Bounded Incoherence: A Programming Model for Non-Cache-Coherent Shared Memory Architectures

Yuxin Ren
The George Washington University
Washington, DC, USA
ryx@gwmail.gwu.edu

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

Dejan Milojicic
Hewlett Packard Labs
Palo Alto, CA, USA
dejan.milojicic@hpe.com

Abstract

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 programming and 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.

CCS Concepts • **Computer systems organization → Multicore architectures**; • **Computing methodologies → Parallel computing methodologies**;

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

ACM Reference Format:
Yuxin Ren, Gabriel Parmer, and Dejan Milojicic. 2020. Bounded Incoherence: A Programming Model for Non-Cache-Coherent Shared Memory Architectures . In *The 11th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'20), February 22, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3380536.3380541

1 Introduction

Recently, rack-scale systems have been gaining momentum. These include FireBox [2] from Berkeley, Rack-scale Architecture [25] from Intel, and The Machine from Hewlett Packard Enterprise [17]. 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 multiprocessors. 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 [14, 23, 40, 41, 47]. 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, periodically invalidates possibly stale cache-lines and delays memory reuse.

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

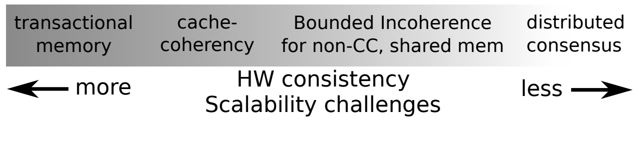

Figure 1. Varying hardware support for consistency.

PMAM'20 , February 22, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7522-1/20/02...$15.00
https://doi.org/10.1145/3380536.3380541

scalability of distributed systems. Most prior research focuses on the right end of the spectrum [3, 21, 37, 38, 44], leaving alternatives in the design space unexplored. As pointed out by Harris [22], 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 read-path, while it exploits message passing to ease synchronization on update-path.

We first have a general discussion about the overhead and challenge when sharing data-structures on top of incoherence cache in §2. After discussing the design and implementation of BI in §3, we apply BI to a modified `PowerGraph` (§4) and compare against distributed implementations. The evaluation in §5 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:

- the design and implementation of bounded incoherence as a means for reasoning about data-structure modification in non-CC architectures,
- the application of BI to the `PowerGraph` system, and
- the evaluation of BI versus distributed approaches to cache coherency management.

2 Motivation

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. Our design is applicable to other models, such as DRAM serving as a cache to NVM. We focus on creating abstractions to handle non-CC memory, instead of on its non-volatility.

2.1 Cache Operation Overheads

There are three typical cache operations: (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 2 reports the per-cache-line overhead for a number of different memory and cache operations. Results are from HPE Superdome Flex server (more hardware details can be found in §5).

The `data-structure read` and `flush+read` lines in Figure 2 represent random-accesses to cache-lines of a given working set. This represents the cost of accessing a simple data-structure using only reads (loads), 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 bounded incoherence to enable cache-speed access to shared data-structures.

The lines for `clflush` demonstrate the cost of flushing a random sequence of cache-lines after either reading the data into cache (r), modifying it (m), or invalidating it (i). This shows the cost of flushing cache-lines in different states in the cache. Alternatively,

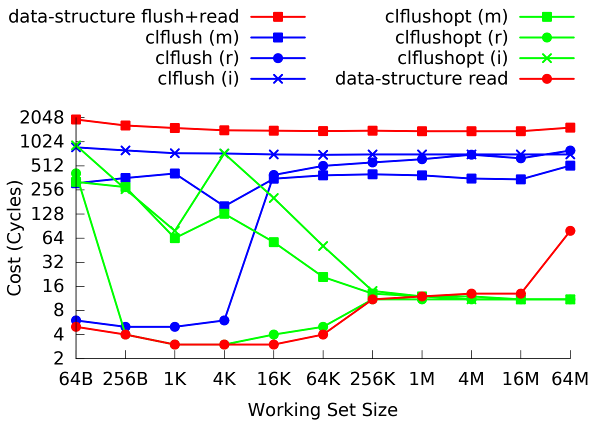

Figure 2. Cache operation per-cache-line overheads.

the `clflushopt` instruction, available on modern x86 processors, enables micro-architectural pipelining of the flush instructions. All accesses are serialized after the flushes with a memory barrier (via `mfence`). For 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. There is more overhead for cache-lines that are modified and require write-backs, than if they are only read or not present in the cache. 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.

