Efficient Self-Invalidation/Self-Downgrade for Critical Sections with Relaxed Semantics

Alberto Ros, Carl Leonardsson, Christos Sakalis, and Stefanos Kaxiras, *Member, IEEE*

Abstract—Cache coherence protocols based on self-invalidation allow simpler hardware implementation compared to traditional write-invalidation protocols, by relying on data-race-free semantics and applying self-invalidation on synchronization points. Their simplicity lies in the absence of invalidation traffic. This eliminates the need to track readers in a directory, and reduces the number of transient protocol states. Similarly, the use of self-downgrade on synchronization eliminates directory indirection, and hence the need to track writers in a directory. These protocols, effectively without a directory, have the potential to reduce area, energy consumption, and complexity, without sacrificing performance — *provided*, that self-invalidation and self-downgrade are performed prudently.
In this work we examine how self-invalidation and self-downgrade are performed in relation to atomicity and ordering. We show that self-invalidation and self-downgrade do not need to be applied conservatively, as so far implemented. Our key observation is that, often, critical sections which are not ordered in time, are intended to provide only atomicity and not thread synchronization. We thus propose a new type of self-invalidation, *forward* self-invalidation (FSI), which invalidates solely data that are going to be accessed inside a critical section. Based on the same reasoning, we propose a new type of self-downgrade, *forward* self-downgrade (FSD), also restricted to writes in critical sections. Finally, we define the semantics of locks using FSI and FSD, which resemble the semantics of relaxed atomic operations in C++.
Our evaluation for 64-core multiprocessors shows significant improvements using the proposed FSI and FSD —where applicable— in Splash-3 and PARSEC benchmarks, over a directory-based protocol (17.1% in execution time and 33.9% in energy consumption) and also over a state-of-the-art self-invalidation/self-downgrade protocol (7.6% in execution time and 9.1% in energy consumption), while still retaining the design simplicity of the protocol.

Index Terms—Cache coherence; memory consistency; self-invalidation; self-downgrade; critical section; atomicity

1 Introduction and Motivation

Recently, a number of proposals aim to simplify coherence by relying on data-race-free (DRF) semantics and on self-invalidation to eliminate invalidation traffic and the need to track readers at the directory [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. With the addition of self-downgrade, the directory can be eliminated [5] and virtual cache coherence becomes feasible at low cost, without reverse translation [7].

The motivation for simplifying coherence has been established by many [1], [2], [3], [4], [5], [8], [16]. Significant savings in area and energy consumption without sacrificing performance, have been demonstrated in many recent papers [2], [3], [4], [5], [6], [7], [8], [11], [12], [13], [14], [15]. Additional benefits regarding ease-of-verification, scalability, time-to-market, etc., ensue as a result of simplifying rather than complicating such fundamental architectural constructs as coherence. And recently, a Software-DSM system that employs self-invalidation and self-downgrade has been developed showing high scalability [17].

In these coherence protocols, writes to memory are not explicitly signaled to sharers, and the written value will be visible to the sharers when they self-invalidate their local copy. It is straightforward to show that data races throw such protocols in disarray, producing non-sequential-consistent executions [1], [5]. Most of these proposals [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] offer sequential consistency for data-race-free (SC for DRF) programs [18]. Although, conceptually, one could eliminate the requirement for DRF by self-invalidating after every use of data, this would defeat the purpose of caching.

Data-race-free semantics require that conflicting accesses (e.g., a read and a write to the same address from different cores) must be separated by synchronization (perhaps transitive over a set of threads) [18], [19]. Self-invalidation is therefore initiated on synchronization. Conservatively exposing all synchronization to the hardware, causes indiscriminate self-invalidation on locks, barrier, and wait constructs. This is the approach taken by proposals such as SARC coherence [4], VIPS-M [5], RC3 [12], and DeNovoSync [11].

In this paper, we show that self-invalidation does not need to be applied conservatively, as commonly done. Instead, we propose a general approach to reduce the cost of self-invalidation. Our key observation is that lock/unlock synchronization (i.e., ensuring atomic execution of a critical section which respect to other critical sections —weak atomicity [20]—) is often *not* intended to constitute DRF synchronization for its *surrounding code* (Section 3). The execution of such critical sections is therefore unordered in time and cannot successfully separate conflicting accesses that straddle critical sections. This indicates that self-invalidation in a critical section should be restricted only to the data accessed inside it, and should not affect any data accessed

- *Alberto Ros is with the Department of Electrical and Computer Engineering, Universidad de Murcia, Spain.*
  E-mail: aros@ditec.um.es
- *Carl Leonardsson, Christos Sakalis, and Stefanos Kaxiras are with the Department of Information Technology, Uppsala University, Sweden.*
  E-mails: Carl.Leonardsson@it.uu.se, chrissakalis@gmail.com, and stefanos.kaxiras@it.uu.se

prior to its execution.

