Callback: Efficient Synchronization without Invalidation with a Directory Just for Spin-Waiting

Alberto Ros
Department of Computer Engineering
Universidad de Murcia, Spain
aros@ditec.um.es

Stefanos Kaxiras
Department of Information Technology
Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

Abstract

Cache coherence protocols based on self-invalidation allow a simpler design compared to traditional invalidation-based protocols, by relying on data-race-free (DRF) semantics and applying self-invalidation on racy synchronization points exposed to the hardware. Their simplicity lies in the absence of invalidation traffic, which eliminates the need to track readers in a directory, and reduces the number of transient protocol states. With the addition of self-downgrade these protocols can become effectively directory-free. While this works well for race-free data, unfortunately, lack of explicit invalidations compromises the effectiveness of any synchronization that relies on races. This includes any form of spin waiting, which is employed for signaling, locking, and barrier primitives.

In this work we propose a new solution for spin-waiting in these protocols, the callback *mechanism, that is simpler and more efficient than explicit invalidation. Callbacks are set by reads involved in spin waiting, and are satisfied by writes (that can even precede these reads). To implement callbacks we use a small (just a few entries) directory-cache structure that is intended to service only these "spin-waiting" races. This directory structure is self-contained and is not backed up in any way. Entries are created on demand and can be evicted without the need to preserve their information. Our evaluation shows a significant improvement both over explicit invalidation and over exponential back-off, the state-of-the-art mechanism for self-invalidation protocols to avoid spinning in the shared cache.*

1. Introduction and Motivation

In self-invalidation protocols, writes on data are not explicitly signaled to sharers. It is straightforward to show that data races throw such protocols in disarray, producing non-sequential-consistent executions [14, 23]. All proposals thus far [5, 7, 12, 13, 14, 22, 23, 26] offer sequential consistency for data-race-free (DRF) programs [3].[1]

Despite the advantages of self-invalidation, there are situations where explicit invalidation is better. For instance, spin-waiting can be performed more efficiently with invalidations and local spinning on a cached copy, rather than repeatedly self-invalidating and re-fetching, which is the way self-invalidation protocols get the value updated by writes.

Without explicit invalidations, complex and expensive solutions have been proposed for synchronization. To implement locks, VIPS-M blocks last-level cache (LLC) lines and queues requests at the LLC controller [23]; while DeNovoND uses complex hardware queue locks (based on QOSB [11]) with lock bits and queue pointers in the L1 caches [25, 26]. Spin-waiting on a variable causes repeated self invalidation in VIPS-M, while DeNovoND [26] assumes the existence of barriers,[2] later implemented with exponential back off [25].

Repeated self-invalidations of a spin flag mean that we spin directly on the LLC. This dramatically increases network traffic and LLC accesses and, consequently, is a very costly solution in terms of energy consumption. To alleviate this problem exponential back-off is used, as for example in VIPS-M [23] and in DeNovoSync [25]. To cap the back-off interval to reasonable levels —and hence the delay penalty incurred on the last access that exits the spin-waiting— exponentiation must cease after a few times.

Figure 1 compares invalidation, LLC spinning with exponential back-off using four limits for the number of exponentiations: 0 (no back-off), 5, 10, and 15 times. The two sets of bars in each of the graphs show the results (normalized to the largest value) for spin-waiting in a CLH queue lock [8, 17] and in a tree sense-reversal barrier [19], while the two graphs show number of LLC accesses and latency (in cycles), respectively.[3]

ISCA'15, June 13-17, 2015, Portland, OR USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750405

[1]Conceptually, one could eliminate the requirement for DRF by self-invalidating after every use of data, but this would defeat the purpose of caching.

[2]Hardware barrier implementations would tie very well with self-invalidation as they can be directly exposed to the coherence layer and can initiate the proper self invalidations. Unfortunately, we cannot assume their availability in the general case.

[3]Results represent the geometric mean of the synchronization of all benchmarks evaluated in this work (see Section 5) for a 64-core system.

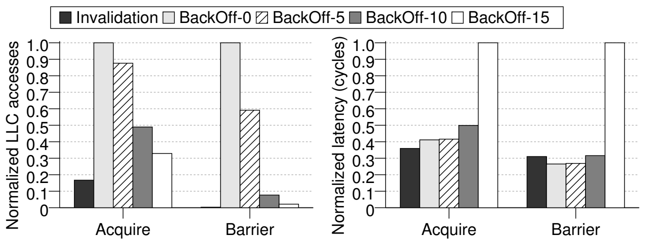


Figure 1: Explicit invalidation vs. self-invalidation

It is obvious from this figure that invalidation is very efficient while LLC spinning trades off the number of LLC accesses to latency, depending on the limit of the number of exponentiations. While it can be made competitive to invalidation, this comes at a significant cost in traffic. Conversely, LLC spinning requires precise tuning as it can incur significant delays when one tries to minimize traffic. Thus, there is no "best" back-off for both time and traffic because it is always a trade-off between the two metrics.

Thus, it is tempting to consider reverting back to explicit invalidation for a small set of addresses, namely spin variables. However, explicit invalidations are unsolicited and unanticipated, giving rise to a number of drawbacks that make them unappealing —if our overarching goal is to simplify coherence. Because they are unanticipated, invalidations cause significant protocol state explosion to resolve protocol races. Because they are unsolicited, invalidations break the mold of a simple request-response protocol, meaning that they cannot be used for virtual caches without reverse translation [13].

Our objective is to address a weakness of self invalidation, but at the same time refrain from compromising some of its important properties such as protocol simplicity and compatibility with virtual cache coherence [13].

To this end, we propose a new simple and transparent *callback* mechanism to avoid repeated self-invalidation and re-fetch of data. Callbacks follow a request-response paradigm, are efficiently implemented in a small structure, and eliminate wasteful re-fetches of data while at the same time allowing the cores to pause (e.g., to save energy) rather than actively spin on a cached copy waiting for a change of state. Callbacks are inherently more efficient than invalidation in terms of traffic. Communicating a new value requires five messages with invalidation while only three with callback. (Section 3).

Callbacks are set by reads wishing to be notified of a new value if one is not readily available. A write that creates a new value notifies all (or alternatively just one) of the readers that are waiting for it. We propose a number of callback variants to optimize common synchronization idioms (Section 2) and we give a detail description of the resulting algorithms (Section 3).

Callbacks are implemented efficiently in a small directory-cache structure that is not backed-up by external storage. Entries in a callback directory are created and evicted without introducing the protocol complexity involved in directory replacements. This is due to the simple semantics of the callback that allow the loss of directory information and re-initialization at a known state.

More abstractly, we propose a directory just for races involved in spin-waiting. The reasoning is the following: in an environment with self-invalidation and self-downgrade, responsibility for coherence is distributed to the core caches when it comes to data-race-free accesses. In other words, there is no need to track data-race-free data in a centralized directory. What remains are the conflicting accesses, that need a meeting point. This is straightforward in SC for DRF [3], the model of self-invalidation protocols, since all synchronization must be exposed to the hardware, and any conflicting accesses must be synchronization (otherwise the model would be violated). Further, we exploit the fact that races are unordered and specify how directory entries can be evicted without having to preserve their information.

We evaluate our proposals using simulation and an extensive set of benchmarks (Section 5). Our results show that, the proposed callbacks invariably retain the benefits of self-invalidation and self-downgrade, but most importantly, severely limit the penalties when these appear in synchronization-intensive benchmarks, bringing a truly simple and competitive coherence a step closer to reality.

2. Optimizing Synchronization with Callbacks

While self-invalidation can be optimized for race-free data, it shows an inherent weakness when it comes to spin-waiting. Entering a critical section or just spin-waiting for a change of state requires repeated self-invalidation of the lock or flag variable —we assume that atomic instructions always go to the LLC as in [23]. And herein lies the problem: spin loops cannot spin on a local copy of the synchronization variable which would be invalidated and re-fetched only with the writing of a new value. Repeated self-invalidation leads to excessive traffic to the LLC.