2.2 Cache-Incoherence & Data-structure Consistency

To investigate the impact of incoherent memory accesses on the consistency properties of a data-structure, Figure 3 depicts a simple linked list data-structure undergoing modification with non-CC caches. We use a *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 inconsistencies arise.

- *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.
- *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.
- *Dangling references:* in (c), stale cache-lines still reference a removed and `freed` object.
- *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 of the resource* can be different

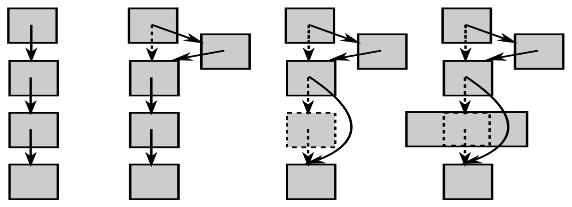

Figure 3. A singly-linked list going through a sequence of modifications with non-CC memory. Each list is a configuration after a modification. Dashed lines and boxes represent stale cache contents for that object, while dark continuous lines denote the state in memory. (a) is the initial configuration, (b) adds an object, leaving a stale link in some node's caches, (c) removes and `free`s 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).

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 [39]. In the absence of a mutually exclusive abstraction over all data-structure modifications and access (*e.g.* locks), 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 [14, 23, 34, 47] to avoid re-allocating objects until there are no parallel threads possibly accessing them.

Conclusion #3: 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.

3 BI Design and Implementation

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. BI also focuses on general applicability and adheres to classic RCU API, allowing it to be employed in, from lowest-level system component, such as kernel, up to higher-level data center applications.

3.1 The Bounded Incoherence Consistency Model

To provide the benefits of cached read-path access, and to avoid many of the consistency problems from §2.2, BI is designed to have a number of properties:

P1 Cached object access is used for all read-paths, thus eliding expensive cache-line invalidation operations.

P2 Cache-lines are stale only for at most a bounded amount of time.

P3 The memory backing stale freed data-structure objects will not be reused while any cache references to them *can* exist. 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 in different nodes. For example, sequential consistency [30] 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:

- *Stores* of one node are visible to other node's loads in at most a *bounded amount of time.* Due to cached-access, loads don't immediately observe another node's store.
- *Stores* are made directly to memory and those made to a single address are seen in sequential order.
- *Stores* to different addresses can be reordered on another node. Loads of different cache-lines can subsequently observe cached, then uncached data, which can reverse the order of stores.

3.2 BI API

The BI API mainly inherits from RCU [14], but extends it to explicitly manage cache coherence. On the reader side, BI use the same API as RCU to access shared data-structures.

- `bi_enter()` to declare the start of a code section in which references to objects can exist. Its usage is the same as `rcu_read_lock()` in RCU.
- `bi_exit()` to declare the end of that section, same as the RCU counterpart, `rcu_read_unlock()`. No thread-local references to objects can remain after this.
- `bi_dereference(void *)` to fetch a shared pointer which can be safely dereferenced. It also executes any needed memory barrier.

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

- `bi_free_object_quiescent()` flushes local cache to drop any reference to freed objects. While a thread does not hold any references to objects after `bi_read_unlock()`, those references can still exist in the processor's local cache. Hence, a reader needs to call this API properly to flush its cache.
- `bi_stale_object_quiescent(call_back_fn)` to invalidate stale cache lines to get their updated value. While BI provides built-in support to track modified objects (§3.4), it also allows users to pass in a call back function. The call back function enables applications to flush any necessary cache lines in their specific way.

