Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory

Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros†, Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden
†Computer Engineering Department, University of Murcia, Spain

ABSTRACT

A coherent global address space in a distributed system enables shared memory programming in a much larger scale than a single multicore or a single SMP. Without dedicated hardware support at this scale, the solution is a software distributed shared memory (DSM) system. However, traditional approaches to coherence (*centralized via "active" home-node directories*) and critical-section execution (*distributed across nodes and cores*) are inherently unfit for such a scenario. Instead, it is crucial to make decisions locally and avoid the long latencies imposed by both network and software message handlers. Likewise, synchronization is fast if it rarely involves communication with distant nodes (or even other sockets). To minimize the amount of long-latency communication required in both coherence and critical section execution, we propose a DSM system with a novel coherence protocol, and a novel hierarchical queue delegation locking approach. More specifically, we propose an approach, suitable for data-race-free (DRF) programs, based on self-invalidation, self-downgrade, and passive data classification directories that require *no message handlers*, thereby incurring no extra latency. For fast synchronization we extend Queue Delegation Locking to execute critical sections in large batches on a single core before passing execution along to other cores, sockets, or nodes, in that hierarchical order. The result is a software DSM system called *Argo* which *localizes* as many decisions as possible and allows high parallel performance with little overhead on synchronization when compared to prior DSM implementations.

1. INTRODUCTION

Regardless of scale, the prevailing architectural paradigm to build shared memory parallel computers is intrinsically distributed: CPUs are tightly coupled with local memory in which copies of the data in global memory are cached and manipulated. This paradigm spans the spectrum from the ubiquitous multicore (where cores are serviced by their private caches which are further serviced by a proxy of the main memory in the form of a large shared last level cache), to the other end of systems consisting of networked nodes (processors+memory) where shared memory is implemented as a software layer that distributes virtual memory across the nodes (distributed shared virtual-memory or simply DSM). Hardware DSM can be made to scale well, but at a considerable cost [32], and thus it is not the dominating architecture in large installations. In this work, we re-examine the case of the software distributed shared memory.

We believe that the way the prevailing shared memory architectural paradigm is implemented instills it with two fundamental flaws that restrict its scale. The first is that coherence control of *shared* data is *centralized* to a single point in the system, to a directory entry responsible for keeping coherence for these data. The second is that inherently *serial execution*—in particular critical sections using the same lock—is *distributed* across the compute nodes, forcing unnecessary movement of data to where the execution takes place (migratory sharing [41]). Considering that coherence control of the migratory data is centralized (as per our previous point), distributing the serialized execution of critical sections can only worsen performance and, according to Amdahl's law, this effect will be detrimental to scalability.

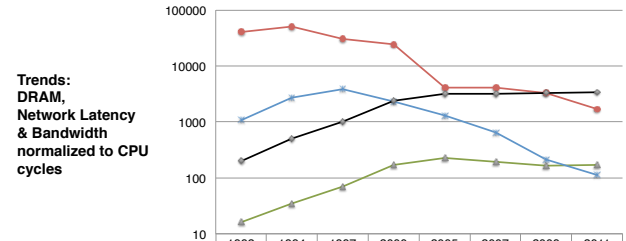
These two flaws can be safely ignored when the latencies involved are small. Thus, it is feasible to build an efficient (single-chip) multicore using *hardware* directory coherence and distributed critical section execution, which becomes only moderately worse with the increased latencies found in a (single-node) multi-socket board. Industry examples abound. However, the situation changes dramatically in a software DSM system consisting of commodity networked nodes, because of three reasons: i) we are faced with larger network latencies, ii) software message handlers introduce further latency on every coherence action, and iii) because of the increased latencies, critical-section synchronization becomes a serious bottleneck.

Why now? Historically, when the first DSM systems appeared both network latency and network bandwidth were orders of magnitude worse than the corresponding main memory latency and bandwidth. In that setting, the overhead of message handler execution was relatively small (compared to the network latencies) making software coherence possible. On the other hand, coherence protocols were burdened to minimize both latency (e.g., hiding latency with relaxed consistency models [17]) and bandwidth (e.g., by minimizing data transfer using diffs).

Advances in network technology over the last two decades closed the latency and bandwidth gap with exponential im-

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

ACM 978-1-4503-3550-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2749246.2749250.



	1992	1994	1997	2000	2005	2007	2009	2011
DRAM min Latency (CPU cycles)	16	35	70	168	224	192	165	170
Net. max BW as CPU cycles/KB	1092	2731	3901	2313	1311	655	211	111
Network min Latency (CPU cycles)	40000	50000	30000	24000	4160	4160	3300	1700
CPU speed (MHz)	200	500	1000	2400	3200	3200	3300	3400

Figure 1: Trends for network bandwidth and latency normalized to CPU speed. (Adapted from Ramesh's thesis [35].)

provements. Figure 1 plots the trends for the top CPU speed (in MHz), DRAM latency, network latency, and the *inverse* of network bandwidth, all normalized to CPU cycles. The last metric is interesting with respect to a DSM implementation because it gives the latency in CPU cycles to transfer a block of data (e.g., 1 Kbyte). In the past, despite the rapid growth in network bandwidth, processor speed increased even faster. Consequently, *increasingly more* CPU cycles were needed to transfer a block of data with each passing year! This trend reversed as network bandwidth continued to improve rapidly but CPU frequency improvement stalled. Whereas the evolvement of network bandwidth with respect to CPU speed was a *deterrent* for DSM development in the past, it now is an incentive. On the other hand, whereas in the early days a DSM was easily realizable with software message handlers, this is not the case today. As Figure 1 shows, in terms of CPU cycles, network latency improved from being tens of thousands of CPU cycles (easily overshadowing any message handler latency) to just a couple of thousand CPU cycles (just an order of magnitude worse than DRAM latency). These technology trends point to the tradeoffs that we should make today to scale shared memory to a distributed system: i) trade using more bandwidth to reduce latency, ii) minimize message handler execution, and iii) minimize movement of data for *dependent* computation (e.g., critical section execution) as network latency is still an order of magnitude worse than memory latency.

Contributions. Motivated by these tradeoffs, we flip the way coherence and critical-section execution is done.

- Inspired by recent efforts to simplify coherence in multicores [37, 15], we use self-invalidation (no invalidations and no write-misses) and self-downgrade (no indirection via a home directory) to reduce the latency of servicing each individual miss. Effectively, we are *distributing* coherence control, so that coherence decisions (invalidation and downgrade) are made locally by each node without communication.
- To reduce the *number* of self-invalidation misses and consequently the *average access latency*, we introduce a novel home-node directory protocol *without* message handlers, where all operations are performed by RDMA reads and writes, initiated by the requesting core—no other node is involved in servicing a coherence miss. This approach increases network traffic, but in return we achieve a *message-handler-free* protocol. Further, we systematically trade bandwidth for latency, e.g., by prefetching and by avoiding to create and transmit diffs when we can.
- We consolidate hierarchically the execution of critical sections that use the same lock (to the core or node that happens to get the lock first) in order to minimize wasteful movement of migratory data. This dramatically reduces synchronization latency and critical-section overhead. We achieve this by extending a state-of-the-art queue delegation locking algorithm [18, 20]. To take full advantage of this approach some source code changes are necessary, but the required effort is modest.

The result is a highly scalable DSM system called *Argo*[1] with a novel coherence protocol (*Carina*) based on passive classification directories (*Pyxis*) and a new locking system (*Vela*).

Argo is a page-based, user-space, DSM and its prototype implementation is built on top of MPI. It runs unmodified Pthreads (data-race-free) shared memory programs and with some source code modifications, optimizes the locking performance of such programs with a novel hierarchical extension of queue delegation locking [18, 20].