The solutions that have been proposed so far are costly. For locks, they involve some form of hardware queuing. The VIPS-M approach uses a blocking bit in the LLC cache lines and queues requests in the LLC controller when this bit is set [23]. DeNovoND proposes a full-blown hardware implementation of queue locking based on the QOSB operation [11, 25, 26]. The cost and complexity of these proposals is not trivial. Further, they tie the lock algorithm to the specifics of the hardware implementation (so the lock algorithm inherits, for better or worse, whatever fairness, starvation, live-lock properties, etc. are offered by the hardware mechanism [24]). For spin-waiting, both VIPS-M [23] and DeNovoSync [25] rely on exponential back-off.

What we need is a similar effect to these complex proposals, only much simpler and much more general (not restricting the lock algorithm by the hardware, nor imposing exponential back-off for spinning).

2.1. Callback

Our solution is a "callback" read operation that can be applied to loads that participate in spin-waiting. A *Callback read* blocks waiting for a write, if no intervening write happened since its last invocation. In many respects the callback concept reminds of a C++11 "future" [2] but is more low-level, and more flexible. Similarly to a future, if the value has been created it can be readily consumed; otherwise, like a future, a callback awaits the creation of the value. Our proposal is inspired by the Full/Empty concept [21] but is different. It works as follows: A core issues a callback read to an address. If the address has been written (i.e., the value is in state "full") the read completes immediately; otherwise it blocks and awaits a write. So far it is similar to the functionality of a Full/Empty bit. The novelty of our approach is that we allow any set of cores to simultaneously issue callback reads to the same address and simultaneously consume the same write, but at the same time allow complete freedom on when a callback can be set by a core with respect to writes. Writes never block as is the case with Full/Empty bits. The callback optimizes reads in spin-waiting in an environment lacking explicit invalidations.

Thus, our objective is to provide the benefit of explicit invalidation, *for only these few accesses that need it*, but without the cost of implementing a full-blown invalidation protocol, and without having to track indiscriminately all data in a directory (as this would obviously defeat the purpose of simplifying cache coherence with self-invalidation). A callback is different than an invalidation as it is explicitly requested and waited upon.[4] This is what makes callback simple. From the point of view of the cache, a callback read is still an ordinary request-response transaction —no other complexity is involved. It introduces no protocol races because cores are either blocked waiting for the callback response or immediately complete the read. From the point of view of the LLC, a callback is a simple data response without any further bookkeeping.

In contrast, invalidations are unsolicited and their arrival unanticipated. An invalidation can arrive to a core at any time and at any cache state, which dramatically increases the protocol race conditions and therefore the number of states required in the caches to account for all the scenarios. In addition, invalidations are not suitable for efficient virtual-cache coherence [13], for the same reason. Invalidations are not anticipated at a virtual cache, which means that a reverse translation (physical to virtual) must be performed to ensure correct delivery to the virtual cache. In contrast, callbacks are explicitly waited for (as a result of a request) so they require no address identification (other than an MSHR index).

In terms of energy efficiency, callbacks have two advantages over invalidations. First, callbacks are more efficient in the number of messages needed to communicate a new value. A callback requires three messages: {callback, write, data} or {write, callback, data} depending on the relative order between the read and the write. Invalidation, however, requires five: {write, invalidation, acknowledgment, load, data}. A further important benefit of a callback is that a core can easily go into a power-saving mode while waiting. Many works have shown that significant energy savings are possible by slowing down non-critical threads that are simply waiting for critical threads to reach a synchronization point (typically a barrier) [15, 16]. However, demonstrating this benefit is outside the scope of this paper and is left for future work.

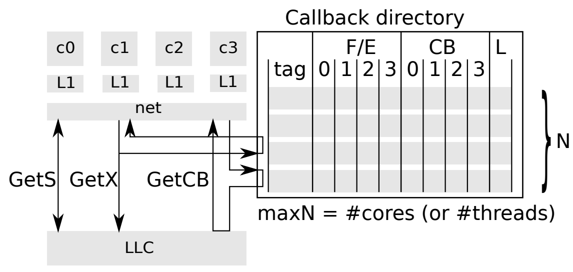


Figure 2: Callback Directory

2.2. Implementation

To implement callbacks we need a callback bit (CB) and a Full/Empty (F/E) bit per core. Potentially every address could have its own set of CB and F/E bits, but of course, we argue that this is not needed, nor wanted. First, as we argued in the introduction, CB and F/E bits are needed only for a limited set of addresses (for synchronization variables, locks and flags) so the callback directory does not need to cover the entirety of the address space or even the entirety of the cached data. Second, CB and F/E bits are primarily needed to handle the case where a (repeating) read is trying to match a specific value of a write. Such "ongoing" races at any point in time typically concern very few addresses. Thus, a very small directory cache at the LLC maintains this set of bits just for few addresses (Figure 2).

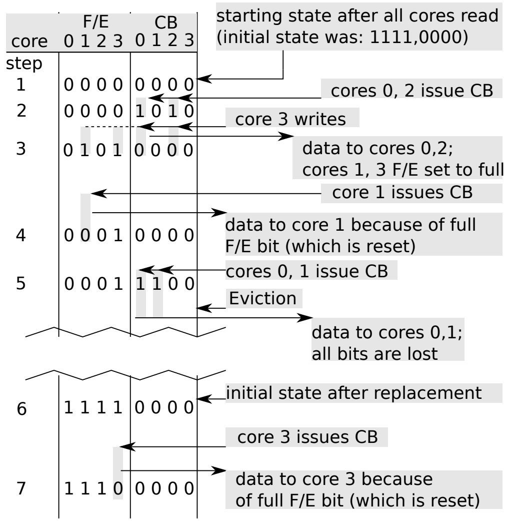
The callback directory is managed as a cache (with evictions and a replacement policy) but is completely *self-contained* and *not* backed by main memory. This is achieved by the way entries and initialized and evicted and is explained in detail below. Only callback reads can install an entry in the directory (any other read or write cannot affect the contents of the directory).

A reasonable upper limit for the number of entries in this directory would be equal to the number of cores (since there can be only so many callbacks set on different variables as the number of cores).[5] Further, since we only need to track the CB bits for a limited set of addresses, we can afford to do this at a word granularity, thus allowing independent callback reads for individual words in a cache line.[6] We explain below what happens on replacement (and thus complete loss of the CB and F/E bits for some address).

[4]For similar reasons, a callback is different than unsolicited update operations (e.g, as in update protocols).

[5]This can be *optionally* extended to the number of threads for multi-threaded cores.

[6]This works well with the individual-word updating of the LLC lines in a self-downgrade protocol such as VIPS-M [23].



F/E
CB
core 0 1 2 3 0 1 2 3
step
starting state after all cores read (initial state was: 1111,0000)
1 0 0 0 0 0 0 0 0
cores 0, 2 issue CB
2 0 0 0 0 1 0 1 0
core 3 writes
3 0 1 0 1 0 0 0 0
data to cores 0,2; cores 1, 3 F/E set to full
core 1 issues CB
data to core 1 because of full F/E bit (which is reset)
4 0 0 0 1 0 0 0 0
cores 0, 1 issue CB
5 0 0 0 1 1 1 0 0
Eviction
data to cores 0,1; all bits are lost
initial state after replacement
6 1 1 1 1 0 0 0 0
core 3 issues CB
7 1 1 1 0 0 0 0 0
data to core 3 because of full F/E bit (which is reset)

Figure 3: Callback example

It is also important to note that in contrast to invalidation directories, a callback directory is not in the critical path of reads (GetS) or writes (GetX). In the latter case, the callback directory is accessed in parallel with the LLC. As shown in Figure 2 only callback reads (GetCB) need to consult the callback directory before accessing the LLC.

2.3. Callback Example

Figure 3 shows how the callback directory functions. The example concerns a single directory entry for a variable accessed by four cores (core0..core3). There are four F/E bits and four CB bits, one per core. When an entry is allocated in the callback directory it is initialized with all the F/E bits set to 1 (full), and all callback bits set to 0 (no callback) —recall that the directory is not backed up, so its entries must be created anew. This is also the default state *after* F/E and CB bits are lost due to a replacement.

The first time a core issues a callback to a newly initialized entry it will find the corresponding F/E bit full. This means that the core simply reads the current value of the data and the F/E bit is reset. Assume that all cores read the variable after its callback entry is installed in the directory so the starting state of all the bits is 0, in our example.