The BI runtime invokes above quiescent APIs periodically on every reader core to achieve cache quiescence. However, applications can also explicitly call them, which provides self-managed, on-demand cache quiescence (§3.5).

On the writer side, BI needs more APIs to manage cache and coordinate concurrent readers.

- `synchronize_bi()` detects an elapsed grace period. It differs from `synchronize_rcu()` in two ways. First, it does not block waiting for quiescence. Instead it calculates the most recent time

```
 1 void writer(D,  identifier ) {
 2     n = bi_alloc(size, flag );      //  allocate  a  new node
 3     bi_enter();
 4     p = lookup(D,  identifier );    //  find  existing  node
 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_relcaim();                   //  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  nodes (e)
22     bi_free_object_quiescent();
23     // drop  stale  cache of modified memory (p)
24     bi_stale_object_quiescent();
25 }
```

Figure 4. Typical usage of BI API with a shared data-structure D.

in the past when quiescence was achieved. Second, in addition to checking if readers exit read-side section, it also checks if necessary cache flushes are performed.

- `bi_assign_pointer(void *, value)` to assign 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.

To integrate memory and cache coherency management, BI provides an extra set memory operations. BI runtime invokes above quiescent APIs periodically on every reader core to achieve cache quiescence. However, applications can also explicitly call them, which provides self-managed, on-demand cache quiescence (§3.5).

On the writer side, BI needs more APIs to manage cache and coordinate concurrent readers.

- `bi_alloc(size, flag)` which allocates memory for use within the data-structure. flag indicates if BI tracks modifications to the allocated memory.
- `bi_free(void *)` to deallocate the object without actually freeing up its memory.
- `bi_reclaim()` to reclaim and free previously deallocated objects. It uses `synchronize_bi()` to make sure all reclaimed objects can be safely reused.

Figure 4 depicts example pseudocode illustrating the use of BI to operate a shared data-structure. A reader (writer) finds the node 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 node (line 2), then performs the update by copying (a part of) the old node (line 5), updating the copy (line 6), and replacing the old version with the new one (line 7). The old node is marked to be freed later (line 8). The reclamation of freed nodes (line 12) happens after quiescence detection (line 11), which contains the logic to ensure a quiescence period has elapsed. On the reader side, extra care needs to be taken to deal with incoherent cache. After `bi_exit()` (line 19), while readers do not hold any references to freed node, stale cache lines could still contain those reference. Thus the reader has to flush stale lines of freed nodes (line 22). Furthermore, as the reference itself is modified by a writer, the reader needs to invalidate that cache 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 introduce staleness before then as a result. Therefore, it is essential for BI to guarantee the staleness is sustained within at most a bounded amount of time.

3.3 Data-structure Semantics for BI

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

Data-structure lookups with BI. Lookups exclusively use loads on objects. Modifications to these objects from 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 that assumes transparent cache evictions.

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 [1, 10, 32]. To cope with such difficulty, BI supports two mechanisms:

- *Mutually exclusive writers.* Mutual exclusion alone is not sufficient, and must be paired with explicit cache-line writeback and invalidation operations.
- *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. This was 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. The wide-spread use of RCU in the Linux kernel [14, 32] demonstrates that there are data-structures with relaxed consistency requirements. The question is if the additional requirements of BI prohibit interesting applications.

3.4 Bounded Incoherence Runtime

We implement BI prototype as a run-time library off of ParSec [47] which provides a slab allocator, Scalable Memory Reclamation (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 §3.2. ParSec uses invariant TSC [24] support to compare different core's accesses with the case when memory was freed, to determine if it can be reused. This implementation minimizes loads and stores to SMR variables for other cores, thus enabling stronger scalability compared to U-RCU [47].

The use of synchronized time stamp counters in ParSec is convenient, and scalable. Unfortunately, in rack-scale systems, such architectural support cannot be assumed as nodes are more loosely coupled than cores on sockets. §3.5 details how BI extends ParSec to use a *logical clock* that is monotonically increasing at a frequency related to the periodicity of data-structure cache-line invalidation.

