on modified cache lines are the most expensive, since modifications are written back to memory. When cache lines are invalidated, the operating core is not aware if other cores contain the same cache line; thus, it needs to wait for invalidation confirmations from other cores in the coherency domain (one socket in this case), causing some overheads.

Periodic Cache Invalidation Overheads When BI is configured to use periodic cache quiescence, two factors determine the total overhead of its cache invalidation: invalidation frequency and working set size. Furthermore, the cache invalidation impacts application performance in two ways. First it reduces the available CPU time to applications running on the same core with cache invalidation. Second, it causes cache misses of operations accessing the invalidated memory on other cores inside the same coherency domain. To understand the impact of these factors, we measure memory load throughput over different working sets in the presence of cache invalidation at various frequencies.

Figure 12A depicts the load throughput while cache invalidation is executed on the same core as the test. Figure 12B reports results of tests running on a different core. All throughputs are normalized to the performance without periodic cache invalidation (`noflush`). `aggressive` represents the case that cache is invalidated immediately once it is assessed, instead of being delayed to periodic invalidation. Those results show aggressive cache invalidation performs the worst in all cases. With a larger working set, cache invalidation has more impact as expected, due to the increased cache miss overhead. For example, when the working set fits into L2 cache, periodic cache invalidation at most introduces 2% performance degradation. When the working set becomes 64MB, load throughout decreases 20% and 10%, on the same and separate core, respectively. Similarly, more frequent cache invalidation has more overhead as well. For example, in Figure 12B, a 100ms invalidation period has 2% throughput degradation, while 10ms period has 10% degradation.

7.2 | BI Applied to a balanced-tree data-structure

To evaluate BI on complex and efficient data structures, we implement a BI-based concurrent red-black tree. Red-black trees are commonly used in operating system kernels and language runtimes. A red-black tree represents a set of unique key and value pairs. It has three main operations, `lookup` which is read only, `insert` and `delete` which are mutating operations. Recent research improve the scalability of a concurrent red-black tree by utilizing RCU techniques.[17,23] Hence, the red-black tree is a representative use case for BI, and we port BI to the BONSAI tree implementation.[23] BONSAI tree's `lookup` operation is guarded by RCU sections, and its write operation is protected by a lock[‡]. We use the U-RCU[7]

[‡]https://github.com/tpapagian/pk/blob/rcuvm-pure/lib/cbtree.c

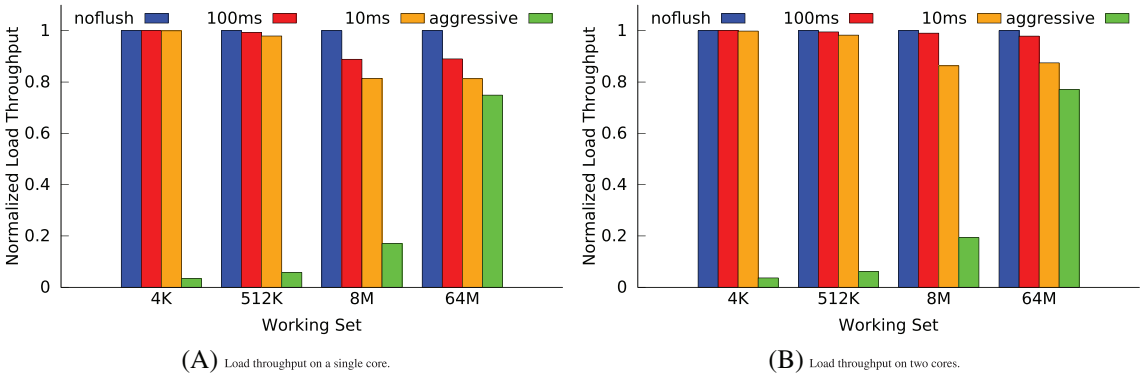


FIGURE 12 Load throughput with cache invalidation

TABLE 1 Red-black tree performance results. Working set is the number of tree nodes

A. Per-core throughput improvements of reader section optimization (higher is better)				
Working set	**1K**			
Update percentage	10%	30%		
non-opt BI	40K ops/s	14.3K ops/s		
BI	147.5K ops/s	40K ops/s		
B. Comparison of write operation latency (lower is better)				
Update percentage	10%			
Working set	64K	64M		
U-RCU	1700K cycles	811K cycles		
BI	544K cycles	467K cycles		
C. Comparison of read operation latency (lower is better)				
Working set	64K		64M	
Update percentage	0%	10%	0%	10%
U-RCU	207 cycles	400 cycles	2600 cycles	2700 cycles
BI	200 cycles	550 cycles	1500 cycles	2500 cycles

library `liburcu`[§] for RCU APIs. Porting BI to BONSAI tree is straightforward. BI APIs are used as a drop in replacement of RCU APIs in the `lookup` operation. All write operations are delegated to a single core by message passing, thus avoiding concurrent write operations and cache invalidation inside writers.