Assume now that cores 0 and 2 issue callback reads to this address, setting the corresponding CB bits (step 2). The callback reads block since there is no value to consume. When later a write is performed on the address from core 3, the callbacks are activated and two wakeup messages carry the newly created value to cores 0 and 2. Now, the corresponding callbacks are set to false (0) and *the unset F/E bits of the cores that did not have a callback are set to full* (step 3). Cores that had a callback have now consumed the write, but cores that did not set a callback can henceforth directly access the written value because their corresponding F/E bits are set to full. When a core issues a callback and finds its F/E bit set (full), it consumes the value and leaves both its F/E bit and callback bit unset (empty and no callback). This is shown in step 4. It is clear from this example that a callback can consume a single write, whether it happens before or after it. If the write happened before then the callback immediately returns, otherwise it will get called back from the write itself. It is also clear that we have a bulk behavior for all the cores that are waiting on a callback.

2.3.1. Loss of F/E and CB Bits. But what happens if a replacement in the directory causes the F/E and CB bits to be lost for an address? Since we do not back the directory in the main memory, our solution for this is to send the data to all callbacks that are set on the address (step 5). The cores that have a callback are notified that their callback is answered but without getting a newer value.

In a directory replacement both the CB bits and the F/E bits are lost. When a new entry is created as a result of a callback read that misses in the directory, all its F/E bits are set to full, and since there cannot be any outstanding callbacks all the CB bits are set to 0 (step 6).

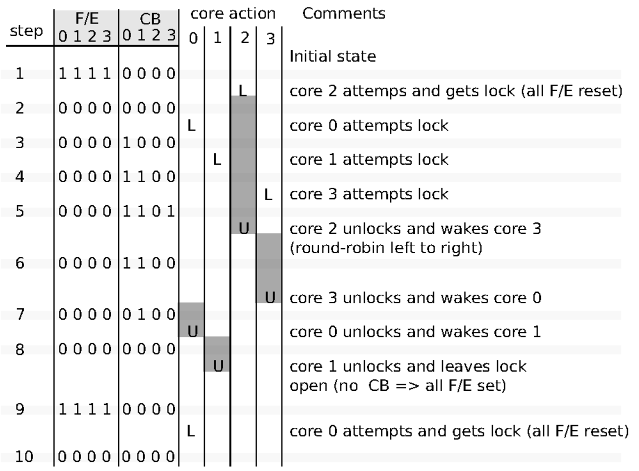
2.4. Callback-All vs. Callback-One

The callback mechanism described so far is intended to optimize the case of a data race involving multiple reads conflicting with a write. When a new value is produced by a write all (waiting) reads are woken up. Likewise, reads from many different cores may consume the same value that was previously written. This fits well synchronization idioms having a broadcast or multicast behavior (e.g., barriers). However, if one considers lock synchronization, a callback mechanism that wakes up all waiting reads may be inefficient. In lock synchronization only one out of a set of competing lock acquires succeeds. Thus, releasing a lock, which is a write on the lock variable, should wake up one waiting acquire instead of all. Likewise, a free lock could be read by the first lock acquire that arrives, rather than all. This kind of behavior requires a functional change to our basic callback mechanism. But the structure of the callback directory remains unchanged —only functionality changes.

Write$_{CB1}$. To optimize the case of locks, we use a variant of the write that wakes up a single waiting callback: write$_{CB1}$. Note that the number of callbacks that are satisfied is specified by the write —not the reads that set the callbacks.

Using a write$_{CB1}$ wakes up a single callback, if there are any callbacks set. But it also has another equally important effect: it forces all F/E bits to act in unison, i.e., behave as a single F/E bit. This "mode" change in the functionality of the F/E bits is encoded in a "All/One" (A/O) bit in the callback directory entries. This bit is set to "All" by default and the F/E bits act individually as described in the previous section. In this case the entry is a "*callback-all*" entry. A write$_{CB1}$ sets the A/O bit to "One" causing the F/E bits to behave in unison and making the entry "*callback-one*." (Any normal write or read resets the A/O bit to "All.")

Example. Figure 4 gives a high-level example, without going into much detail on the lock implementation, something that



step	F/E 0 1 2 3	CB 0 1 2 3	core action 0	1	2	3	Comments
							Initial state
1	1 1 1 1	0 0 0 0					
					L		core 2 attemps and gets lock (all F/E reset)
2	0 0 0 0	0 0 0 0					
			L				core 0 attempts lock
3	0 0 0 0	1 0 0 0					
				L			core 1 attempts lock
4	0 0 0 0	1 1 0 0					
						L	core 3 attempts lock
5	0 0 0 0	1 1 0 1					
					U		core 2 unlocks and wakes core 3 (round-robin left to right)
6	0 0 0 0	1 1 0 0					
						U	core 3 unlocks and wakes core 0
7	0 0 0 0	0 1 0 0					
			U				core 0 unlocks and wakes core 1
8	0 0 0 0	0 0 0 0					
				U			core 1 unlocks and leaves lock open (no CB => all F/E set)
9	1 1 1 1	0 0 0 0					
			L				core 0 attempts and gets lock (all F/E reset)
10	0 0 0 0	0 0 0 0					

Figure 4: High-level view: Lock-optimized callback with write$_{CB1}$: only one callback is serviced when the semantics of a write (in this case lock release) indicate so. In such case, all the F/E bits act in unison (all ones or all zeroes).

we will revisit shortly. Assume that the A/O bit of the callback entry is already set to "One" and the entry has the F/E bits set to "full" (step 1). Core 2 gets the lock and since it reads the lock value *all* the F/E bits are set to "empty" (step 2). Thus, no other core can now read the value of the lock (since it is "empty") and have to set their corresponding CB bit and wait (steps 3,4,5).

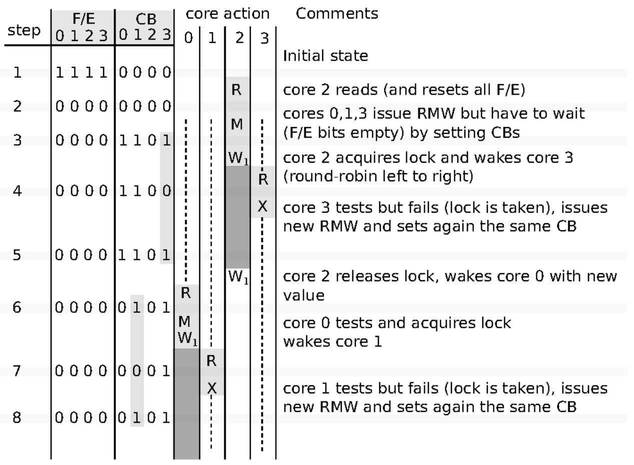
When core 2 releases the lock with a write$_{CB1}$ it wakes up just one of the waiting cores (step 6). Which one is a matter of policy: random, FIFO, round-robin, etc. Each policy has an extra cost to implement, that we do not study here. We use a simple, easy-to-implement, pseudo-random policy, starting from any set CB bit and proceeding round-robin towards cores with higher IDs (wrapping around at the highest ID). In Figure 4, with the pseudo-random round-robin policy the order that the cores get the lock is 2,3,0,1, which is different that the arrival of acquires at the callback directory (2,0,1,3).

In contrast to callback-all, when a write$_{CB1}$ satisfies a callback, it leaves the F/E bits *undisturbed*, set to "empty" (step 9). (In a callback-all we would set to "full" the F/E bits of the cores that did not have callbacks.) Note that in all cases, no individual F/E bit is set or reset but rather all of them as if they were one. In fact this is the abstraction of the callback-one: a value can only match one read that either precedes or succeeds the write creating this value.

2.4.1. Loss of Directory Information in Callback-One. Similarly to callback-all, a callback-one directory entry can be evicted simply by satisfying all its callbacks with the current value. The starting state, when it is brought back in the directory, is all F/E bits set to "full" and all CB bits cleared. The All/One bit is reset to "All."