When a node `free`s an object, its header is annotated with the current logical time, it is tracked on a quiescence queue, and it is only removed and re-used once the logical clock increases to the point where the object cannot be in a stale cache-line (nor referenced) on *any* node. This delay effectively converts the cache-incoherence problems around dangling references and type inconsistencies, into an SMR problem that can prevent type inconsistencies. Quiescence queues are ordered by logical clock value which enables a batched reclamation of multiple objects for a single computed quiescence value (Q). To track modified objects, BI uses another set of modification queues, which are similar to above quiescence queues. When an object is updated, it is put into the modification queue along with the current logical time. It will be removed only after the BI runtime determines that all cores have dropped their old cache lines, and are able to see the updated value. All above queues are per core and are implemented as a ring buffer instead of as a linked list in ParSec.

3.5 Cache Quiescence

Cache quiescence ensures no stale cache-lines exist at the point of quiescence. Global quiescence is achieved by invalidating stale cache-lines on each node. Any resources or objects deactivated *before* that time may then be reused. To determine this ordering, each node has to have access to the time when each other node has flushed stale entries from their cache.

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. The logical time when each node invalidates its data-structure cache-lines is recorded. Unfortunately, this requires frequent modifications to the shared logical clock which not only involves memory-scale latency, but also contends 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, and only when the node is quiescent.

The BI runtime assumes that each node has an accurate source of time such as a periodic timer, or a steadily increasing cycle counter. Each node maintains a logical clock (Q_i for node i) that it updates each time it performs quiescence. When a node needs to calculate what time all nodes in the system have quiesced ($Q = \min_{\forall i}(Q_i)$), it simply iterates through each node's logical times. Thus memory freed at time $t < Q_i$ can only be reused if no core can have it in stale cache-lines, thus if $t < Q$. However, it is possible that when computing if $t < Q$, that Q is pessimistic as it is based on reading the last cache-invalidation period for a node, Q_j where the cache-line holding that logical time itself is stale. This stale logical clock value is updated by the periodic flushes on the local core, thus this might delay memory reclamation by a logical tick, $Q = \min_{\forall i}(Q_i) - 1$.

Achieving quiescence. 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-structure. When the working set is not large, invalidating only accessed cache lines efficiently avoids unnecessary flush of unused memory. Invalidating modified objects works best in the case of read-heavy workload.

Currently BI provides built-in support to periodically invalidate cache lines of freed objects and modified objects tracked by BI. Despite this built-in support, BI also provides on-demand cache quiescence to allow applications to employ their specific object tracking and cache quiescence policy. To utilize the on-demand cache quiescence, an application marks its objects as self-managed during their memory allocation. During BI quiescence detection, it passes in a call back function, which iterates and flushes its managed objects. Furthermore, an application can also invoke BI quiescence regardless of BI built-in quiescence period. §4 presents how to utilize both periodic and on-demand BI cache quiescence in a graph processing framework.

Maintaining and using logical time. In our prototype, we use the cycle counter (via `rdtsc`) on each node to calculate the logical time. BI assumes that nodes, though not tightly coupled, can maintain logical times within an error margin of a single logical tick. This means quiescence calculations must assume a logical tick offset between nodes: $Q = \min_{\forall i}(Q_i) - 2$.

This design enables the efficient, cached access to all node's logical clocks. Updating a node's own logical clock requires only

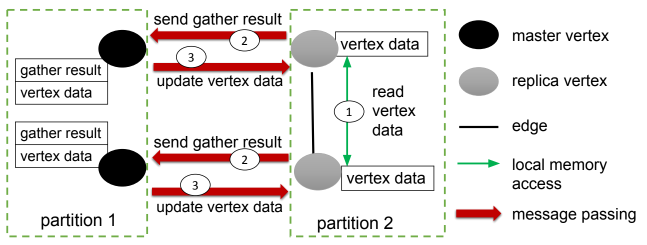

Figure 5. The communication pattern in the original PowerGraph. During the gather phase, a vertex reads adjacent vertices data locally (①). Then each replica sends gather result to the master via message passing (②). The master sends message with the updated vertex data to its all replicas (③).