Methodology There are two main dimensions we parameterize the experiments around. (1) Update percentage. BI works best with a read-mostly data-structure. Thus studying the impact of update operations can shed light on BI limitations. (2) Working set size. Different working sets introduce different cache and memory footprints, and thus stress the cache coherency and memory management in BI. We use the number of tree nodes inside a tree as a measurement of the working set size. All experiments use all available cores in system, with one benchmark thread per core. Each benchmark populates a tree with the given working set size, and a key range doubling the initial size. In each thread's iteration, it picks a key from the key range randomly, then determines if the operation is an update operation according to the update percentage. Update operations roughly include half `insert`s, and half `delete`s. The entire tree is saved in the global shared memory. Hardware cache coherency is enabled when running U-RCU experiments, and is disabled for BI. BI uses the default periodical cache quiescence policy with one millisecond period, and flushes all modified objects. We measure throughput and latency, and present all performance results in Table 1. These results allow us to study how close to hardware-managed coherence performance we can achieve while using software-managed non-coherent memory with BI.

[§] https://github.com/urcu/userspace-rcu

Optimization improvements. Table 1A studies the performance improvement of the optimization introduced in §5.2, which defers the write-back of timing records inside `bi_enter` and `bi_exit`. With this optimization, read operations avoid the cost of cache operation and memory access. As a result, the average per-core throughput is over three times higher than the non-optimized implementation.

Write operations. The latency of write operations is shown in Table 1B. BI shows significantly better performance than U-RCU in all cases. The blocking quiescence detection in U-RCU cannot scale to such a large machine, and the writer lock triggers too much cache coherency traffic. In contrast, BI serializes updates into a single core, thus using a delegation-based synchronization rather than locks or atomic instructions. If a single core becomes the bottleneck, BI can utilize more advanced delegation techniques[24] to use more cores. In both U-RCU and BI, writer operations run faster with larger working set sizes. This is an indirect result of slower read operations in a larger working set. Slower read operations decrease the frequency of updates, thus introduce less contention on locks and the delegation core, in U-RCU and BI, respectively.

Read operations. The read operation latency in Table 1C shows a number of interesting effects. With the read-only case (0% update percentage), BI tends to be faster than U-RCU. We believe this is the outcome of increased hardware efficiency when cache coherency is disabled. However, with 10% update percentage and smaller trees, BI performs worse than U-RCU. This is the effect of lazy invalidation and the defensive checking for the hardware prefetcher's actions (§5.2). We expect future rack-scale machines will have more facilities to better handle the interleaved prefecher and cache invalidation, thus eliminating this overhead. When tree size becomes larger, as expected, both approaches get higher latency due to the algorithmic time complexity of red-black tree. However, the slowdown of BI is smaller than U-RCU. With larger tree, more cache-lines are touched, resulting in U-RCU demonstrating more cache coherency costs, which are avoided in BI.

We find these results to be compelling for BI. They demonstrate that a relatively complex data-structure written for the RCU can be adapted to BI in a relatively straightforward manner. This is aided by the fact that the BI API was designed as a derivative of the RCU API. That BI achieves roughly comparable performance as RCU, and better in some cases demonstrates that even if future systems scale to the point that they require non-coherency, BI will still enable shared data-structure processing.

7.3 | Graph processing framework

This section evaluates how `PowerGraph` can harness the benefit of BI. To study the different trade-offs of different approaches, we compare the original distributed `PowerGraph` with the two BI variants discussed in §6.3.