2.5. A Step Further: Write$_{CB0}$

In the previous section we treated the lock acquire as a read. However this is a simplification for clarity. In reality, lock



step	F/E 0 1 2 3	CB 0 1 2 3	core action 0	1	2	3	Comments
							Initial state
1	1 1 1 1	0 0 0 0					
					R		core 2 reads (and resets all F/E)
2	0 0 0 0	0 0 0 0					cores 0,1,3 issue RMW but have to wait (F/E bits empty) by setting CBs
					M		
3	0 0 0 0	1 1 0 1					
					W$_1$	R	core 2 acquires lock and wakes core 3 (round-robin left to right)
4	0 0 0 0	1 1 0 0					
						X	core 3 tests but fails (lock is taken), issues new RMW and sets again the same CB
5	0 0 0 0	1 1 0 1					
					W$_1$		core 2 releases lock, wakes core 0 with new value
			R				
6	0 0 0 0	0 1 0 1					
			M W$_1$				core 0 tests and acquires lock wakes core 1
7	0 0 0 0	0 0 0 1					
				R			
				X			core 1 tests but fails (lock is taken), issues new RMW and sets again the same CB
8	0 0 0 0	0 1 0 1					

Figure 5: A more detailed view: RMW inefficiencies with write$_{CB1}$. Core 3 is prematurely woken up by the RMW of core 2 but cannot take the lock (steps 4-5). Core 0 instead takes the lock after it wakes up by the lock release of core 2 (5-6). In turn, core 0 prematurely wakes up core 1 only for the latter to lose its turn since it cannot get the lock (7-8).

acquires are implemented using atomic primitives such as T&S, Fetch&*func*, CAS, or others. In general, we consider the case where a lock acquire is a read-modify-write (RMW) operation. While the read of the RMW is the callback we referred to in the previous section, there is also a write that we did not account for. In the case of a lock implemented with Test&Set (T&S) the write sets the lock to "taken" when the test succeeds (i.e., when it finds the lock "not-taken").

As per our discussion above, the write can be a write$_{CB1}$ that satisfies only one out of many waiting callbacks. However, in the case the Test succeeds and the Set takes place, then there is no need to wake up *any* callback, as its corresponding RMW is doomed to fail. This expectation holds only for the write of successful lock acquires, the target of our optimization here.

Example. Figure 5 shows the performance issue with write$_{CB1}$. Core 2 performs a RMW and gets the lock. In the process, its read sets all F/E bits to "empty" (0). No other core can now read the value of the lock. Instead, subsequent reads must set a callback (steps 2-3 in Figure 5). If the RMW succeeds and writes the lock using a write$_{CB1}$, it will wake up one of the waiting cores —in this example, core 3. However, since the lock was just acquired by core 2, the acquire of core 3 is bound to fail and has to be repeated. In effect, core 3 loses its turn because it was woken up prematurely. When the lock is then released with a write$_{CB1}$, core 0 is woken up. Its RMW succeeds (the lock just changed value) and core 0 enters its critical section by writing the lock. Core 0, in turn, prematurely wakes up core 1.

This situation is avoided if the write of the RMW does not wake up any callbacks: i.e., a write$_{CB0}$. Figure 6 shows the same example using write$_{CB0}$ in the RMW operations. By exploiting knowledge of the semantics of a lock acquire, write$_{CB0}$ optimizes the hand-off of the lock among cores, without unnecessary traffic.

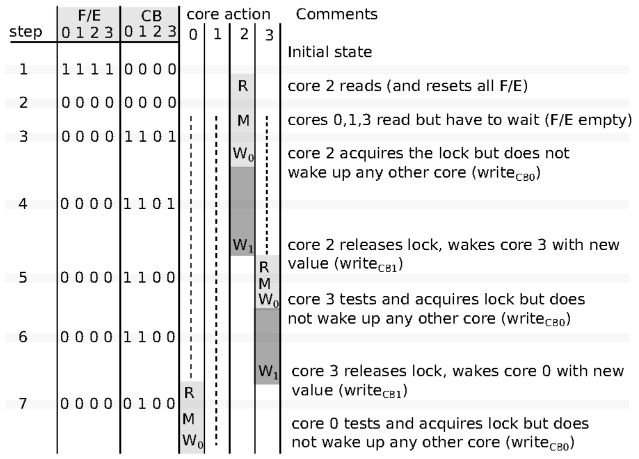

step
F/E 0 1 2 3
CB 0 1 2 3
core action 0 1 2 3
Comments
Initial state
1 1 1 1 1 0 0 0 0
R
core 2 reads (and resets all F/E)
2 0 0 0 0 0 0 0 0
M
cores 0,1,3 read but have to wait (F/E empty)
3 0 0 0 0 1 1 0 1
W_0
core 2 acquires the lock but does not wake up any other core (write$_{CB0}$)
4 0 0 0 0 1 1 0 1
W_1
core 2 releases lock, wakes core 3 with new value (write$_{CB1}$)
R M W_0
5 0 0 0 0 1 1 0 0
core 3 tests and acquires lock but does not wake up any other core (write$_{CB0}$)
6 0 0 0 0 1 1 0 0
W_1
core 3 releases lock, wakes core 0 with new value (write$_{CB1}$)
R M W_0
7 0 0 0 0 0 1 0 0
core 0 tests and acquires lock but does not wake up any other core (write$_{CB0}$)

Figure 6: Optimizing RMW with write$_{CB0}$: no callback is serviced when the semantics of a write (in this case RMW taking a lock) imply that it is unnecessary

2.6. Ensuring Atomicity for RMW Operations

Atomicity for a RMW operation is provided independently of the callback mechanism, since using a callback in a RMW is optional and we should be able to freely intermix callback and non-callback RMWs correctly.

Atomicity is provided by a lock bit in the LLC MSHR that handles the RMW operation. While the MSHR is locked any other operation must be queued until the corresponding write or Unblock (in case the RMW failed and does not write anything). This temporary and short-lived queuing is provided in the LLC controller similarly to any other protocol. Note that in Figure 2, a callback RMW (any callback read for that matter) has to first consult the callback directory and then go to the LLC. In other words, locking in the LLC starts only after the read is allowed to reach the LLC and send back a value to the core. At that point the prompt completion of the RMW is guaranteed. However, the entire RMW can be held off in the callback directory if it sets a callback as shown in Figure 5 and Figure 6.

2.7. Callback Semantics

A callback is designed to optimize spin-waiting so that it blocks between consecutive writes (creation of values). The property of a callback is that it will return either the last written value or it will block waiting for the next (or for a future write in the case of a callback-one entry). The last written value is always returned in the case of a replacement in the callback directory.

Callback semantics are different than Full/Empty semantics [21]. In the latter, designed for producer-consumer synchronization, both reads and writes block: reads block on Empty and writes block on Full. Callback has the semantics of a blocking read; it cannot block writes which proceed unconstrained to update the value, but it can be "delayed" until some future value is created. A callback can consume a value only once and to guarantee forward progress additional support is provided as we will discuss in Section 3.3.

3. Using Callbacks

3.1. Mechanisms for Races in Self-Invalidation Protocols

We assume that private and DRF data follow the VIPS-M protocol: DRF data self-invalidate and/or self-downgrade on synchronization points; while private data are excluded from coherence. We assume that two separate fence instructions, *self-invl* and *self-down*, are available to self-invalidate and self-downgrade the contents of a private cache, respectively.

The protocol for races should include not only atomics as in VIPS-M, but also load and stores, so that all possible race idioms can be covered. To implement races, loads and stores operate directly on the LLC (atomic instructions do so by default, as in VIPS-M). One can achieve this effect by self-invalidating before a load and self-downgrading after a store that are involved in a race. To avoid this heavy-handed approach we assume "load-through," and "store-through" instructions specifically for races, that skip the L1 caches and go directly to the LLC, but do not cause the self-invalidation or self-downgrade of any other address. Separate fences are needed in this case to enforce desired orderings.

In this work we add a few more variants of such instructions to optimize spinning. Racy loads can be of two types: Load-through (`ld_through`) and Load-callback (`ld_cb`). Both types bypass the L1, since they need to get the value of the last write in an invalidation-less protocol. The only difference is that the load-callback waits at the callback directory for a write if its corresponding F/E bit is empty.