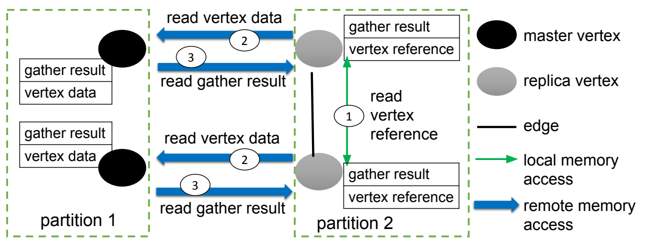

Figure 6. The communication pattern in the BI enabled PowerGraph. For every replica, BI replaces the actual vertex data with a reference to its master, and save the gather result locally. During the gather phase, a vertex first gets the reference to adjacent vertices master (①)), then fetches the actual data from remote master (②). After replicas calculate and save gather results locally, the master collects gather results from its remote replicas (③), and update the vertex data accordingly. The master no longer needs to update remote replicas thenceforth.

writing the cache-line back to memory, and does not require atomic memory operations (*e.g.* a write-back via clwb is sufficient). The frequency of cache quiescence on each node 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 batches cache flush operations, but provide a coarser granularity of quiescence. Determining the best frequency is out of scope for this work.

4 BI Use Case: PowerGraph

4.1 PowerGraph Background

PowerGraph [19] 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 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.

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, the rest are read-only replicas. While vertex data can be retrieved locally from 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 the result using the same sum function. The final combined result is passed to apply phase.

After the gather phase has completed, the apply phase is invoked only on 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 5 shows the communication among replicas and masters within each phase.

4.2 Challenges with non-CC memory

With non-CC memory, there are a number of complications to PowerGraph design. The communication across replicas makes no use of shared-memory and it exposes message passing overhead. At scale, these message passing can prohibit effective use of an increasing number of cores. The potential large amount of memory used by PowerGraph also complicates the cache quiescence, which might require a huge amount of cache invalidation. 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 cache coherence domains. 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. Edge data is totally local, therefore we only need to focus on maintenance of vertex data.

4.3 BI PowerGraph Implementation

We port PowerGraph to BI based on the open source GraphLab C++ implementation[1]. 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 run-time 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

[1]https://github.com/jegonzal/PowerGraph

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

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 replicas data-structures to read their results and combine them to get the final result. Consequently, all modifications in this phase are pure local. Only masters need to read remote cache lines, which are made visible by 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. BI runtime guarantees that 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 6 shows the details of the communication among replicas and masters in BI enabled `PowerGraph`.

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 updated vertex data. In both cases, we choose to invalidate only *modified* cache lines as `PowerGraph` potentially access a huge amount of unmodified memory.

Cache quiescence is implemented in two ways. First, we mark gather result and replica structure as BI manageable. 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 invalidation on one core will flush all other cores within the same socket. 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. While this implementation requires more changes to the application, it greatly reduces the memory usage by eliminating BI quiescence queues. It also achieves less staleness which is discussed next.

Staleness Analysis. Staleness is introduced by BI periodic cache flush. 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 previous iteration. However, such staleness is bounded by BI quiescence period. As studied in previous research [12, 45], a large class of iterative graph and machine learning algorithms are proved to be converged even in the face of staleness between iterations, as long as such staleness is restricted within a limited amount of time.

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. While not implemented in this work, we can also trigger cache quiescence only within specific iterations to achieve the A-BSP or SSP model [12];

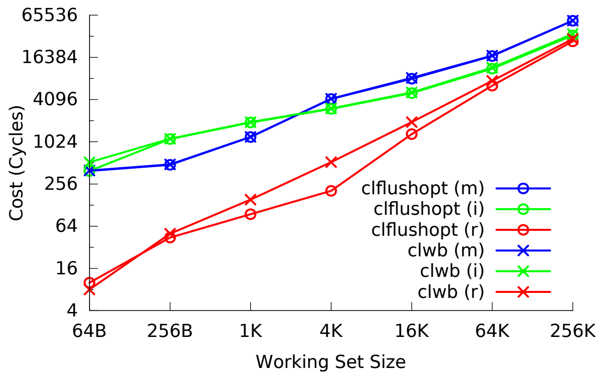

Figure 7. `clwb` and `clflushopt` overhead.