We evaluate the prototype implementation with a set of seven benchmarks and characterize the effects of our design decisions. We show that Argo is able to scale the set of benchmarks to a large number of nodes, competing with message-passing or Partitioned Global Address Space implementations, as well as to sustain the performance level of critical section execution in a distributed environment.

2. BACKGROUND

2.1 Coherence

Of course, we are not the first to note distributed shared memory problems and aim to address them. A large body of prior work set the same goals. Regarding the problem of centralized coherence the line of attack has been to relax the memory consistency, notably going from Sequential Consistency (SC) [21] to weak memory models that rely on synchronization to enforce ordering. These memory models made it possible to relax the constraints on coherence and ameliorate the centralization problem. We call attention here to Release Consistency (RC) [10], Lazy Release Consistency [17], Entry Consistency [2], and Scope Consistency [13], all of which were invented to optimize the coherence of distributed shared memory architectures: a hardware architecture for RC—the DASH prototype [23] built at Stanford—and software distributed shared virtual memory systems for the rest [24, 2, 16]. However, none of these approaches arrived at a truly distributed coherence solution. Self-invalidation and self-downgrade (write-through) enables fully distributed coherence protocol (that theoretically requires no directory). Practically, however, it is not enough. There is a steep price to pay for excessive self-invalidation. One of the contributions of our work is to address this deficiency by introducing a directory to classify pages according to the number of sharers and the number of writers on each. The novel aspect of our approach is that the directory is *passive*, i.e., it is only accessed by RDMA and does not introduce message handlers in the system.

PGAS/UPC. Using only RDMA to access shared data is comparable to Partitioned Global Address Space (PGAS)

[1] In ancient times Argo Navis (or simply Argo) was a large constellation in the southern sky that has since been divided into a number of smaller constellations: Carina (the Hull), Pyxis (the Compass), Vela (the Sail), among others.

approaches such as Unified Parallel C [40]. For this reason we will use UPC as a representative base case in our evaluation. In PGAS the address space is logically divided between processes but the main difference with our approach is that there is no remote caching. Shared data can of course be local to a node, but determining this at runtime is costly, and therefore programmers are advised to cast such pointers to local pointers where necessary. Remote accesses are fine grained, and, similarly to our approach, using a relaxed memory model [40] each access can be delayed until the next synchronization point, making it possible to hide the latency of remote accesses with computation on local data. Without caching, it is often necessary for UPC programmers to manually move data from the global address space to thread-local space in bulk transfers, which is then similar to message passing paradigms in the way that the programmer needs to think about communication patterns rather than thinking about how threads are exploiting locality. Critical sections in UPC are programmed with UPC-specific lock/unlock primitives, which ensures that all memory operations have been completed before both the lock and unlock call. Since critical sections by nature make most threads access and write non-local data they incur a high penalty in UPC, as each operation in the critical section is a remote operation without any possibility to run it locally. We address all these issues with our approach.

2.2 Synchronization

Over the years, many locking algorithms have been developed to speed up critical section execution. Traditionally, these algorithms focus on acquiring and releasing a lock as quickly as possible, while also considering the time required for handovers. Queue-based locks, like MCS [28] and CLH [5, 27], reduce cache coherence traffic to achieve higher performance on multicore systems. At the same time they order critical sections without taking the location of data into consideration, which results in performance degradation on NUMA systems, due to the higher latency and lower bandwidth between processors. On NUMA systems these locks are outperformed by algorithms that distribute work in a less fair manner, e.g. the HBO lock [34], the hierarchical CLH lock [26] or more recently the *Cohort lock* [7]. They all aim to hand over the lock to a thread that is "nearby" to exploit the faster transfer of data to some parts of the system.

Delegation locking is a different approach, which takes this idea to the extreme by sending operations to where the lock is held instead of transferring the lock and data to each thread. This allows data to stay in the caches of a helper thread for all cache levels, resulting in higher throughput. Algorithms enabling delegation locking include flat combining [12], remote core locking [25], CC- and H-Synch [8]. Additionally, some algorithms allow threads that delegate critical sections to detach the execution, continue with other work, and only wait for the critical sections' execution if and when needed. Of these, detached execution [31] does not perform as well as other delegation algorithms while Queue Delegation Locking (QDL) [18, 20] combines a fast delegation method with the ability to detach execution. It allows to either collect operations from the entire system concurrently, or (in the case of a NUMA system) restrict the collection to a single NUMA node [20]. This Hierarchical Queue Delegation Locking (HQDL) allows for faster delegation at the cost of less exploitable parallelism in the system.

In a related approach, Suleman *et al.* [39] propose consolidating the execution of critical sections (which is inherently serial) in one fat, fast, and power-hungry core, while distributing parallel execution on thin and power-efficient cores, in a heterogeneous multicore consisting of both types of cores. This addresses Amdahl's law by speeding up the serial part of the programs.

2.3 Other Forward-Looking Approaches

A new DSM system that has focused on latency tolerant programming is Grappa [30]. Similarly to our motivation, the authors recognize that latency is the critical issue to address, at the expense of increasing bandwidth demands. Grappa has, similarly to this paper, tackled the problem of executing critical sections in the global address space. They use another synchronization mechanism called flat-combining. Grappa implements a system where it is possible to send critical sections to the node where the data the critical section wants to access is residing in the global address space. However, Grappa is a new programming paradigm that focuses more narrowly on throughput computing with a vast number of threads. We concentrate on offering efficient shared memory that can run the large library of parallel algorithms that have been developed over the years.

3. THE ARGO SYSTEM

Similarly to other DSM systems [24, 2, 16, 33], Argo implements shared memory using the facilities provided by the virtual memory system. Argo is a user-level DSM, implemented entirely in user space on top of MPI. While a custom network layer tailored to Argo would likely offer greater performance, we opted for the portability and flexibility of MPI for the prototype implementation.

Argo works on page granularity (4KB) and sets up a shared virtual address space spanning all nodes. This address space is first initialized by each node, allocating the same range of virtual addresses using the `mmap` system call. These addresses are then available for allocation using our own allocator.

Argo is a home-based DSM where each virtual page is assigned a home node. Currently, virtual pages are interleaved across nodes so that for a system containing N nodes, node_0 serves the lower addresses of the global memory and node_{N-1} serves the higher addresses. While this is a simplistic approach, we note that more sophisticated data distribution schemes are orthogonal to what we describe here and are left for future work.

Each node caches remote pages in a local page cache. Coherence must be enforced among all the page caches in the system. Typically in prior work, a directory would be responsible to keep a home node's pages coherent. This means keeping track of the readers and the writers of every page and sending invalidation and downgrade messages when necessary. The most important optimization proposed for such DSM systems is to relax the memory model and allow consolidation of coherence traffic on synchronization points [17]. In many cases, this means that downgrading dirty data necessitates the creation and transmission of diffs [17].

Our aim is to eliminate the centralized bottleneck of directories and arrive at a genuinely distributed solution. The reason is twofold: First, any operation that involves a directory is costly because of the latencies involved in a DSM. This is especially true for invalidation and indirection (locating the last writer and obtaining the latest value of the

data via the directory) where multiple network hops may be needed. Second, directory operations that involve even simple state transitions or sending new messages, require that a message handler be invoked at the directory side to perform such operations. Potentially a handler is needed also on the receiving side of a directory message to take an action there.

Thus, prior DSM systems rely on directories and page caches that are *active agents* requiring their own computing resources to *process* (receive and respond to) messages. Whether these active agents are implemented as software message handlers that poll the network interface for coherence messages, or are invoked via interrupts, is of little consequence; they consume resources, introduce latency into every access, and can easily become bottlenecks and compromise scalability in the presence of hot spots. Given our analysis in the introduction, the improvement in network latency has increased the relative overhead of software message handlers. Instead, what we would ideally like to do is use remote memory accesses—RDMA—for all coherence actions without the need to execute *code* on the receiving sides. This is perhaps one of the most significant differences between Argo and prior DSM systems.