Racy stores can be of three types: Store-callback0 (`st_cb0`), Store-callback1 (`st_cb1`), and Store-through or Store-callbackAll (`st_through`). All of them perform the write-through immediately, and wake up 0, 1, or all callbacks waiting at the callback directory.

Atomics are composed of a load-through and store-through performed atomically. Either the load or the store can be one of the previous types. To keep the name short, we denote them as `{ld|ld_cb}&{st_cb0|st_cb1|st_cbA}`. Table 1 lists all the types with an example of where they are used.

3.2. Memory Consistency

The SC for DRF model allows non-conflicting reads and writes to be freely reordered, that is, all these reorderings are allowed: R→R, R→W, W→R, and W→W. However, if these memory accesses are separated by synchronization, reordering is not allowed. Enforcing order between DRF accesses across synchronization is achieved by using the `self-invl` and `self-down` fences.

More specifically, we model a weaker model, release consistency (RC) [10], that defines acquire and release semantics for synchronization. These semantics order reads with respect to an acquire (ACQUIRE→R) and writes with respect to a release

Operation	Example and Comments
ld_through	General conflicting load. First load in spin-waiting. LLC responds immediately. Resets the F/E bit (see Sec. 3.3).
ld_cb	Subsequent (blocking) loads in spin-waiting. Waits for F/E bit to be full. Resets F/E bit.
st_cb0	Not used. Does not service any callbacks.
st_cb1	Lock release. Services one callback.
st_through (or st_cbA)	General conflicting store. Barrier release. Services all callbacks.
{ld}&{st_cb0}	Test&Test&Set to acquire a lock and enter a critical section.
{ld}&{st_cb1}	Fetch&Add to signal one waiting thread.
{ld}&{st_cbA}	Fetch&Add in a barrier.
{ld_cb}&{st_cb0}	Spin-waiting on Test&Set to acquire a lock and enter a critical section.
{ld_cb}&{st_cb1}	Not used.
{ld_cb}&{st_cbA}	Not used.

Table 1: Synchronization primitives for self-invalidation

(W→RELEASE). The fence instructions we consider enforce the same ordering of accesses across them: `self-invl`→R and W→`self-down`.[7]

Regarding the conflicting accesses that we use for synchronization, load_through, store_through, and atomics, they are coherent and sequentially consistent among themselves. They bypass the L1 and go directly to the LLC where they meet at a single point and update atomically or read. Thus, they are coherent because cores cannot observe different write orders, only the order in which writes reach the LLC. They are SC because, in addition to the above, load_throughs, store_throughs, and atomics are blocking so no later "_through" operation or atomic can be initiated until they complete.

The callback variants of these accesses do not change the memory model. When a callback read is held back in the callback directory waiting for the next write, we are effectively changing the interleaving of accesses while still respecting program order. The effect is the same as if cores that have issued a callback are completely stalled until the next write. Similarly, a write$_{CB1}$ has the effect of further stalling all waiting cores except one, and a write$_{CB0}$ has the effect of prolonging the stall of all waiting cores until (at least) the subsequent write.

Atomicity for RMW instructions is provided by locking in the LLC MSHRs and LLC controller as we described previously. It is because of their SC semantics and atomicity for RMW that these LLC operations can be collectively employed for the synchronization in SC for DRF protocols.

3.3. Forward Progress

As we have discussed so far, a callback load blocks if it has consumed the previous value (setting its F/E bit to "empty" and/or clearing its CB bit). It is clearly intended to be used just for spinning. But what happens when we want to consume the same value more than once? This situation may occur with back-to-back spin loops as in the example in Figure 7.

[7]A `self-invl` fence also performs a self-downgrade of all the transient dirty blocks in order to invalidate them. As a result it also enforces W→`self-invl`

```
flag = 1;                     | while(flag == 0);
                              | while(flag == 0);
```

Figure 7: Example of callback deadlock

If the flag is read with callback reads, then the final callback that exits the first spin-loop consumes the value 1 and leaves the callback entry "empty" and without a callback. The next callback (the first of the second spin-loop) cannot see that the correct value is already in the L1 (since it skips the L1) and will immediately block in the callback directory. Yet, the correct value that we are seeking for exiting the second spin-loop is already there and no new value will be written to unblock the callback! This happens because two consecutive callbacks from the same core wish to consume the same value without an intervening write. This situation leads to deadlocks and forward progress is forsaken.

For this reason, a callback spin-loop is always preceded by a `load_through` that checks whether the value we want is already there, regardless of whether the value was consumed in the past. A `load_through` has the behavior of a *non-blocking* callback. It consumes a value if one is available (i.e., sets the F/E to "empty" if it was "full") but does not block and returns the current value of the data if there is no new value (i.e., F/E bit previously set to "empty"). In the same vein, if any spin loop is interrupted for any reason it must restart from the `load_through` (or *with* a `load_through`) as a thread migration may have ensued.

3.4. Synchronization Algorithms

With all of the above in mind, in the rest of this section we compare the encodings of the most frequent synchronization idioms (adapted directly from the code in [1]) for an SC (unfenced) MESI implementation, a fenced VIPS-M implementation with LLC spinning, and a fenced VIPS-M using callbacks.[8]

3.4.1. Test&Set Lock Algorithm. Figure 8 shows a simple Test&Set spin-lock algorithm. The code for MESI is shown on the left and a fenced (`self-invl, self-down`) VIPS-M code on the right. Code using callbacks is shown in Figure 9. A callback-all implementation is shown on the left and a callback-one on the right. We use a single non-callback T&S (`{ld}&{st_cb0}`) as a guard just before the callback T&S (`{ld_cb}&{st_cb0}`) spin-loop (label `tas:`), as per our discussion in Section 3.3. If this succeeds we enter the critical section, otherwise the callback T&S sets a callback and waits in the spin-loop.

3.4.2. Test-and-Test&Set Lock Algorithm. Figure 10 (MESI:left, VIPS-M:right) shows the more common Test-and-Test&Set (*T&T&S*) lock algorithm [24]. The first "Test" is implemented with a `ld_through` in VIPS-M. In the callback implementations in Figure 11 (callback-all on

[8]Because of space constraints we cannot include the original high-level code from [1]. We suggest consulting this high-level code for an easier understanding of our examples.

```
acq:  t&s $r, L, 0, 1
      bnez $r, acq
      /* CS */
rel:  st L, 0
```

```
acq:  t&s $r, L, 0, 1
      bnez $r, acq
      self_invl
      /* CS */
rel:  self_down
      st_through L, 0
```

Figure 8: Test&Set algorithm for MESI and VIPS-M

```
acq:  ld&st $r, L, 0, 1
      beqz $r, cs
spn:  ld_cb&st $r, L, 0, 1
      bnez $r, spn
cs :  self_invl
      /* CS */
rel:  self_down
      st_through L, 0
```

```
acq:  ld&st0 $r, L, 0, 1
      beqz $r, cs
spn:  ld_cb&st0, $r, L, 0, 1
      bnez $r, spn
cs :  self_invl
      /* CS */
rel:  self_down
      st_cb1 L, 0
```

Figure 9: Test&Set callback algorithm

the left and callback-one on the right), we use `ld_cb` as the first test and the non-callback versions of the atomic T&S: `{ld}&{st_cbA}` and `{ld}&{st_cb1}`. Only the first "Test" blocks in the callback directory and a guard `ld_through` (see Section 3.3) precedes the `ld_cb`.

3.4.3. CLH Queue Lock Algorithm. Figure 12 (MESI:left, VIPS-M:right) shows the CLH queue lock algorithm [1, 8, 17]. The atomic in this case is an unconditional `fetch&store`. The callback implementation is shown in Figure 13. This algorithm only has one thread spinning on a variable, so callback-all and callback-one behave in the same way. The spin-loop is in all cases a simple `ld` which is replaced in the callback implementation by a guard `ld_through` and a spin-loop with `ld_cb`.

3.4.4. Sense Reversing Barrier. Figure 14 (MESI:left, VIPS-M:right) shows the sense reversing (*SR*) barrier [1, 24]. Code using callbacks is shown in Figure 15. The atomic used is a non-callback `fetch&decrement` of the form `{ld}&{st}`. In this algorithm all threads waiting for the barrier spin until the sense changes, so a callback-all works efficiently.