5 Evaluation

Experiment Platform. All experiments are run on HPE Superdome Flex servers[2]. We deploy two enclosures, with four 28-core sockets per enclosure. 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. Cache coherency domain is at socket level. That is to say, when cache coherency is disabled, only cores inside the same socket are coherent, cache between different sockets (even within the same enclosure) is non-coherent. Each enclosure runs an independent copy of the SLES-15 operating system, with Linux 4.12.14 kernel.

5.1 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 7 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 under different cache-line status.

Those overheads are linear with the working set size. With large working set, the memory latency dominates the cache overhead, so all of their cost have negligible difference, and we only show results from working set smaller than 256KB. With small working set, the cost of both instructions are observable but not prohibitive. More important, on the non-coherent architecture, such overhead does not increase with the scale of the machine, as their impact is limited only to its own socket. In general, `clwb` has a little less overhead than `clflushopt`, as it does not invalidate operated cache lines. Furthermore, this avoids cache miss of following memory access. The cache line status has a bigger impact on the cache operation

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

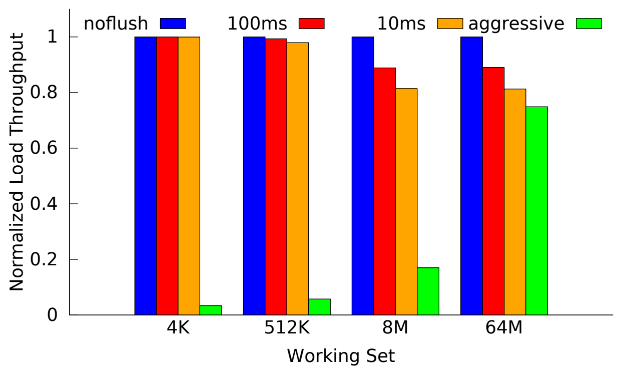

Figure 8. load throughput on a single core.

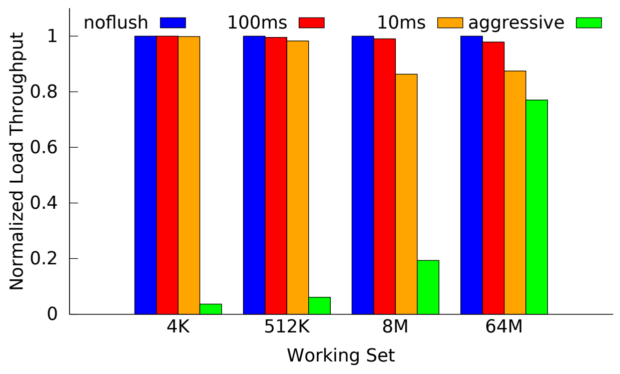

Figure 9. load throughput on two cores.

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 other cores in the coherency domain (one socket in this case), causing some overheads.

5.2 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 8 depicts the load throughput while cache invalidation is executed on the same core as the test. Figure 9 reports results of tests running on a different core. All throughput 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 larger working set, cache invalidation

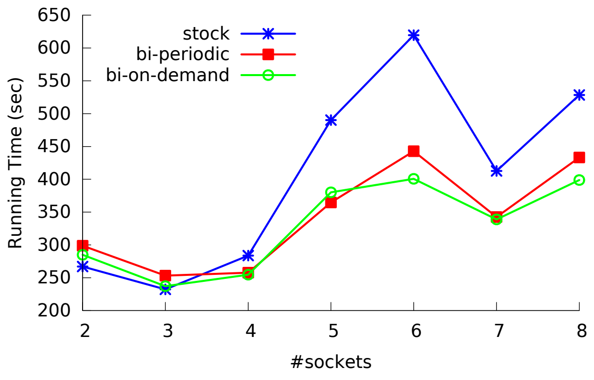

Figure 10. pagerank on twitter graph.

has more impact as expected, due to the increased cache miss overhead. For example, when working set fits into L2 cache, periodic cache invalidation at most introduce 2% performance degradation. When 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 9, 100ms invalidation period has 2% throughput degradation, while 10ms period has 10% degradation.