We achieve this goal by relying on self-invalidation and self-downgrade as the main mechanisms of our coherence. Self-invalidation means that any node is allowed to read any data block as long as it promises to self-invalidate this block before passing a synchronization point. Similarly, self-downgrade means that a node can freely write any data block without obtaining any permission from a directory, as long as it makes the write visible to all other nodes *before* crossing a synchronization point.

The significance of this approach in DSM is twofold. First, self-invalidation eliminates explicit invalidations on writes. This means that no sharers need to be tracked by a home-node directory. Second, self-downgrade eliminates directory indirection on read misses to find the latest version of the data; instead, the correct data are always found in the home node on any read miss. This means neither the writers need to be tracked in the directory, thereby eliminating the need for a directory altogether.

While on first sight this would seem ideal for a DSM, in reality self-invalidation can seriously degrade performance if applied without constraint. This is where our first contribution comes in. We propose a *classification* directory to manage self-invalidation. To the best of our knowledge, this is the first *passive* directory protocol, i.e., without *any* message handlers (active agents) but using exclusively RDMA initiated by requestors to perform all protocol actions. This approach enables a distributed and highly-scalable coherence solution, which we call *Carina*.

As with any protocol that is based on self-invalidation, e.g., [22, 14, 37, 4], our approach imposes a weak memory model in which Sequential Consistency (SC) for DRF programs can be guaranteed when synchronization is fully exposed to the system [1]. We discuss this in the following section.

3.1 Exposing Synchronization to Carina

We expose program synchronization to the coherence layer using "fences" in the application programs. Fences are implicit for synchronization primitives in the Argo synchronization libraries and no source code modification is needed for DRF programs using the Argo Pthreads or HQDL libraries. There are two fences available in Argo: an SI_fence for self-invalidation and an SD_fence for self-downgrade. On an SI_fence, all pages in a page cache are invalidated before the fence completes. On an SD_fence, all the writes of modified pages are made globally visible before the fence completes.

The memory model of Argo corresponds to a weak memory model if on every synchronization (lock, unlock, signal, wait, barrier, or even synchronization via spin loops and flags), we apply *both* fences. For Release Consistency [10], it is sufficient to use an SI_fence on acquire points and SD_fence on release points. In Section 4 we will show how we use these fences for the memory model of queue delegation locks.

3.2 Carina: Argo's Coherence

Unrestricted self-invalidation (and to a lesser extent unrestricted self-downgrade) can seriously degrade performance considering that: i) on each acquire point, all the data cached in a node are self-invalidated, even if they have not been modified; and ii) on each release point, all the writes must be reflected back to the home node, even if they are not subsequently read by another node, or even if further updates to the data would render exposing these writes to other nodes premature and unproductive. It is therefore important to reduce these costs. We achieve this by classifying pages at home-node classification directories [11, 6, 37].

Any node accessing a page at a home node simply *deposits* its ID (separately for reads and writes) in the directory. This directory information indicates to future accessing nodes whether:

- the page is *Private, P,* (only one node is accessing it) or *Shared, S,* (more than one node are accessing it)
- the page has *No Writers, NW*[2] or the page has a *Single Writer, SW,* (only one node writes it) or *Multiple Writers, MW,* (more than one node write the page).

Note that classification transitions (P→S, NW→SW, and SW→MW) happen at most once per page, i.e., they are rare. Also note that this information concerns the global address space (where potentially everything can be shared) and not the thread-private space which is not shared across nodes. Thus, the term "Private" is somewhat of a misnomer here, as it does not imply perpetual privacy, but temporary privacy for data declared as shared.

In our current implementation these two types of classifications are one-way and non-adaptive: from private to shared and from no-writers to a single-writer to multiple-writers. However, it is straightforward to extend the classification to adaptive (reverting back to private or no-writers/single-writer) using simple "decay" techniques [36], if need arises. For the workloads we examined, such adaptation was not deemed necessary and is left for future work.

Table 1 shows the types of classifications we can do. The first, S, represents the case where no classification is performed in the system and all pages self-invalidate and self-downgrade. The second, P/S, simply distinguishes between private and shared pages, while the third, P/S3, further distinguishes shared pages to NW, SW, and MW, depending on the number of writers. We use this data classification to filter self-invalidation, e.g., exclude private pages, or shared pages without any writer from self-invalidation. While we could also optimize self-downgrade, we actually chose to do the

[2]Equivalently: *Read-Only, RO*

Classification	State	SI	SD	Comment
S: No classification	S	✓	✓	All pages shared
P/S: Simple P/S classification	P	—	✓	SD to avoid P→S forced downgrade
	S	✓	✓	
P/S3: Full P/S and Writer classification	P	—	✓	SD to avoid P→S forced downgrade
	S, NW	—	✓	
	S, SW	— (✓)	✓	SW does not SI but other nodes do
	S, MW	✓	✓	

Table 1: Three classifications: S, P/S, P/S3. Argo uses the full P/S3 classification.

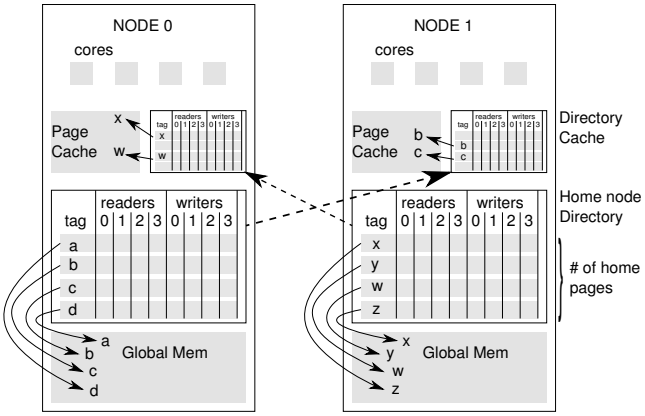


Figure 2: Directory organization (two nodes). Each node holds a "home" directory and a directory cache caching the remote directory entries.

opposite. For example, we chose to self-downgrade private pages. We do this to eliminate the need for message handlers to service classification transitions.

Our classification is in line with the philosophy we set forth in the introduction of trading bandwidth (increase traffic from self-downgrades) for latency (reducing self-invalidation and eliminating message handlers). Further, the distinction between SW and MW allows us to suppress the creation of diffs (latency) in the case of a single writer, since there can be no false sharing to corrupt data on downgrade, at the expense of transmitting more data (bandwidth). However, this optimization increases the complexity of the protocol and is left for future work.

3.3 Pyxis: Passive Classification Directory

But how do we implement a coherence protocol so that *no active agents—software message handlers—are needed* to respond to messages or generate new requests? In our approach the home node directories are simply metadata structures that are remotely read and written (RDMA) by the requesting nodes. Information is simply deposited in the directories for the *passive* classification of data and all protocol actions are performed by the requesting node.

Figure 2 shows the directory organization. The directory is simply a structure holding, for each page, its reader(s) and its writer(s). There is no explicit state for the page—this is inferred by the accessing nodes. We use a *full map* to keep track of the readers and writers.

When a node (read or write) misses in its page cache it accesses the directory for the page using a Fetch&Add atomic operation. This operation sets the node ID in the reader or writer full map (depending on the miss) and *returns the*

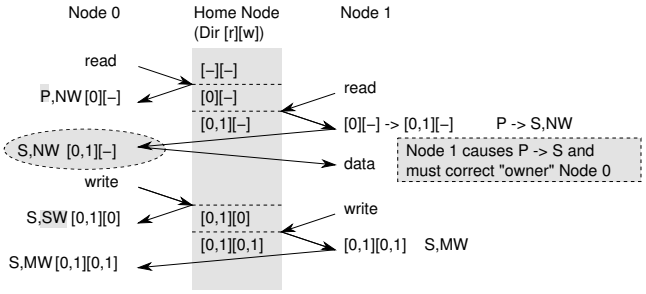


Figure 3: Overview of P→S and NW→SW→MW. The example shows "instant" transitions of state in node 0 (shaded) which would require a local active agent.

updated reader and writer full maps which are stored locally.