3.4.5. Tree Sense Reversing Barrier. Figure 16 (MESI:left, VIPS-M:right) shows the simple scalable tree-based (sense reversing) barrier (*TreeSR*) [1, 19]. Code using callbacks is shown in Figure 17. In this algorithm there are two spin-loops (labels `bar:` and `spn:`) and no atomics are used. Only one

```
acq:  ld $r, L
      bnez $r, acq
      t&s $r, L, 0, 1
      bnez $r, acq
      /* CS */
rel:  st L, 0
```

```
acq:  ld_through $r, L
      bnez $r, acq
      t&s $r, L, 0, 1
      bnez $r, acq
cs:  self_invl
      /* CS */
rel:  self_down
      st_through L, 0
```

Figure 10: Test-and-Test&Set algorithm for MESI and VIPS-M

```
acq:  ld_through $r, L
      beqz $r, tas
spn:  ld_cb $r, L
      bnez $r, spn
tas:  ld&st $r, L, 0, 1
      bnez $r, spn
cs :  self_invl
      /* CS */
rel:  self_down
      st_through L, 0
```

```
acq:  ld_through $r, L
      beqz $r, tas
spn:  ld_cb $r, L
      bnez $r, spn
tas:  ld&st0 $r, L, 0, 1
      bnez $r, spn
cs :  self_invl
      /* CS */
rel:  self_down
      st_cb1 L, 0
```

Figure 11: Test-and-Test&Set with callbacks

```
// $l = lock pointer (L); $i = my node pointer (I)
```

```
acq:  st $i->succ_wait, 1
      f&s $p, L, $i
spn:  ld $r, $p->succ_wait
      bnez $r, spn
      /* CS */
rel:  ld $p, $i->prev
      st $i->succ_wait, 0
      st I, $p
```

```
acq:  st_thr. $i->succ_wait, 1
      f&s $p, L, $i
spn:  ld_thr. $r, $p->succ_wait
      bnez $r, spn
cs :  self_invl
      /* CS */
rel:  self_down
      ld $p, $i->prev
      st_thr. $i->succ_wait, 0
      st I, $p
```

Figure 12: CLH queue lock algorithm for MESI and VIPS-M

```
...
try:  ld_through $r, $p->succ_wait
      beqz $r, si
spn:  ld_cb $r, $p->succ_wait
      bnez $r, spn
cs :  self_invl
...
```

Figure 13: CLH queue lock algorithm with callbacks

thread waits for the value to be modified in both spin-loops, so any callback (callback-all or callback-one) works efficiently. The example is coded with callback-all.

3.4.6. Signal/Wait and Spin Waiting on Flags. Figure 18 (MESI:left, VIPS-M:right) shows a signal/wait synchronization. Each signal wakes up only one thread. Code using callbacks in Figure 19. The signal-wait only wakes up one thread, so callback-one is the efficient solution. In the case of spin waiting on flags, the safe way is to use callback-all, but if the programmer knows that only one thread should wake up, callback-one is more efficient.

4. Related Work

Our work, of course, builds on a number of previous works that paved the way for simplifying coherence [7, 12, 13, 22, 23, 25, 26]. We use a VIPS-M protocol [23] as a representative SC for DRF protocol with self-invalidation and self-downgrade. VIPS-M suffers from the inability to efficiently handle data races intended for synchronization (spin-waiting) as we discussed in previous sections. In this work we address this shortcoming for SC for DRF protocols.

```
// $s = local sense (L); $p = number of processors
```

```
bar:  not $s, $s
      f&d $c, C
      bne $c, 1, spn
      st C, $p
      st S, $s
spn:  ld $r, S
      bne $r, $s, spn
```

```
bar:  not $s, $s
      self_down
      f&d $c, C
      bne $c, 1, spn
      st_through C, $p
      st_through S, $s
spn:  ld_through $r, S
      bne $r, $s, spn
si :  self_invl
```

Figure 14: Sense reversing barrier for MESI and VIPS-M

```
bar:  not $s, $s
      self_down
      f&d $c, C
      bne $c, 1, sw
      st_through C, $p
      st_through S, $s
try:  ld_through $r, S
      beq $r, $s, si
spn:  ld_cb $r, S
      bne $r, $s, spn
si :  self_invl
```

Figure 15: Sense reversing barrier with callbacks

```
// $s = sense; $r = child not ready (R);
// $h = have child; $p = parent ptr; $c = child ptr
```

```
bar:  ld $r, R
      bnez $r, bar
      st R, $h
      st 0($p), 0
      bnez $pid, sen
spn:  ld $r, $p->sense
      bne $r, $s, spn
sen:  st 0($c), $s
      st 1($c), $s
      not $s, $s
```

```
bar:  self-down
      ld_through $r, R
      bnez $r, bar
      st R, $h
      st_through 0($p), 0
      bnez $pid, sen
spn:  ld_through $r, $p->sense
      bne $r, $s, spn
sen:  self-invl
      st_through 0($c), $s
      st_through 1($c), $s
      not $s, $s
```

Figure 16: Tree sense reversing barrier for MESI and VIPS-M

```
bar:  self-down
      ld_through $r, R
      beqz $r, res
      ld_cb $r, R
      bnez $r, bar
res:  st R, $h
      st_through 0($p), 0
      bnez $pid, sen
try:  ld_through $r, $p->sense
      beq $r, $s, sen
spn:  ld_cb $r, $p->sense
      bne $r, $s, spn
sen:  self-invl
      st_through 0($c), $s
      st_through 1($c), $s
      not $s, $s
```

Figure 17: Tree sense reversing barrier with callbacks

The DeNovoND work of Sung et al. [26] is also close to our work. We see our works as complementary. DeNovo [7], and by extension DeNovoND and DeNovoSync, starts from a disciplined programming language to drive coherence; we aim for the general case with as little interaction with the software as possible (i.e., limited to exposing synchronization to the hardware). One of the main contributions of Sung et al. is to implement a "synchronization" protocol for critical sections, that was lacking in DeNovo. For this, DeNovoND relies on a specialized hardware implementation of queue locks. For critical sections, a form of immediate notification is needed to signal changes on data. Without direct invalidations this is accomplished by tracking each core's changes (while in the critical section) and conveying them to the next core that enters the critical section. Changes encoded in a signature, are compounded from core to core in the same order the lock is taken. In essence, Sung et al. extend the work of Ashby et al. [5] from barrier synchronization to critical section synchronization. However, the DeNovoND approach is bound to a specific lock implementation, which carries a significant complexity and cost and is overloaded with tracking invalidation signatures.

```
sig:  f&i $c, C
...
spn:  ld $c, C
      beqz $c, spn
tad:  t&d $r, C
      beqz $r, spn
```

```
sig:  self_down
      f&i $c, C
...
spn:  ld_through $c, C
      beqz $c, spn
tad:  t&d $c, C
      beqz $c, spn
      self_invl
```

Figure 18: Signal-wait for MESI and VIPS-M

```
sig:  self_down
      ld&stA $c, C
...
try:  ld_through $c, C
      bnez $c, tad
spn:  ld_cb $c, C
      beqz $c, spn
tad:  ld&st0 $c, C
      beqz $c, spn
      self_invl
```

```
sig:  self_down
      ld&st1 $c, C
...
try:  ld_through $c, C
      bnez $c, tad
spn:  ld_cb $c, C
      beqz $c, spn
tad:  ld&st0 $c, C
      beqz $c, spn
      self_invl
```

Figure 19: Signal/wait with callbacks

Our focus is different. We propose a low-level mechanism to assist spin-waiting thus covering both locking and spin-waiting on flags, and therefore a variety of algorithms implemented using these primitives, while DeNovoND focuses on a single high-level lock construct.