Contribution 1: Besides the well known *backward* self-invalidation (which invalidates data accessed in the past), we propose an additional type of self-invalidation, *forward* self-invalidation (FSI), which invalidates data that we are going to access in the future. FSI is activated when entering a critical section (i.e., a lock acquire —`lock`—), disabled when exiting (i.e., lock release —`unlock`—), and does not affect unrelated data. Informally, it is intended for critical sections whose execution does not provide to the program information on thread-ordering (Section 4).

Our first contribution is independent of the downgrade approach of the protocol. More specifically, FSI works both in protocols with a directory for the writers (e.g., DeNovo [3], SARC [4] and RC3 [12]) or with self-downgrade (e.g., Ashby *et al.* [2] and VIPS-M [5]). In the first case, we assume the existence of a directory that indirects read misses to the last writer. In the second case, which eliminates the directory altogether, writes are explicitly put in the shared cache without the need of a read request from another core. Our second contribution is specific to the latter.

Contribution 2: We propose *forward* self-downgrade (FSD), similar in spirit to forward self-invalidation, to selectively downgrade data modified inside critical sections (Section 5).

Using similar reasoning as with self-invalidation, we examine the relation of self-downgrade to synchronization. In a critical section —again, providing no information on thread ordering— only data modified between the lock and unlock operations should be made globally visible when exiting the critical section; not necessarily data that were modified prior to the critical section. Conservative, backward, self-downgrade on synchronization reduces write coalescing and increases network traffic and synchronization overhead.

Contribution 3: We define the semantics of locks using FSI/FSD versus conservative self-invalidation/self-downgrade, which describe the guarantees for *atomicity* and *ordering* that are provided by critical sections protected by locks. This semantics resemble relaxed atomic operations in C++ [21], with the main difference that each critical section can contain multiple instructions, complex control flow, and multiple memory accesses (Section 7).

We do not advocate the use of FSI and FSD ubiquitously but rather as a selective optimization. In particular, for our approach to guarantee SC for DRF programs we still require backward self-invalidation and self-downgrade. This happens when data written inside a critical section are used to expose thread-ordering and thus provide thread synchronization in time; in other words, when critical-section execution *confirms* happens-before relationships that regulate (under SC for DRF) when previously written data become visible to other threads. However, if a critical section is intended solely to provide atomicity, FSI and FSD should be employed for optimal performance.

Evaluation: We evaluate our proposal for an extensive set of benchmarks modified appropriately to take advantage of FSI and FSD where possible (Section 8). Our results for 64-core multiprocessors show that our techniques invariably retain the benefits of self-invalidation and self-downgrade, but most importantly, significantly limit the penalties when these appear in synchronization-intensive benchmarks. In particular, we obtain significant improvements over a traditional directory-based coherence protocol (17.1% in execution time and 33.9% in energy consumption) and also over state-of-the-art VIPS-M coherence protocol [5] that employs Callbacks [22] for efficient spin-waiting (7.6% in execution time and 9.1% in energy consumption). More critically, our proposal exploits inherent program properties to further enhance simple and efficient coherence approaches without incurring any additional costs.

2 Cache Coherence based on Self-Invalidation

Self-invalidation cache coherence protocols allow simpler hardware implementation compared to traditional write-invalidation protocols by relying on relaxed consistency and/or a DRF synchronization model for correctness [1], [2], [3], [4], [5], [12]. In a DRF synchronization model, *all* conflicting accesses must be ordered by a happens-before relation dictated by both program order and synchronization order [18]. Further, all synchronization operations must be recognizable by the hardware [18] and therefore can be exposed readily to the cache coherence layer to initiate self-invalidations and/or self-downgrades. More explicitly, upon a synchronization operation with acquire semantics (e.g., lock, barrier, wait) a self-invalidation operation should be performed in absence of write-invalidation actions, and upon a synchronization operation with release semantics (e.g., unlock, barrier, signal) a self-downgrade operation should be performed in absence of directory indirection.

A simple but overly conservative implementation is to perform the self-invalidation (or self-downgrade) of all blocks stored in the processors' private caches. Of course this leads to unnecessary invalidation. A more efficient solution is to perform a selective invalidation of those blocks that may incur conflicts, based on bloom filters [2], touched bits [3], or a private-shared classification [5] (see Section 9 for more details).

The intuition behind this work is that this self-invalidation (and self-downgrade) of cached blocks is still conservative, since the programmer does not require coherence for those blocks on all synchronization points. Our approach, therefore, proposes new semantics for some synchronization constructs (e.g., locks and unlocks) that can be guaranteed with a new type of self-invalidation/downgrade action called *forward*. In essence, this lets programmers selectively specify their requirements for memory consistency and allows to curb the penalty of self-invalidation and self-downgrade, particularly in the cases where it matters most.

3 Ordering vs Atomicity

Ordering. Some synchronization primitives, such as process barriers or signal/wait, are clearly intended to establish order between memory accesses from different threads. The expectation is then that all data written before a synchronization in thread 1 become visible in thread 2 after its corresponding synchronization. Thread 2 is then disallowed from using its stale data found in its cache. Symmetrically, in the case of barriers, any data written by thread 2 after the

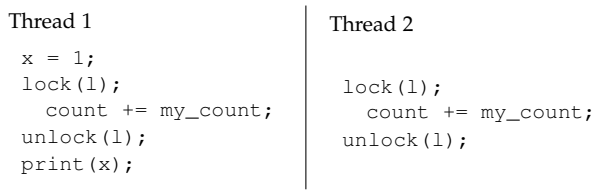


Fig. 1. Critical sections used for atomicity.