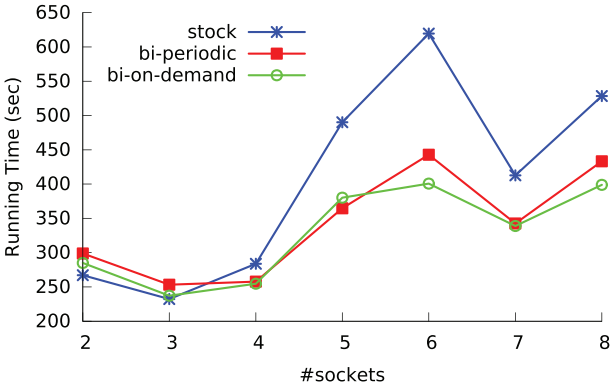
Methodology We run one `PowerGraph` instance per socket, which uses all available cores in that socket. All graph vertex data is loaded into the global shared memory, and is coordinated among `PowerGraph` instances differently according to different implementations. Edge data is saved in local memory, and needs no synchronization. To compare alternative design decisions studied in the literature, we consider three implementations. (1) `stock` – all data synchronisation is achieved by MPI message passing using the system with coherency enabled. (2) BI`-on-demand` – cache quiescence is invoked by `PowerGraph` inside the BSP barrier after each phase thus ensuring no cross-phase stale data exactly matching the semantics of `stock`. (3) BI`-periodic` – data coherency is handled by BI periodic cache quiescence which adds potential inter-phase staleness.

Experiment Set-up. To characterize the performance, we measure the total running time of PageRank algorithm provided by `PowerGraph`. PageRank runs on a Twitter follower graph,[25] which has 4.1 million vertices and 1.4 billion edges. Message passing is based on MPICH2 library. The cache quiescence period in BI`-periodic` is set to one millisecond.

Result Discussion. Figure 13 reports the running time with different number of sockets. All sockets are evenly assigned to two enclosures. With a small number of sockets, BI variants run slightly slower than the original `PowerGraph`. For example, on two sockets, BI`-on-demand` and BI`-periodic` is 11% and 6% slower respectively. This is because `PowerGraph` incurs less message passing overhead on fewer sockets, while BI pays the cost of its cache quiescence. When the socket count grows, as expected, `PowerGraph` degrades due to the high communication overhead among instances. On the other hand, both BI implementations become faster than the original version, thanks to their shared memory access. With eight sockets, BI`-on-demand`and BI`-periodic` runs 32% and 21% faster respectively. This confirms that the batched cache invalidation made by BI has much less cost than message passing, while providing local cache access and data coherency within bounded time. On average, BI`-on-demand` runs 5% faster than BI`-periodic` resulting from two factors. First, BI`-on-demand` tracks modified cache lines more accurately because it utilizes more application specific information. Second, BI`-on-demand` avoids stale data by flushing cache lines immediately after each phase, and BI`-periodic`'s stale accesses might impact the graph algorithm's convergence which can increase the runtime.

The `PowerGraph` BI results demonstrate that even an application not written for the RCU API can benefit from the modification tracking and quiescence that BI provides. The ability for the BI variants to perform better than the stock version demonstrates the potential for maintaining shared memory in spite of potential non-coherency. The relatively comparative performance of the BI variants demonstrates that for applications that can accept some stale data, automatic, periodic quiescence and modification tracking doesn't have to come at a prohibitive cost.

FIGURE 13 Pagerank on twitter graph



8 | RELATED WORK

Scalable memory reclamation. BI borrows heavily from SMR techniques such as epoch-based reclamation,[6] RCU,[7] ParSec,[4] and IBR.[26] Such approaches seek to determine if references exist into a data-structure from any parallel execution before re-using a freed allocation. However, these techniques only check if parallel executions are completed, ignoring the fact that references can possibly remain in stale cache lines. BI extends these techniques to determine if stale cache references can exist on any node, and by optimizing batched flushes.

Data-consistency and non-CC memory. Atlas[27] integrates the cache flushes into an acquire/release concurrency model based on locks, mainly targeting NVM. Atlas takes advantage of the acquire-release consistency guarantees provided by locks, and batches cache operations until a lock is released, at that point making all memory changes globally visible. In this way, cache operations on objects accessed in a critical section are delayed until its exit. BI instead focuses on cache-latency data-structure lookups, and batched, delayed cache-line invalidation, and trades being less general across data-structures. Similarly, Treadmarks[28] integrates consistency with lock semantics, and distributed shared memory implementations manually overlays consistency over a network.[29-31] Some research[21,22,32] explicitly relax data consistency and introduce data staleness in distributed systems. Bounded staleness[21] is exploited to accelerate big data analytics, where the algorithm can see old data from previous iterations. Lazygraph[22] proposes lazy data coherency among vertex replicas, causing replicas to have different views of each other.