4.1. Hardware-Assisted Spin-Waiting

`Quiesce` instructions [9] found in Intel, Alpha, and other processors, use cache coherence to implement functionality reminiscent of a callback (specifically the *callback-all*) mechanism. Invariably, instructions of this type halt the execution of the program until an event occurs. This requires an event monitor next to the core (that has to be explicitly "armed") to check for the occurrence of an event. In the case of spin-waiting, the event is a write signaled by an invalidation that reaches the L1. Another related approach is the *lock-box* mechanism by Tullsen et al. [27] which takes the event monitor inside the core to detect events among SMT threads. The fundamental difference between such approaches and ours is that event monitors cannot "detect" a write prior to the arming of the mechanism. In other words, they have no concept of a value already present for consumption, as we do with the F/E functionality. This is why event monitors *must* have an abort time-out mechanism that introduces unnecessary latency in this case. In addition, the functionality of the callback-one (`st_cb1` and `st_cb0`) *cannot be replicated* in a quiesce mechanism.

While the goal of these approaches is the same as ours, we propose callbacks for a system that lacks directory invalidations and forwardings where prior proposals are not applicable. Further, we show that a small and self-contained directory cache (not backed up by memory) handles callbacks with ease without introducing any complexity to deal with evictions.

5. Evaluation

5.1. Simulation Environment

We use Wisconsin GEMS [18], a detailed cycle-accurate simulator for multiprocessor systems. We model a chip multiprocessor comprised of 64 in-order cores. The interconnect is fully modeled with the GARNET network simulator [4]. Table 2 shows the main parameters that define our system. Energy consumption is modeled with the CACTI 6.5 tool [20], assuming a 32nm process technology.

The evaluation covers a wide variety of parallel applications. In particular, we evaluate the *entire* Splash-2 suite [28] with the *recommended* input parameters. Additionally, we run

Parameter	Value
Block and page size	64 bytes and 4KB
Private L1 cache	32KB, 4-way
L1 cache access time	1 cycle
Shared L2 cache	256KB per bank, 16-way
L2 cache access time	Tag: 6 cycles; tag+data: 12 cycles
Callback directory	4 entries per bank (1 cycle)
Memory access time	160 cycles
Network topology	8×8 2-dimensional mesh
Routing technique	Deterministic X-Y
Flit size	16 bytes
Switch-to-switch time	6 cycles

Table 2: System parameters

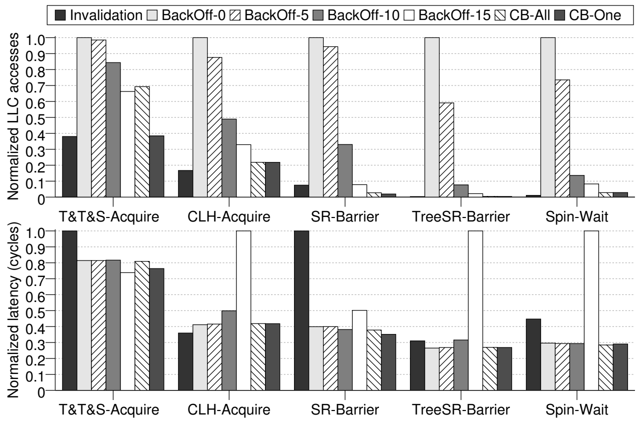


Figure 20: Effect of callbacks on synchronization.

several benchmarks from the PARSEC benchmark suite [6], all of them with the *simmedium* input, except *streamcluster* that uses the *simsmall* input due to simulation constraints. We simulate the entire application, but collect statistics only from start to completion of their parallel section.

5.2. Configurations Evaluated

The goal of our evaluation is to assess the impact of the callback mechanism. We use two base cases. One is a conventional invalidation-based, directory-based MESI protocol (*Invalidation*). The other is a simple protocol using self-invalidation and self-downgrade similar to VIPS-M [23] but with acquire and release semantics, so self-invalidation is necessary only on acquire, and self-downgrade only on release operations. Further, loads used for synchronization (racy loads) always bypass the L1 cache, and racy stores always perform a write-through operation without any delay. In essence, we employ the synchronization algorithms described in Section 3.

VIPS-M avoids spinning on the LLC with an exponential back-off mechanism for racy loads. This mechanism must be tuned to obtain a good trade-off between LLC accesses and latency, as shown in Figure 1. By experimentation we found that an exponential back-off algorithm with approximately 10 exponentiations and a ceiling fits the performance numbers reported in [23]. However, we show results for four different number of exponentiations before the ceiling: 0 (*BackOff-0*), 5 (*BackOff-5*), 10 (*BackOff-10*), and 15 (*BackOff-15*).

We evaluate both the callback-all (*CB-All*) and the callback-one (*CB-One*) mechanisms with just four entries per bank (64 in total) for the callback directory. We simulated more than 4 (16, 64, and 256) entries per bank *without any noticeable change in our results*.

Finally, we use the combination of T&T&S and SR barrier (naïve synchronization) or CLH and TreeSR barrier (scalable synchronization), unless otherwise noted. Following the Splash-2 POSIX implementation of the SR barrier, in the evaluation we use a lock to atomically decrement the barrier counter and not a single atomic as shown in Figure 14.

5.3. Synchronization Behavior

Figure 20, extends our motivation graph (Figure 1) and shows the behavior of all the analyzed synchronization algorithms (T&T&S, CLH, SR barrier, and TreeSR barrier, and the wait procedure of a signal/wait) for all the techniques (*Invalidation*, exponential back-off, and callbacks). Results are normalized to the *highest* result for each synchronization algorithm.

LLC accesses: Exponential back-off techniques dramatically increase the number of accesses to the LLC for all algorithms, even for the largest number of exponentiations. Callback-all and callback-one obtain similar results for all constructs except for the T&T&S acquire (where only callback-one approaches *Invalidation*) and the SR barrier (which uses the T&T&S). The reason in both cases, as explained in Section 2.4, is that callback-all wakes up all threads, but only one can go into the critical section.

Latency: Interestingly, *Invalidation* is outpaced by all other techniques for the naïve synchronization algorithms (T&T&S and SR barrier). This behavior is mainly caused in highly contended T&T&S locks, where the `t&s` operation has to invalidate all copies requested by other threads in an invalidation-based protocol, which in turn also affects its companion synchronization the SR barrier. This behavior is absent from the scalable CLH locks and less pronounced in the TreeSR barrier.

5.4. Execution Time, Network Traffic, and Energy

Figure 21 show execution time and network traffic for all 19 benchmarks running on 64 cores with scalable synchronization (CLH and TreeSR barrier). Results are normalized to *Invalidation*. Overall, we see that the self-invalidation variants, either with exponential back-off or with callbacks, in many cases outperform *Invalidation*.

5.4.1. Execution Time and Network Traffic. From the protocols with exponential back-off, one has to find a very fine balance (in the number of exponentiations) in order to outperform *Invalidation* in *both* traffic and execution time. For the cases we studied, *BackOff-10* seems to strike this balance for most benchmarks. The more aggressive *BackOff-15* misses the target in execution time, and the more tame *BackOff-5* cannot reduce the traffic below *Invalidation* in many cases.

The callback variants achieve as good execution time as the best exponential back-off case (even compared to *BackOff-0*

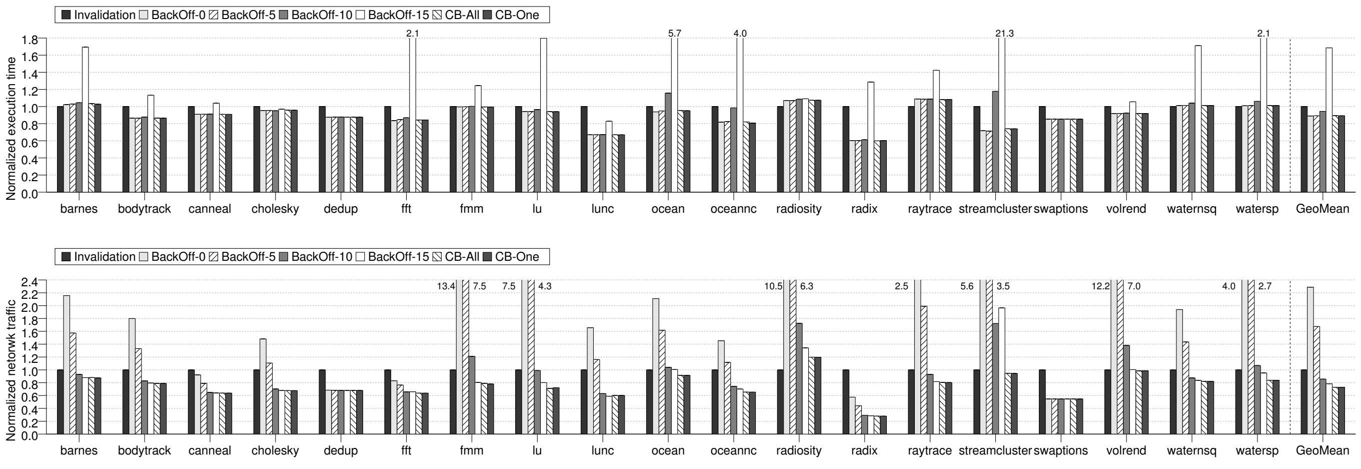


Figure 21: Normalized execution time and network traffic for 64 cores CLH and Tree