To facilitate notifying a node on a classification change without a message handler, we keep copies of all remote directory entries for all the pages in its page cache (Figure 2).[3]

A node is notified of a directory change by simply updating its local copy of the directory entry. The update is observed by the node on its next synchronization or its next request. I.e., a node's perception of the home node classification may be stale until its next synchronization point. This is allowed simply because of the DRF semantics.

3.4 P/S Classification

Let us see now how this classification can reduce the number of self-invalidations. Initially all pages start as uninitialized and without any readers or writers. Initialization writes to the pages do not count and this is a rudimentary form of adaptation that we implement: at the end of initialization and the start of the parallel section of the programs the reader and writer full maps are reset to null.

The first node accessing a page leaves its ID in the directory (as a reader). A single ID signifies that the page is Private as shown in Figure 3 for Node 0 when it first reads a page. The page does not need to be self-invalidated or self-downgraded as there is no sharing. When a new node (Node 1 in Figure 3) misses in its page cache and accesses the same page it observes that there is already a node (Node 0) that considers the page as private, the "private owner." The private owner needs to be notified that the classification has changed and the page is now shared, which means that:

1. the page should be self-invalidated at any subsequent synchronization point
2. any updates to this page previously performed by the private owner (Node 0) need to be made globally visible and certainly visible to the newcomer node (Node 1) before the latter continues.

Recall that the directory is passive, i.e., there is no directory agent to detect the classification change and appropriately notify the private owner. Recall also that the private owner is also "passive:" it can initiate requests but cannot respond to such. The burden for this falls on the node that causes the classification change (e.g., Node 1). Since this node updates the directory with its ID and in the process

[3]For simplicity we keep a *full* copy of any accessed remote directory. This is not a significant overhead in the systems we examined, and one can always "compress" the directory copies to a few bits per entry, if need be. Essentially, we want to know if a page has a single reader or single writer; if there is more than one reader or more than one writer, we do not care to know their identity.

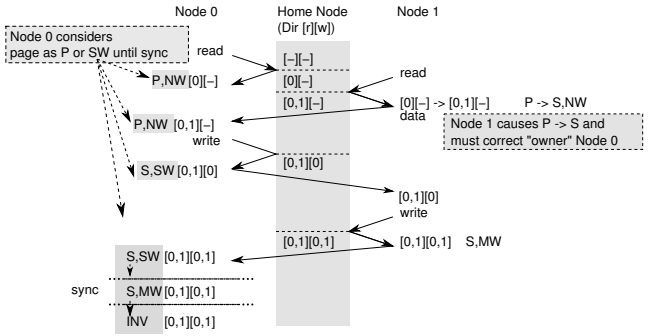


Figure 4: Deferred invalidation: Node 0 ignores P→S (and SW→MW) until its next synchronization.

receives the reader and writer full maps, it detects that the private owner needs to be notified. There is always just a single node that needs to be notified in the P→S transition. Since the private owner is known (from the reader full map), updating it is simply a remote update of its *cached copy* of the directory entry. *In essence, we use the cached directory entries in each node to reflect the updates in the home directory without the need to have active agents in the node.*

3.4.1 Deferred Invalidation

The private owner does not need to immediately notice the P→S transition as shown in Figure 3. There is no action taken at the time of the transition and this means that *no active agent is needed in Node 0.* The transition will be detected and action taken by Node 0 at its *next synchronization point* where it will include the shared page in the self-invalidation. We call this *deferred (self-)invalidation* and show it in Figure 4. In general, a node can consider a page "private" from the initial read to the next synchronization (Figure 4), and—as we already noted—this is valid simply because of the DRF semantics.

3.4.2 Self-Downgrading Private Pages

While deferred invalidation eliminates the need to have an agent to respond to a directory-cache update, the same unfortunately is not true for the second requirement of a P→S transition: obtaining (possible) private-owner modifications before continuing with the Node 1 access that causes the P→S transition in the first place. The problematic point is shown in Figure 3 marked with a grey oval. Normally, this would require an active agent in Node 0 to stall all local threads from accessing (potentially modifying) this page, until the home node and Node 1 are updated with the latest version of the data. Note that as per the DRF semantics Node 1 can only expect to see the writes of Node 0 at most up to the point of Node 0's last synchronization point. In addition, from the moment Node 0's data are made shared, any further writes must be self-downgraded at Node 0's *next* synchronization point as a *diff*. The last two points are critical in making the P→S transition work correctly.

Naïve Solution. A naïve solution would then be to checkpoint all modified private pages prior to any synchronization point and service P→S transitions from these checkpoints. As we will see in Section 5.1 creating these checkpoints is an expensive proposition as it significantly delays synchronization. The overhead largely invalidates the potential benefit of PS classification.

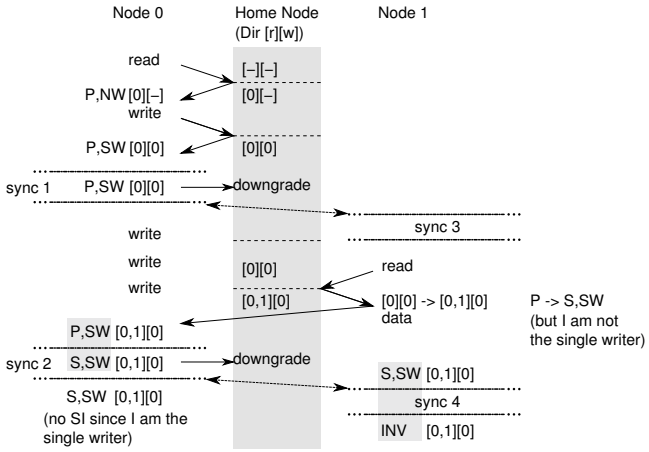


Figure 5: Self-downgrade of private pages.

Carina's Solution. To solve this conundrum, we trade-off bandwidth for latency. In this case, we *self-downgrade all Private pages at synchronization points as if they were Shared*! The latency benefit is twofold: 1) private pages are still excluded from self-invalidation (and do not need to be re-fetched as long as they remain private) and 2) we update private owners without active agents. In addition, checkpointing for diffs happens only on a write miss and its latency affects only the thread that incurs the write miss.

An example is shown in Figure 5. Node 0 is the private owner that starts to downgrade once it writes the page. At synchronization points all its modifications are put back to the home node. When Node 1 reads, it obtains the latest data directly from the home node. It does not matter that Node 0 keeps writing in parallel. As per the DRF semantics Node 1 is only guaranteed to "see" Node 0's modifications *prior* to `sync 1`.

3.5 P/S3 Classification

P/S3 classification adds writer classification (NW, SW, MW) on top of P/S classification. The distinction is meaningful for the shared, S, classification, since as we have explained the private, P, classification always self-downgrades and does not self-invalidate. The reason why it matters for the S classification is that it adds two more opportunities to eliminate self-invalidation: shared pages with no writers, and shared pages with a single writer. A comparison of P/S and P/S3 classification with respect to self-invalidation and self-downgrade is depicted schematically in Figure 6.

If there are no writers (yet) for a page, it can be considered as a read-only page and is classified as "No-Writer" (NW). Any node caching a page may go ahead and write it at any point without needing to obtain any kind of permission to do so. This is because of the DRF semantics: nodes will have no conflicting accesses (i.e., accesses not separated by synchronization). Therefore, any node can write its cached page copy at will, considering itself the single writer. If it is the first time a node writes a page, it deposits its ID at the directory and obtains the full map of writers to check whether there are others. We discern the following cases depending on the home-node state of the page:

- Private, and written by the private owner: In this case, the page becomes P,SW and starts to self-downgrade as described earlier (e.g., Figure 5 `sync 1`).