Non-CC nodes as a distributed system. Scale-out systems distribute data across a cluster,[33] in some cases by relaxing consistency.[34] Some systems treat a single system as one that is distributed,[35-37] and use message-passing-based coordination.[9] Message passing is traditionally used to implement distributed shared memory[28,29,31,38-40] and provide partitioned global address space (PGAS) abstraction.[41,42] Grappa[39] distributes computation across a cluster with an optimized PGAS implementation. Argo,[40] a software distributed shared memory system, distributes coherence decisions using self-invalidation and self-downgrade combined with hierarchical queue delegation locks. Hare[10] uses message passing to implement a distributed file-system across nodes in a non-CC system. libMPNode[43] implements an OpenMP runtime for incoherent domains. It leverages thread migration and distributed shared memory to provide consistency between incoherent nodes. Instead, BI enables global shared data-structures to be accessed locally at cache-latency, while avoiding message passing as much as possible.

CREW data-structures and RDMA. The concurrent-read, exclusive writer model simplifies modifications as it prevents writer concurrency. Many RCU structures require this model, and rely on single atomic modification to update the data-structure. These structures often require locks to serialize concurrent modifications,[4,7] though some techniques use fine-grained locking.[16-18]

GAM[44] provides a directory-based cache coherence protocol over RDMA. Systems such as FaRM and RackOut[45-47] treat a cluster as a non-CC NUMA machine with RDMA-accessible remote memory. They use similar techniques (e.g., epoch-based memory reclamation[6]), but don't support cached-access to remote memory. In contrast, BI enables the cache-based access to global structures on rack-scale systems.

9 | FUTURE WORK

Though we believe that this research demonstrates the ability of BI to provide a programming model for specific types of applications on non-coherent, shared memory systems, there are potential directions to take the research further. These include: (1) increasing the scope of the technique by generalizing the programming model to enable applications to more tightly control the batching of modifications and quiescence to more tightly control staleness, (2) to evaluate BI on various other RCU-based data-structures, and (3) to port the ideas behind BI to other non-coherent, remote memory systems such as RDMA. Though a simpler version of BI has been applied to a single-system image for a simple microkernel,[48] an interesting direction is to apply tighter controls on staleness to more conventional OS data-structures like those in Linux (beyond those that are RCU-based).

The substantial memory scaling of rack-scale systems is enabled by Non Volatile Memory (NVM). In this paper we assume that local DRAM and global NVM are accessed independently. We focus on creating abstractions to handle non-CC memory, instead of on its non-volatility. Our design is applicable to other models, such as DRAM serving as a cache to NVM. In the future work, we will explore non-volatility in more detail.

10 | CONCLUSIONS

This paper has introduced the bounded incoherence memory consistency model for non-CC systems that enables cache-speed reads, and effective use of delayed, batched coherence. We apply BI to `PowerGraph`, and demonstrate that efficient, local access to cached data-structures can provide 30% performance improvements over distributed approaches.

We believe that BI mark significant steps toward enabling efficient management and sharing of non-coherent memory in future rack-scale systems.

ACKNOWLEDGMENTS

We'd like to thank Brad Tanner, Rocky Craig, Bill Hayes, Michael Woodacre, Keith Packard, Paolo Faraboschi and Robert Peter Haddad for their enormous help.

DATA AVAILABILITY STATEMENT

Data is not available.