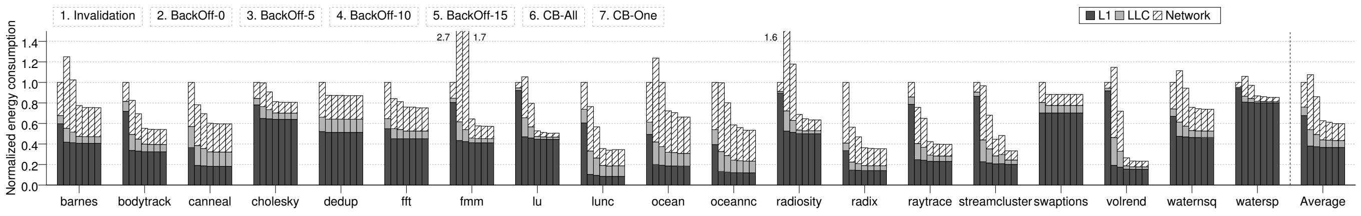


Figure 22: Normalized energy consumption for 64 cores CLH and Tree

which is direct LLC spinning), while at the same time incur on average 7% *less* traffic than the best-in-traffic exponential back-off case (*BackOff-15*). The difference between callback-all and callback-one is not visible here since the scalable synchronization spinning is performed by only one thread per variable.

Overall callbacks achieve: i) 11% better execution time than *Invalidation* and ii) 5% better execution time than *BackOff-10*, which is a configuration similar to but better than VIPS-M (recall that we use acquire-release semantics for fencing). In terms of traffic a callback-enhanced self-invalidation protocol can outpace an invalidation protocol by 27% and a VIPS-M exponential back-off version (*BackOff-10*) by 15%.

We also experimented with the non-scalable (naïve) version of the synchronization algorithms by running all benchmarks (we cannot show detailed results due to lack of space). As it is evident form Figure 20, T&T&S and SR barrier do not perform particularly well for invalidation in 64 cores. In fact, in this case the callback-enabled self-invalidation protocols outperformed invalidation by 40% in execution time and by 34% in network traffic. Compared to *BackOff-10*, which is the best-in-time exponential back-off version for the naïve synchronization, callbacks obtain similar execution time but saving 12% traffic, on average.

These results are significantly better than the corresponding scalable-synchronization results, which brings up the question of whether callbacks can make up for less scalability in the synchronization algorithms. For this we run another set of

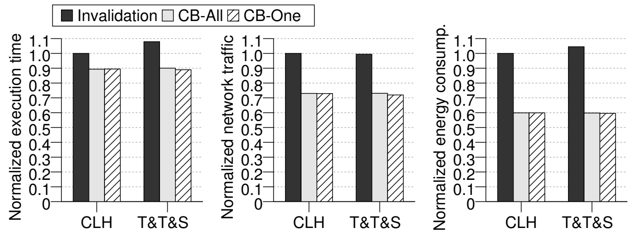


Figure 23: Test-and-Test&Set versus CLHQueue

experiments using the TreeSR barrier and changing only the lock implementation between T&T&S and CLH. The results (geometric mean of the total execution time and total network traffic over all benchmarks) are shown in Figure 23. Indeed, while the effects of scalable synchronization algorithms are visible in invalidation (in execution time), the same is not true for callbacks! In fact, naïve synchronization with callbacks is as good as scalable synchronization with callbacks and enables a self-invalidation protocol to outperform an invalidation protocol in both cases.

5.4.2. Energy. Figure 22 shows energy consumption by converting L1, LLC accesses and network traffic into energy results with the use of Cacti and Garnet respectively. This figure is interesting in that it shows how spinning affects the L1, the LLC, and the network and how the energy consumption "weight" is transferred among them depending on the technique used. *Invalidation* which spins in the L1 (which is relatively more expensive to access than the LLC) shows increased L1 energy consumption, while the exponential back-off versions transfer some of this energy to the (relatively cheaper) LLC and network (for any increase in the LLC we experience a corresponding increase in the network that transfers the requests and responses to and from the LLC). The callback versions minimize all three types (L1, LLC, network) of energy consumption. Overall, callbacks reduce energy consumption by 40% compared to *Invalidation* and by 5% over *BackOff-10*.

6. Conclusions

There is an inherent difficulty in a self-invalidation/self-downgrade protocol to handle spin-waiting. In this case explicit invalidations work better but carry significant complexity. Spinning on the LLC with exponential back-off is a solution but trades traffic for latency and requires precise fine tuning to avoid run-offs in latency. We introduce callbacks as a new solution to a weakness of self-invalidation protocols: intentional data races for spin-waiting. Our callback proposal is general and efficient without the overhead and complexity of including an invalidation protocol alongside with self-invalidation, reverting to specialized and complex hardware queue locking, or trying to fine tune exponential back-off to strike a balance between latency and traffic [5, 23, 25, 26]. Callbacks are efficiently implemented with a small self-contained "callback" directory cache that is not backed-up by memory, avoiding the complexities of replacements and reloads. Our callback proposal retains the valuable properties of self-invalidation protocols (simplicity, low cost, compatibility with virtual caches [13]) while at the same time brings this protocols on par with invalidation protocols in synchronization performance.

Acknowledgments

This work was supported in part by the "Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia" under grant "Jóvenes Líderes en Investigación" 18956/JLI/13, by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03, and by the Swedish VR (grant no. 621-2012-5332).

References

[1] "Sclalable Synchronization Algorithms," http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html.

[2] *C++11: ISO/IEC 14882:2011. ISO. 2 September 2011*, 2011.

[3] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[4] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[5] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers (TC)*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[7] 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 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[8] T. S. Craig, "Building fifo and priority-queuing spin locks from atomic swap," Univerisy of Washington, Technical report FR-35, Feb. 1993.

[9] J. S. Emer, R. L. Stamm, B. E. Edwards, M. H. Reilly, C. B. Zilles, T. Fossum, C. F. Joerg, and J. E. H. Jr., "Method and apparatus to quiesce a portion of a simultaneous multithreaded central processing unit US 6493741 B1," http://www.google.com/patents/US6493741.

[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 Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 15–26.

[11] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," in *3th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 1989, pp. 64–75.

[12] S. Kaxiras and G. Keramidas, "SARC coherence: Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sep. 2011.

[13] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.

[14] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 48–59.

[15] J. Li, J. F. Martínez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2004, pp. 14–23.

[16] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, "Exploiting barriers to optimize power consumption of CMPs," in *19th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 2005, p. 5.1.

[17] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *8th Int'l Symp. on Parallel Processing (IPPS)*, Apr. 1994, pp. 165–171.

[18] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[19] J. M. Mellor-crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[20] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0," HP Labs, Tech. Rep. HPL-2009-85, Apr. 2009.

[21] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 82–91.

[22] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies," in *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 186–197.

[23] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[24] M. L. Scott, *Shared-Memory Synchronization*, ser. Synthesis Lectures on Computer Architecture, M. D. Hill, Ed. Morgan & Claypool Publishers, 2013.

[25] H. Sung and S. V. Adve, "DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations," in *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2015, pp. 545–559.

[26] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient hardware support for disciplined non-determinism," in *18th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2013, pp. 13–26.

[27] D. Tullsen, J. Lo, S. Eggers, and H. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, Jan 1999, pp. 54–58.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.