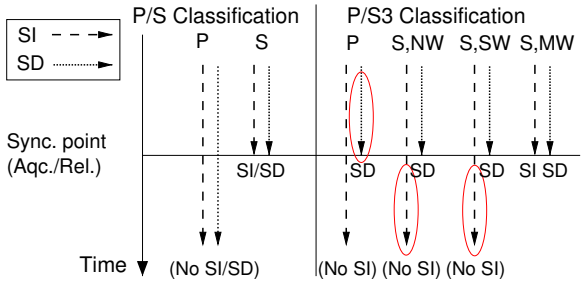


Figure 6: Trading bandwidth for latency. Self-invalidation (extra latency) is represented by dashed lines stopping at the synchronization point. Self-downgrade (extra bandwidth) is represented by dotted lines stopping at the synchronization point. Lines that cross the synchronization point do not require any action.

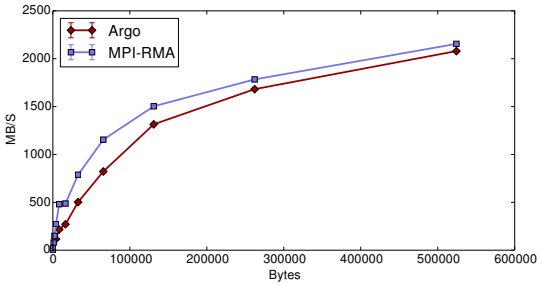


Figure 7: Bandwidth comparison: Argo cache-line read vs passive one-sided communication in OpenMPI 1.6.4. x-axis is cache line or MPI message size.

- Private, but written by a new node: The page transitions both to Shared and SW and the (single) private owner is notified accordingly.
- Shared, NW: The page is already shared by more than one node, and this is the first write (that reaches the home-node directory). In this case *all* nodes caching the page must be notified that there is a writer.
- Shared, SW: The page is already shared with a single writer, written now by a new node. *Only* the single writer is notified.
- Shared, MW: The page is shared with multiple writers. A new writer does not need to take any action.

A single writer node self-downgrades on synchronization. At downgrade, it checks its local directory cache to see if more writers have appeared (Figure 5 `sync 2`). If it is still the only writer then it does not need to self-invalidate—there are no other updates to this page! An example is shown in Figure 5 `sync 2`, where the single writer of a shared page can keep it after the synchronization. This is an optimization that works well in a producer-consumers scenario: the producer creates new data without invalidating its cached page on synchronization, and the consumers read the data directly from the home node—no indirection—as they invalidate their copies on their respective synchronization points (e.g., Figure 5 `sync 4`, the page is S,SW but Node 1 is *not* the single writer and must invalidate).

However, if more writers appear then the page transitions to Multiple Writers (MW) and *all* nodes must both self-invalidate and self-downgrade (if they write) this page. The transition SW→MW is exactly analogous to the P→S transition. As with the private owner there is initially only one "single writer" in the home-node directory and *only* this single writer needs to be notified (by the second writer) about the transition. No other node needs to be notified as it makes no difference: for all other nodes except the single-writer, the SW and MW states are equivalent and mean both self-invalidation and self-downgrade (if written). The transition of the single writer to MW is also *deferred*: it is discovered at the single writer's next synchronization point.

3.6 Additional Techniques

3.6.1 Write Buffers

Downgrading only on synchronization may cause considerable traffic (all the "dirty" pages need to be put downgraded to their corresponding home nodes) and thus is a serious performance bottleneck. To address this problem we use the equivalent of a write buffer that drains slowly (and downgrades pages) as it is filled with new dirty pages. When a page is marked as dirty we put in a FIFO write buffer of configurable size. Each time the buffer is full we simply write back the copy that was first entered in the buffer to its home node, then setting the memory protection for the virtual addresses mapping to the cache to read-only, meaning that any subsequent write to that memory will trigger a new write miss. Using a write buffer we ensure that all writes are propagated at a good pace to their home nodes, and guarantee an upper bound when flushing the write buffer on synchronization. Again this is an example of trading bandwidth for latency: a write buffer increases write–through traffic but limits the latency of SD_fence synchronization.

3.6.2 Prefetching

For underlying communications, our current implementation (prototype), uses passive one-sided communications of MPI. A limitation imposed by this choice is that only one thread can use the interconnect at any point in time to fetch a page. To mitigate this bottleneck and make better use of the available bandwidth, upon a cache miss Argo fetches not only the requested page but a "line" of consecutive pages. The cache line size and the number of cache lines provided are user configurable. The cache is direct mapped and maps each page to a fixed position in the cache using a simple mapping function. Subsequent threads causing a miss on a page that is being prefetched as part of a cache line (spatial locality) have to wait for that request to complete. Figure 7 shows the achievable bandwidth in Argo while reading a cache line of pages. Clearly, Argo tracks well the MPI data transfer rate with increased cache line size (x-axis).

4. EFFICIENT SYNCHRONIZATION

Efficient synchronization is of paramount importance in making Argo perform well, for two reasons: First, while Argo's coherence handles DRF accesses with ease, thread synchronization essentially constitutes a data race. Data races are problematic in a software DSM, especially in one that is based on self-invalidation and abolishes software message handlers to process (receive and respond to) messages. To implement data races one has to self-invalidate and self-downgrade every access involved in a race. Second, synchronization plays a vital role in Argo's Carina coherence since it determines the points where Carina self-invalidates or self-downgrades. Once synchronization is achieved via a

data race, Carina must self-invalidate and/or self-downgrade (depending on the acquire/release semantics of the achieved synchronization) all cached data. While Carina aims to minimize self-invalidation with the P/S3 classification, it is obvious that programs with intensive synchronization will suffer considerably more misses than a comparable DSM using an explicit invalidation protocol.

These two reasons require a complete re-think of synchronization for the Argo system. We discern two types of synchronization: barrier and lock-based synchronization. They are treated separately in Argo's Vela synchronization system that provides the necessary API for both (among other primitives such as signal/wait that are not discussed further due to the limited space in this paper). We will start with the barrier which is the easiest, and proceed with lock-synchronization which will be the main focus of this section.

4.1 Barrier Synchronization

One of the most prevalent synchronization mechanisms in distributed programming is the barrier. This is due to barriers being usually among the easiest methods to implement efficiently. In Argo all that is required is a hierarchical barrier protocol that first ensures each node has written back its data, and then that each node can only read the updated data. To this effect, a node-local barrier is used before a self-downgrade to ensure that all threads on a node have reached the barrier. An MPI barrier call is then used to make sure all the nodes have successfully completed the self-downgrade, after which they all self-invalidate. Finally, the threads on each node can be notified to continue their work, which for simplicity is implemented using another node-local barrier.

4.2 Lock-based Synchronization Using HQDL

Lock-based synchronization is expensive in any DSM, since it involves frequent long-latency transactions among nodes. Worse, lock-based synchronization is typically used in critical sections that access a common dataset. Not only the lock acquire operations are expensive but also the execution of the ensuing critical section is penalized if it switches nodes. The reason is that accessing the same data inside a critical section necessitates their transfer between nodes; in other words data accessed in critical sections are *migratory*. Coalescing the execution of critical sections on the same node as much as possible reduces the inter-node transfers of the lock-protected data and is a beneficial policy on any DSM.

In Argo, there is a further incentive to execute as many critical sections in one node before being forced to switch. The reason is the destructive effect of self-invalidation on the rest of the cached data on nodes (those that are not-protected by the lock). Argo's Carina coherence enforces self-invalidation on synchronization with acquire semantics and self-downgrade on release. This holds for *all data*. The situation is worsened by the fact that Carina implements a common page cache for all threads running on the same node. While a common page cache (as opposed to a per-thread or per-core cache) brings spatial and temporal locality benefits in the epochs of synchronization-free execution, a synchronization on *any* thread, self-invalidates or self-downgrades *all other threads' data*.