ORCID

Yuxin Ren https://orcid.org/0000-0003-2678-9225

REFERENCES

1. Asanovic K. FireBox: a hardware building block for 2020 warehouse-scale computers. Paper presented at: Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14) USENIX; 2014; Santa Clara, CA.
2. Intel Corporation *Intel Rack Scale Design*. 2016. http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-architecture/intel-rack-scale-architecture-resources.html.
3. Faraboschi P, Keeton K, Marsland T, Milojicic D. Beyond processor-centric operating systems. Paper presented at: Proceedings of the 15th Workshop on Hot Topics in Operating Systems, HotOS XV; May 18-20 USENIX; 2015; Kartause, Ittingen, Switzerland.
4. Wang Q, Stamler T, Parmer G. Parallel sections: scaling system-level data-structures. Paper presented at: Proceedings of the ACM EuroSys Conference; 2016; ACM, New York, NY.
5. Prasad A, Gopinath K. Prudent memory reclamation in procrastination-based synchronization. Paper presented at: Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16); April 2-6, 2016; ACM, Atlanta, GA.
6. Hart TE, McKenney PE, Brown AD, Walpole J. Performance of memory reclamation for lockless synchronization. *J Parallel Distrib Comput*. 2007;67(12):1270-1285.
7. Desnoyers M, McKenney PE, Stern AS, Dagenais MR, Walpole J. User-level implementations of read-copy update. *IEEE Trans Parall Distrib Syst*. 2012;23(2):375-382.
8. Ren Y, Guyue L, Parmer G, Brandenburg B. Scalable memory reclamation for multi-core, real-time systems. Paper presented at: Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). Held at Porto, Portugal; 2018:25-27; IEEE.
9. Baumann A, Barham P, Dagand PE, et al. The multikernel: a new OS architecture for scalable multicore systems. Paper presented at: Proceedings of the Symposium on Operating System Principles (SOSP); 2009:29-44; ACM, New York, NY.
10. Gruenwald C, Sironi F, Kaashoek MF, Zeldovich N. Hare: a file system for non-cache-coherent Multicores. Paper presented at: Proceedings of the 10th European Conference on Computer Systems (Eurosys '15); 2015:1-14; ACM, New York, NY.
11. Van der Wijngaart RF, Mattson TG, Haas W. Light-weight communications on Intel's single-chip cloud computer processor. *ACM SIGOPS Operat Syst Rev*. 2011;45(1):73-83.
12. Harris T. Hardware trends: challenges and opportunities in distributed computing. *ACM SIGACT News*. 2015;46(2):89-95.
13. Prakash S, Lee YH, Johnson T. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans Comput*. 1994;43(5):549-559.
14. Michael MM. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Trans Parall Distrib Syst*. 2004;15(6):491-504.
15. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans Comput*. 1979;28(9):690-691.
16. Matveev A, Shavit N, Felber P, Marlier P. Read-log-update: a lightweight synchronization mechanism for concurrent programming. Paper presented at: Proceedings of the 25th Symposium on Operating Systems Principles SOSP '15; 2015:168-183; ACM, New York, NY.
17. Arbel M, Attiya H. Concurrent updates with RCU: search tree as an example. Paper presented at: Proceedings of the 2014ACM Symposium on Principles of Distributed Computing PODC '14; 2014; ACM, New York, NY
18. Clements AT, Kaashoek MF, Zeldovich N. RadixVM: scalable address spaces for multithreaded applications. Paper presented at: Proceedings of the ACM EuroSys Conference (EuroSys 2013); 2013; ACM, Prague, Czech Republic
19. Intel Corporation Intel-64 and IA-32 architectures software developer's manual, Volume 3A: system programming guide, Part 1. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.
20. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: distributed graph-parallel computation on natural graphs. Paper presented at: Proceedings of the Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12) USENIX; 2012; Hollywood, CA
21. Cui H, Cipar J, Ho Q, et al. Exploiting bounded staleness to speed up big data analytics. Paper presented at: Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14) USENIX; 2014:37-48; Philadelphia, PA.
22. Wang L, Zhuang L, Chen J, et al. Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation. Paper presented at: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '18; 2018; ACM, New York, NY.