5.3 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 §4.3.

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 message passing. (2) BI`-on-demand` – cache quiescence is invoked by `PowerGraph` inside BSP barrier after each phase. (3) BI`-periodic` – data coherency is handled by BI periodic cache quiescence.

Experiment Set-up. To characterize the performance, we measure the total running time of PageRank algorithm provided by `PowerGraph`. PageRank runs on Twitter follower graph [29], 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 10 reports the running time with different number of sockets. All sockets are evenly assigned to two enclosures. With small number of sockets, BI variants run slightly slower than the original `PowerGraph`. Fox 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 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.

6 Related Work

Scalable memory reclamation. BI borrows heavily from SMR techniques such as epoch-based reclamation [23], RCU [14], ParSec [47] and IBR [48]. 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 that references are possible to remain in stale cache. 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 [7] 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 [26, 42, 43]. Some research [12, 45] explicitly relaxed data consistency and introduce data staleness in distributed systems. Bounded staleness is exploited by [12] to accelerate big data analytics, where the algorithm can see old data from previous iterations. Lazygraph [45] 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 [18], in some cases by relaxing consistency [13]. Some systems treat a single system as one that is distributed [4, 8, 20], and use message-passing-based coordination [3]. Message passing is traditionally used to implement distributed shared memory [6, 26–28, 35, 42] and provide partitioned global address space (PGAS) abstraction [9, 11]. Grappa [35] distributes computation across a cluster with an optimized PGAS implementation. Argo [27], a software distributed shared memory system, distributes coherence decisions using self-invalidation and self-downgrade combined with hierarchical queue delegation locks. Hare [21] uses message passing to implement a distributed file-system across nodes in a non-CC system. libMPNode [31] 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 Read-Copy-Update (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 [14, 47], though some techniques use fine-grained locking [1, 10, 32].

GAM [5] provides a directory-based cache coherence protocol over RDMA. Systems such as FaRM and RackOut [15, 16, 36] treat a cluster as a non-CC NUMA machine with RDMA-accessible remote memory. They use similar techniques (*e.g.* epoch-based memory reclamation [23]), but don't support cached-access to remote memory. In contrast, BI enables the cache-based access to global structures on rack-scale systems.

7 Future Work

There is much room for future research with non-coherent memory architectures and BI. After current BI prototype, we aim to improve it and apply it to more application domains.

BI optimization. There are lots of opportunities to improve BI performance, especially optimizing its cache quiescence frequency and cache invalidation. First, we seek to expose more control of cache quiescence frequency to users, or even better, to change the frequency adaptively at runtime. Second, we could run cache invalidation more intelligently based on the working set and access pattern, as discussed in §3.5. Last, we can exploit and integrate more application specific information to provide cache consistency. With the help of application itself, we are able to identify stale cache more accurately while reducing the BI tracking overhead.

BI application. We are working on to apply BI to board application domains, including more graph analytic algorithms, machine learning platforms and key value stores. Especially, quiescence-based techniques are successfully applied to operating system kernels [10, 32, 33, 46]. We envision BI can also be integrated into kernels to enable a single OS image to consistently manage all memory between incoherent nodes.

8 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 the anonymous reviewers for their feedback that has significantly improved the quality of this paper. We also thank Brad Tanner, Rocky Craig, Bill Hayes, Michael Woodacre, Keith Packard, Paolo Faraboschi and Robert Peter Haddad for their enormous help.

References

[1] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14).*

[2] Krste Asanovic. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14).* Santa Clara, CA, USA.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A new OS architecture for scalable multicore systems. In *Symposium on Operating System Principles (SOSP).*

[4] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. Disco: running commodity operating systems on scalable multiprocessors. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 143–156. https://doi.org/10.1145/268998.266672

[5] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1604–1617.