While this is a interesting problem for further research, in this paper we approach it by lock-based synchronization that does not cause global synchronization between each and every critical section. Similar to how barriers use local barrier calls before global ones, locking algorithms like *cohort locks* [7] (developed for NUMA systems) prioritize synchronization on a single node. Such a hierarchical approach brings the overall performance of a chain of critical sections to (at most) the performance achievable using a single node. It is thus essential to use the fastest algorithm available to execute critical sections on a single node. As delegation locking [18, 20, 19] currently outperforms any traditional locking algorithm, the Argo system introduces primitives for *hierarchical queue delegation locking* (*HQDL*).

Let us describe how HDQL is constructed starting from simple (flat) queue delegation. Similarly to delegation locking, HQDL uses a different interface than traditional locks: Each critical section is issued with a single "delegate" call per critical section, and optionally waits (wait) for the section's execution immediately or at a later point. Often, small source code modifications are needed to "package" critical sections into callable functions in order to be able to delegate them to another thread. Note that threads are allowed to execute non-dependent code between the delegate and the wait but we do not take advantage of this feature in this paper, as it requires—sometimes non-trivial—modifications of application source code. HQDL streamlines the execution of many sections into a single thread, thus reusing local hardware caches more efficiently than if many threads each access the same data. As there is no method of synchronization beyond the ones implemented in Vela, no additional work is required from the programmer.

To build a global QD lock in a distributed system one needs a self-invalidate (SI_fence) when opening the delegation queue and a self-downgrade (SD_fence) when flushing (closing) the queue. The trouble is that allowing transmission of operations to a *remote node* necessitates a self-downgrade by the delegating thread. Additionally, when a thread wants to know if its critical section has been executed successfully (e.g., by executing a wait), we also need to perform self-invalidation in order to be able to see the updates performed by the critical section on possibly *other* nodes. Evidently queue delegation to other nodes does not save us any self-invalidations and self-downgrades. Furthermore, the helper thread which executes critical sections may have to slow down to wait for remote nodes delegating sections.

To address these issues, in the Argo system HQDL only allows delegation of critical sections from the same node as the lock holder is on. Permission to open a delegation queue is determined hierarchically by acquiring a global lock. The node that holds the global lock is the currently active node. On the active node, a self-invalidation is performed in order to "see" data possibly written in earlier executions of critical sections in other nodes. However, after this initial self invalidation, there is no need for more self-invalidations or self-downgrades as long as critical sections are delegated and executed locally. Once the execution of the critical sections finishes (either because there are no more, or a limit is reached) a self-downgrade is performed to make the writes of the executed critical sections globally visible and the global lock is released. This type of locking structure uses significantly fewer SI_fence/SD_fence which would otherwise be very costly: First we eliminate the need for self-invalidating data upon waiting for a specific critical section. This is instead done for the whole node when accessing the global lock. Second, when delegating a critical section, we do not need

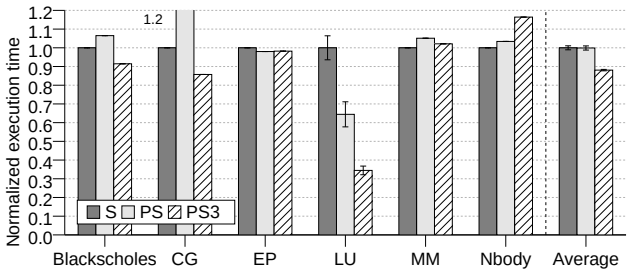


Figure 8: Classification impact on execution time.

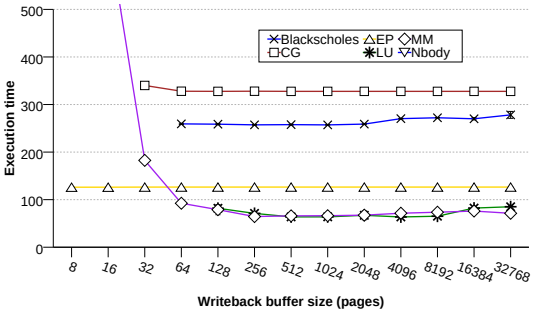


Figure 9: Runtime for different write-buffer sizes.

to flush any buffers, as local node coherence is sufficient.

While this approach reduces the number of threads that can simultaneously delegate to the same lock, it still improves performance: If the program depends on lock performance, it has enough work even on a single node, otherwise there are only negligible stalls on other nodes.

5. EVALUATION

We implement and evaluate Argo on a large distributed cluster in our installation primarily used for HPC workloads. We use at least up to 32 and when possible up to 128 nodes to run our experiments. Each node is equipped with two AMD Opteron 6220 processors featuring two quad-core CPUs per chip, connected using the same interconnect as for communicating with separate processors. Therefore, for NUMA purposes, each machine has four memory nodes, each with only four cores that share a local memory hierarchy (16 cores total). The nodes have at least 64 GB of memory, and are set up with Scientific Linux 6.4, running Linux kernel version 2.6.32-431.20.3.el6.x86_64. All nodes are interconnected with a 2:1 oversubscribed QDR Infiniband fabric. Argo is in its prototype stage and as such it does not have its own custom fine-tuned network layer. Instead, it is built on top of MPI. We use OpenMPI 1.8.4 supporting the MPI3 RMA standard [29]. MPI handles all the bookkeeping needed to start a parallel program on multiple nodes. Each node contributes an equal share of memory to the globally shared memory of the system, and uses an equal amount of memory as a local cache for the global memory. On each run, the amount of memory (user specified) is sized to be large enough to fit the desired workload. Argo uses only 15 out of the 16 available cores of a node, leaving one to take the OS overhead (MPI and UPC use all 16 cores of a node). All results are normalized to a single thread (Pthreads or OpenMP).

5.1 Protocol Choices: S, P/S, P/S3

We first examine the impact of the design choices made for Argo's Carina coherence. As discussed in Section 3.2, Carina implements data classification via the Pyxis passive directory structure. The classification is intended to curb self-invalidation. A fundamental design decision is to avoid message handlers by using more bandwidth, which leads to choices such as the downgrade of private pages (Section 3.4.2). Here, we examine the impact of the three data classification choices: S (no classification—all shared), P/S (the naïve version of P/S classification where private pages are *not* downgraded), and P/S3 the full Carina classification (with private page self-downgrade and writer classification).

Execution time normalized to the S classification for six benchmarks (on four nodes—64 cores) is shown in Figure 8. While the P/S classification aims to reduce latency by not self-invalidating private pages, it incurs a significant checkpointing overhead during synchronization points, in order to service P→S transitions. The end result is that the naïve P/S classification is no better than the S classification. The P/S3 classification overcomes this problem by self-downgrading private pages. The private/shared classification provides the major part of the benefit in terms of reducing the self-invalidation miss rate. For the six benchmarks we examined, additional writer classification further reduced miss rates but by small amounts. The NW (no writer or read-only) and the SW (single writer) classifications exclude shared read-only pages and pages written by a (single) producer from self-invalidation but these behaviors are not prevalent in our benchmark set: i) while there may be abundant read-only data, a single write disqualifies a whole page from being read-only; ii) there is not enough *stable* single-producer/consumer sharing at page granularity.

Lessons learned. P/S classification provides the most benefit in reducing self-invalidation misses but must be performed at a minimal cost: i.e., by self-downgrading private pages. Writer classification offers further small improvements which perhaps are significant at a different granularity, or in benchmarks with different sharing patterns (e.g., dominated by producer/consumer sharing at a coarser granularity) than those we examined.