23. Clements AT, Kaashoek MF, Zeldovich N. Scalable address spaces using RCU balanced trees. Paper presented at: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems; 2012; ACM, New York, NY.
24. Ren Y, Parmer G. Scalable data-structures with hierarchical, distributed delegation. Paper presented at: Proceedings of the 20th International Middleware Conference Middleware '19; 2019; ACM, New York, NY.
25. Kwak H, Lee C, Park H, Moon S. What is twitter, a social network or a news media?. Paper presented at: Proceedings of the 19th International Conference on World Wide Web WWW '10; 2010; ACM, New York, NY.
26. Wen H, Izraelevitz J, Cai W, Beadle HA, Scott ML. Interval-based memory reclamation. Paper presented at: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '18. Held at Vienna, Austria: ACM, New York, NY; 2018.
27. Chakrabarti DR, Boehm HJ, Bhandari K. Atlas: leveraging locks for non-volatile memory consistency. Paper presented at: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14); 2014; ACM, New York, NY
28. Keleher P, Cox AL, Dwarkadas S, Zwaenepoel W. TreadMarks: distributed shared memory on standard workstations and operating systems. Paper presented at: Proceedings of the USENIX Winter Technical Conference; January 17-21 USENIX, 1994; San Francisco, CA.
29. Scales DJ, Gharachorloo K. Towards transparent and efficient software distributed shared memory. Paper presented at: Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97); October 5-8, 1997; ACM, St. Malo, France.
30. Stets R, Dwarkadas S, Hardavellas N, et al. Cashmere-2L: software coherent shared memory on a clustered remote-write network. Paper presented at: Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97); October 5-8, 1997; ACM, St. Malo, France.
31. Johnson KL, Kaashoek MF, Wallach DA. CRL: high-performance all-software distributed shared memory. Paper presented at: Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95), Copper Mountain Resort; December 3-6, 1995; ACM, Colorado.
32. Ren Y, Parmer G, Milojicic D. Bounded incoherence: a programming model for non-cache-coherent shared memory architectures. Paper presented at: Proceedings of the 11th International Workshop on Programming Models and Applications for Multicores and Manycores PMAM'20; 2020; ACM, New York, NY.
33. Glendenning L, Beschastnikh I, Krishnamurthy A, Anderson T. Scalable consistency in scatter. Paper presented at: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11); October 23-26, 2011; ACM, Cascais, Portugal.
34. DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store. Paper presented at: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07); October 14-17, 2007; ACM, New York, NY.
35. Govil K, Teodosiu D, Huang Y, Rosenblum M. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. Paper presented at: Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP'99); December 12-15, 1999; ACM, Kiawah Island Resort, South Carolina.
36. Bugnion E, Devine S, Rosenblum M. Disco: running commodity operating systems on scalable multiprocessors. Paper presented at: Proceedings of the 16th ACM symposium on Operating Systems Principles SOSP '97; 1997:143-156; New York, NY.
37. Chapin J, Rosenblum M, Devine S, Lahiri T, Teodosiu D, Gupta A. Hive: fault containment for shared-memory multiprocessors. *SIGOPS Operat Syst Rev*. 1995;29(5):12-25.
38. Carter JB, Zwaenepoel W. Munin: distributed shared memory based on type-specific memory coherence. Paper presented at: Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming; 1990; ACM, New York, NY.
39. Nelson J, Holt B, Myers B, et al. Latency-tolerant software distributed shared memory. Paper presented at: Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15); 2015; USENIX, Santa Clara, CA
40. Kaxiras S, Klaftenegger D, Norgren M, Ros A, Sagonas K. Turning centralized coherence and distributed critical-section execution on their head: a new approach for scalable distributed shared memory. Paper presented at: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing HPDC '15; 2015; ACM, New York, NY.
41. Charles P, Grothoff C, Saraswat V, et al. X10: an object-oriented approach to non-uniform cluster computing. Paper presented at: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications OOPSLA '05; 2005; ACM, New York, NY.
42. Coarfa C, Dotsenko Y, Mellor-Crummey J, et al. An evaluation of global address space languages: co-array Fortran and unified parallel C. Paper presented at: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '05; 2005; ACM, New York, NY.
43. Lyerly R, Kim SH, Ravindran B. libMPNode: an OpenMP runtime for parallel processing across incoherent domains. Paper presented at: Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores PMAM'19; 2019; ACM, New York, NY.
44. Cai Q, Guo W, Zhang H, et al. Efficient distributed memory management with RDMA and caching. *Proc VLDB Endow*. 2018;11(11):1604-1617.
45. Dragojević A, Narayanan D, Hodson O, Castro M. FaRM: fast remote memory. Paper presented at: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14); April 2-4, 2014:401-414; USENIX, Seattle, WA.
46. Dragojević A, Narayanan D, Nightingale EB, et al. No compromises: distributed transactions with consistency, availability, and performance. Paper presented at: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15); October 4-7, 2015:54-70; ACM, Monterey, CA.
47. Novakovic S, Daglis A, Bugnion E, Falsafi B, Grot B. The case for RackOut: scalable data serving using rack-scale systems. Paper presented at: Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16); October 5-7, 2016:182-195; ACM, Santa Clara, CA.
48. Ren Y, Parmer G, Milojicic D. Ch'i: scaling microkernel capabilities in cache-incoherent systems. Paper presented at: Proceedings of the IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS); 2020; IEEE/ACM, New York, NY.

How to cite this article: Ren Y, Parmer G, Milojicic D. Sharing non-cache-coherent memory with bounded incoherence. *Concurrency Computat Pract Exper*. 2021;e6414. https://doi.org/10.1002/cpe.6414