[6] J. B. Carter and W. Zwaenepoel. 1990. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming.*

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).*

[8] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. 1995. Hive: fault containment for shared-memory multiprocessors. *SIGOPS Operating Systems Review* 29, 5 (1995), 12–25.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05).*

[10] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference (EuroSys 2013)*. Prague, Czech Republic.

[11] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05).*

[12] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA, 37–48.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07), Stevenson, Washington, USA, October 14-17.*

[14] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012).

[15] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14), Seattle, WA, USA, April 2-4.*

[16] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15), Monterey, CA, USA, October 4-7.*

[17] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause, Ittingen, Switzerland, May 18-20.*

[18] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable Consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11), Cascais, Portugal, October 23-26.*

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA.

[20] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. 1999. Cellular Disco: Resource Management Using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP'99), Kiawah Island Resort, South Carolina, USA, December 12-15.*

[21] Charles Gruenwald, III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Hare: A File System for Non-cache-coherent Multicores. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys '15).*

[22] Tim Harris. 2015. Hardware Trends: Challenges and Opportunities in Distributed Computing. *ACM SIGACT News* 46, 2 (2015), 89–95.

[23] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (2007).

[24] Intel Corporation [n. d.]. *Intel-64 and IA-32 architectures software developer's manual, Volume 3A: System Programming Guide, Part 1*. Intel Corporation.

[25] Intel Corporation. 2016. Intel Rack Scale Design. Online. http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-architecture/intel-rack-scale-architecture-resources.html.

[26] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. 1995. CRL: High-performance All-software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95), Copper Mountain Resort, Colorado, USA, December 3-6.*

[27] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15).*

[28] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter 1994 Technical Conference, San Francisco, California, January 17-21.*

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10).*

[30] Leslie. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979).

[31] Robert Lyerly, Sang-Hoon Kim, and Binoy Ravindran. 2019. libMPNode: An OpenMP Runtime For Parallel Processing Across Incoherent Domains. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19).*

[32] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15).*

[33] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. 2013. RCU usage in the linux kernel: One decade later. *Technical report* (2013).

[34] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems* (2004).

[35] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA.

[36] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16), Santa Clara, CA, USA, October 5-7.*

[37] Simon Peter, Jana Giceva, Pravin Shinde, Gustavo Alonso, and Timothy Roscoe. 2011. POSTER: OS design for non-cache-coherent systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11), Cascais, Portugal, October 23-26.*

[38] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. 2011. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer.. In *Proceedings of the 3rd Many-core Applications Research Community Symposium (MARC), Ettlingen, Germany, July 5-6.*

[39] S. Prakash, Yann Hang Lee, and T. Johnson. 1994. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Trans. Comput.* (1994).

[40] Aravinda Prasad and K. Gopinath. 2016. Prudent Memory Reclamation in Procrastination-Based Synchronization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16), Atlanta, GA, USA, April 2-6.*

[41] Yuxin Ren, Liu Guyue, Gabriel Parmer, and Björn Brandenburg. 2018. Scalable Memory Reclamation for Multi-Core, Real-Time Systems. In *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).*

[42] Daniel J. Scales and Kourosh Gharachorloo. 1997. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97), St. Malo, France, October 5-8.*

[43] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97), St. Malo, France, October 5-8.*

[44] Rob F Van der Wijngaart, Timothy G Mattson, and Werner Haas. 2011. Lightweight communications on Intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review* 45, 1 (2011), 73–83.

[45] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. 2018. Lazygraph: Lazy Data Coherency for Replicas in Distributed Graph-parallel Computation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18).*

[46] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. 2015. Speck: A Kernel for Scalable Predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).*

[47] Qi Wang, Tim Stamler, and Gabriel Parmer. 2016. Parallel Sections: Scaling System-Level Data-Structures. In *Proceedings of the ACM EuroSys Conference.*

[48] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-Based Memory Reclamation. (2018).