5.2 Write-Buffer Sensitivity Analysis

While writes are not as critical to performance as reads, excessive write traffic can still be detrimental to performance. The parameter that controls write traffic in Argo is the size of the write buffer that coalesces writes between synchronizations. However, a very large write buffer (that must empty on synchronization) can adversely affect synchronization latency, with serious performance implications. Figure 9 shows execution time as a function of write-buffer size. With small write-buffer sizes, some of our benchmarks exceed their preallocated run time on the HPC cluster and do not even complete (the corresponding points are missing from Figure 9). For all intents and purposes, the performance of these benchmarks is devastated if the write-buffer size falls below a critical point. Execution time correlates well with the number of writebacks, as shown in Figure 10.

On the other hand, for very large buffer sizes there is a slight slowdown, caused by the overhead on synchronization and other overheads directly related to write buffer size. The set of benchmarks shown here has very little synchronization (in the form of a handful of barriers). Through experimenta-

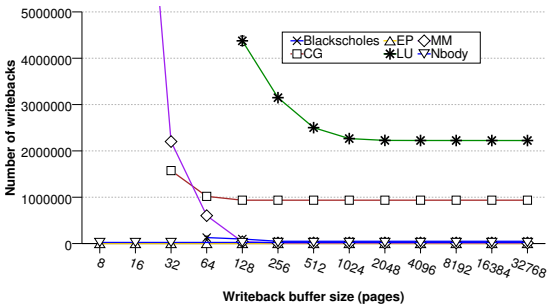


Figure 10: Writebacks for different write-buffer sizes.

tion, the write buffer was chosen to contain 8192 pages for Blackscholes, Nbody and LU, 256 pages for GC, 128 pages for MM, and 32 pages for EP in the remaining experiments.

Lessons learned. A minimum write buffer size is often required to run a program, but increasing the size further neither yields greater benefits, nor hurts performance when synchronization is light.

5.3 Lock Algorithms: Cohort vs HQDL

Since the benchmarks examined so far do not have significant synchronization (no lock synchronization), Argo's behavior in the face of intensive lock synchronization is analyzed using an appropriate micro-benchmark constructing a concurrent priority queue from a fast sequential implementation and a lock to access it. This kind of benchmark is regularly used to evaluate locking algorithms [12, 8, 20]. In particular, the benchmark uses a *pairing heap* [9], which has been shown to outperform non-blocking priority queue implementations when used together with flat combining [12].

The benchmark measures throughput: the number of operations that N threads, starting at the same time and executing a loop, can perform during a period of t seconds. The loop body consists of some amount of thread-local work and a global operation (either `insert` or `extract_min`) which is selected randomly with equal probability. While `extract_min` operations wait for their result, `insert` operations do not expect a result back and therefore can continue immediately after successful delegation. The seed for the random number generator is thread-local to avoid false sharing between threads. The thread-local work can be specified in work units, where each work unit performs updates to two randomly selected elements of a thread-local array of 64 integers. Figure 11 shows the scaling behavior of different locking algorithms on a single machine. As can be seen, Pthreads has performance issues even on a single NUMA machine, and QDL outperforms the alternative Cohort locks.

The benchmark results shown in Figure 12 use 48 local work units and 15 threads per node. Clearly, the benchmark does not scale, but this is to be expected of an application

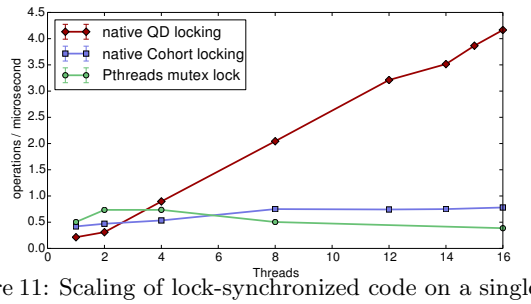


Figure 11: Scaling of lock-synchronized code on a single node.

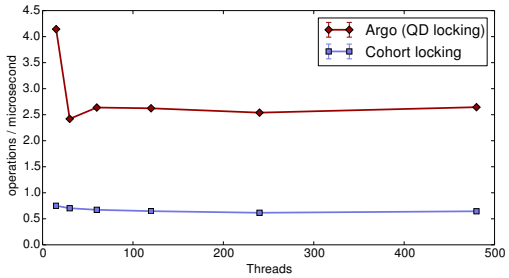


Figure 12: Scaling of lock-synchronized code using DSM.

which is dominated by its critical sections. When going from a single node to two, Argo's Vela HQDL performance drops by 42% compared to the performance of QDL within a node, but stays stable across a large number of nodes and significantly outperforms a state-of-the-art Cohort lock implementation under the same conditions (Figure 12).

Lessons Learned. Delegating critical sections to remote nodes does not save self-invalidations and/or self-downgrades. Instead hierarchical queue delegation should be performed that delegates only locally.

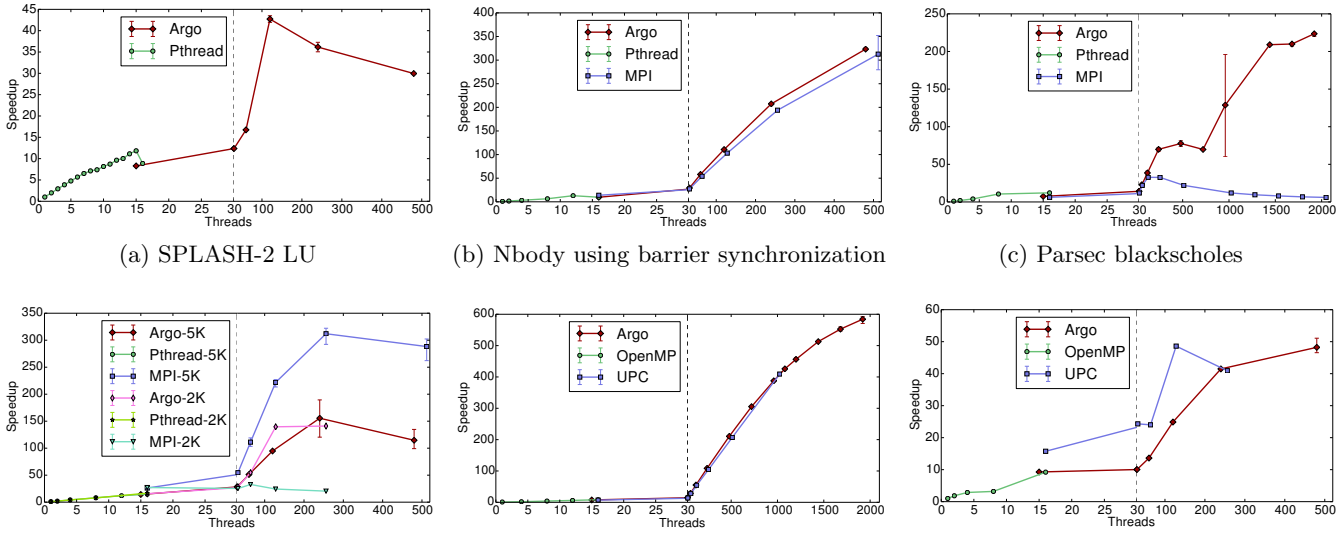
5.4 Scalability and Comparison to MPI

The benchmarks here are written for regular multicore systems using Pthreads, with the necessary adaptions made to run on our system. For comparison, all graphs also show the behavior of the unmodified code on a single machine.

For our first benchmark, we only show scalability compared to its Pthreads implementation. We use the well-known matrix triangulation benchmark from SPLASH-2, commonly known as LU [38]. The results are shown in Figure 13a. As this benchmark involves a lot of data migration within the system, there is significant overhead when running it on Argo. Still, using multiple nodes outperforms the Pthreads version on a single machine, and continues to gain performance up to eight nodes.

Figure 13b shows the scaling of a custom implementation of the n-body problem. It uses a simple iterative approach, separating iteration steps with barriers. The additional cost of synchronization over a network is barely noticeable for large problem sizes and does not prevent scaling of this benchmark up to 32 nodes (512 cores), exceeding that of the MPI port. The blackscholes benchmark from Parsec [3] uses only a single barrier synchronization at the end of each benchmark iteration, resulting in the very good scaling shown in Figure 13c for up to 128 nodes (2048 cores), in contrast to the MPI version that stops scaling at 16 nodes (256 cores). Variations in Argo's speedup (at 32, 48 and 64 nodes) are mainly due to the overly simplistic data distribution and its negative interaction with Argo's prefetching. We implemented a naïve Matrix Multiplication benchmark and show its scaling with two input sets: 2000×2000 and 5000×5000. Speedup is shown in Figure 13d. The MPI version has an algorithmic advantage as it is already faster in a single node. However, whereas the MPI version with the small input cannot maintain its advantage beyond one node, Argo scales well up to eight nodes (128 cores) before flattening out. For the large input, Argo scales in a similar way to the MPI version but the initial difference in the single-node speed is carried all the way up to 32 nodes (512 cores).

[3]Thanks to Tobias Skoglund for providing this benchmark.



(a) SPLASH-2 LU (b) Nbody using barrier synchronization (c) Parsec blackscholes

(d) Matrix Multiply 5000×5000 & 2000×2000 (e) EP CLASS D (ported from NAS) (f) CG CLASS C (ported from NAS)

Figure 13: Benchmark scaling to 32(128) nodes, 512(2048) cores.

5.5 Comparison to UPC

Our final comparison is against UPC benchmarks. The scaling of the highly-parallel EP benchmark, from the NAS Parallel Benchmark suite, is shown in Figure 13e. In this case the scaling behaviour continues similarly to that on a single machine for both UPC and Argo up to 128 nodes (2048 cores; UPC is limited to 1024 cores). This shows that Argo can compete directly with PGAS systems that require significant effort to program in. The CG benchmark from the NAS suite (Figure 13f) is another case of the non-Pthreads benchmark starting with a significant advantage over the Argo implementation in one node. This advantage, however, withers as the UPC version stops scaling earlier than Argo (at eight nodes, 128 cores) whereas Argo continues up to 32 nodes (512 cores). Whereas UPC performance is initially higher, due to an optimized implementation, Argo still achieves good scaling without changing the algorithm.

6. CONCLUSIONS

In this paper we are re-thinking software distributed shared memory in relation to the technology trends as they developed over the last decade. We have identified the latency of a centralized coherence protocol that indirects via a home node directory, the overhead of software message handlers, the latency of remote synchronization, and the accompanied overhead of data migration incurred by the distributed execution of critical sections, as major impediments to the end performance and hence scalability of a DSM. To this end, we propose a new approach to coherence and a new approach to synchronization that overturn standard practices. The result is exemplified in a prototype implementation of a page-based, user-space DSM system called Argo, which currently is built on top of MPI.

Argo's coherence (Carina) takes distributed decisions using self-invalidation and self-downgrade and does not use message handlers. Instead all actions are performed via RMA by requesting nodes. To reduce the number of misses caused by self-invalidation we introduce passive directories (Pyxis) that perform data classification to help nodes filter what to self-invalidate. The novelty of the Pyxis directory structure is that it is implemented without any message handlers. To handle synchronization and critical section execution we turn to the current performance-leading paradigm of queue delegation locking but we observe that it is not appropriate for a distributed environment where one may have to delegate remotely. Thus, we propose a hierarchical extension of queue delegation locking (implemented as the Vela synchronization system), that prioritizes local delegation before switching to other nodes.

Argo is in its prototype phase and uses MPI as its networking layer. We have ported an initial set of benchmarks—expanding rapidly—to assess our design decisions. Despite the limitations of a prototype implementation, results are highly encouraging. Several benchmarks show significant performance improvements due to Argo's distributed coherence and easily match or exceed their message-passing or PGAS implementations. However, the full potential of Argo is not tapped yet. As more benchmarks come online, future work will rework them for *detached delegation* of critical sections where one can overlap their execution with useful work. Further, we plan to examine the relation of granularity, data placement, and classification to enhance the opportunities offered by the P/S3 classification scheme to reduce miss rates.

Acknowledgements

This work was carried out within the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center).

7. REFERENCES

[1] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *17th ISCA*, pages 2–14, June 1990.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. Tech. rep. 865207, Carnegie Mellon University, Jan. 1993.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th PACT*, Oct. 2008.

[4] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *20th PACT*, pages 155–166, Oct. 2011.

[5] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Tech. rep., Dept. of CSE, University of Washington, Seattle, 1993.

[6] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th ISCA*, pages 93–103, June 2011.

[7] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. In *17th PPoPP*, pages 247–256, 2012. ACM.

[8] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *17th PPoPP*, pages 257–266, 2012. ACM.

[9] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th ISCA*, pages 15–26, 1990.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th ISCA*, pages 184–195, June 2009.

[12] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *22nd SPAA*, pages 355–364, 2010. ACM.

[13] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *8th SPAA*, pages 277–287, June 1996.

[14] S. Kaxiras and G. Keramidas. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro*, 30(5):54–65, Sept. 2011.

[15] S. Kaxiras and A. Ros. A new perspective for efficient virtual-cache coherence. In *ISCA*, pages 535–547, 2013.

[16] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *USENIX*, pages 115–132, Jan. 1994.

[17] P. J. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *19th ISCA*, pages 13–21, May 1992.

[18] D. Klaftenegger, K. Sagonas, and K. Winblad. Brief announcement: Queue delegation locking. In *26th SPAA*, pages 70–72, 2014. ACM.

[19] D. Klaftenegger, K. Sagonas, and K. Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par*, LNCS, 2014.

[20] D. Klaftenegger, K. Sagonas, and K. Winblad. Queue delegation locking, 2014. http://www.it.uu.se/research/group/languages/software/qd_lock_lib.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[22] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *22nd ISCA*, pages 48–59, June 1995.

[23] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[24] K. Li. IVY: A shared virtual memory system for parallel computing. In *ICPP*, pages 94–101, Aug. 1988.

[25] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX*, pages 65–76, USA, 2012. USENIX Association.

[26] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. In *12th ICPP*, pages 801–810, 2006. Springer-Verlag.

[27] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *8th ISPP*, pages 165–171, 1994. IEEE Computer Society.

[28] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.

[29] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 2012. available at: http://www.mpi-forum.org/docs (January, 2015).

[30] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Tech. Rep, Dept. of CSE, Univ. of Washington, Feb 2014.

[31] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *IWPDC*, pages 182–204. 1999.

[32] S. Phillips. M7: Next generation SPARC. In *26st HotChips Symp.*, Aug. 2014.

[33] Z. Radović and E. Hagersten. DSZOOM – low latency software-based shared memory. Technical report, Parallel and Scientific Computing Institute, 2001.

[34] Z. Radović and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *9th HPCA*, pages 241–252. IEEE Comp. Society, 2003.

[35] B. Ramesh. *Samhita: Virtual Shared Memory for Non-Cache-Coherent Systems*. PhD thesis, Virginia Polytechnic Institute and State University, July 2013.

[36] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato. Temporal-aware mechanism to detect private data in chip multiprocessors. In *42nd ICPP*, pages 562–571, Oct. 2013.

[37] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *21st PACT*, pages 241–252, Sept. 2012.

[38] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[39] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *14th ASPLOS*, pages 253–264, New York, NY, USA, 2009. ACM.

[40] UPC Consortium. UPC language specifications, v1.2. Tech. Rep, Lawrence Berkeley National Lab, 2005.

[41] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *3th ASPLOS*, pages 243–256, Apr. 1989.