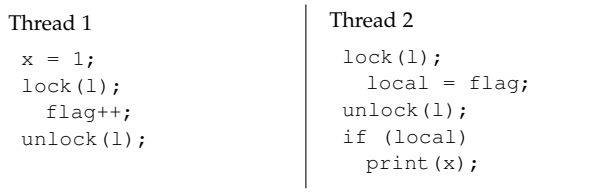


Fig. 2. Critical sections used for ordering.

synchronization will not yet be visible to thread 1 before its synchronization. Such synchronization primitives establish happens-before order between memory accesses, and are often used to accomplish data race freedom [18].

Atomicity. On the other hand, some other synchronization primitives do not inherently establish order. A common example is mutual exclusion locks. Consider the critical sections shown in Figure 1. Assume that we are only interested in performing an atomic read and write of the global variable `count`, but not in enforcing any order between other memory accesses. Assume now that we access data *unrelated* to the critical sections. For example, let us assume that we access the same variable `x` before and after a critical section in thread 1. Should `x` be self-invalidated when entering the critical section (acquire semantics)? The answer is *no*. The reason is that the lock is intended to provide atomicity for the increment of `count`. It is not intended to provide any ordering for `x` or any ordering between thread 1 and thread 2. Therefore self-invalidating `x` is unnecessary, and would hurt performance since it will cause a cache miss when re-accessing `x` after exiting the critical section.

However, it is also possible to write code that detects the order in which different critical sections execute. This knowledge of the execution order can then be used to ensure data race freedom for data accessed *outside* of the critical sections. A typical example is shown in Figure 2. Here the variable `flag` is used to detect the order in which the critical sections execute. The variable `x` is only read by thread 2 when the detected execution order guarantees that the load is not in a data race with the store to `x` by thread 1. Hence, the code in Figure 2 is data race free despite having conflicting accesses to `x` located outside of any critical section.

Thus, we can discern three types of accesses with respect to critical section execution:

- Accesses inside critical sections. The `lock` and `unlock` operations order the conflicting accesses from different critical sections irrespective of the execution order of the critical sections. It is clear that self-invalidation and self-downgrade are needed on entering and exiting a critical section, respectively, so that its accesses will be able to see the newest values created prior to entry [23].
- Accesses outside critical sections where the lock is *not* intended to establish order through observed execution order (e.g., as in Figure 1). The data of such accesses need not be self-invalidated or self-downgraded because of the critical section synchronization.
- Accesses outside critical sections where order is required, and is established by the critical section execution order (e.g., as in Figure 2). The data of such accesses need to be self-invalidated or self-downgraded appropriately.

In this paper, we propose new synchronization primitives for locks, providing weaker consistency. In a critical section, the accesses between lock and unlock are always self-invalidated and self-downgraded. However, the programmer can specify if the accesses outside the critical section should be also self-invalidated or self-downgraded, depending on the semantics of the critical section: pure atomicity or thread-ordering. Hence, our new primitives allow for applications to be optimized by preventing self-invalidation (Section 4) and self-downgrade (Section 5) of variables surrounding critical sections for the case when the locks only provide atomicity. In order to take care of accesses surrounding thread-ordering critical sections, conservative self-invalidation and self-downgrade is employed.

4 Eliding Needless Invalidation: Forward Self-Invalidation

We are now faced with a problem. On one hand, we have to self-invalidate cached data that we are going to access in a critical section between a lock and an unlock operation, but on the other hand this self-invalidation needlessly invalidates unrelated data accessed prior to the critical section and potentially afterwards.

The solution to this problem is to change the way we think of self-invalidation. Normally, we self-invalidate what we have accessed in the past and is already cached. Let us call this backward self-invalidation (BSI).

We propose a second kind of self-invalidation which we call forward self-invalidation (FSI). As the name suggests, from the time of its activation, e.g., on a lock operation, FSI invalidates each cache line that is accessed, exactly once (on its first access). The invalidated cache lines immediately cause misses and need to be re-fetched and cached again. FSI continues until its deactivation, e.g., upon the next unlock operation. The FSI implementation is simple: we use an additional *access* bit per cache line that is set when entering FSI. Accessing a cache line with the FSI bit set, invalidates the cache line and resets the bit. Ending FSI, all the FSI bits are reset. This allows us to limit locking primitives to only enforce consistency for the memory accesses that occur inside their critical section. No additional work is needed to enforce consistency for accesses occurring outside of the critical section.

Since our FSI approach only employs a single bit per cache line, it does not account for false-sharing. For example, imagine that a program accesses a variable (a) before and after the critical section and a variable (b) inside the critical section. If both variables (a and b) belong to the same cache

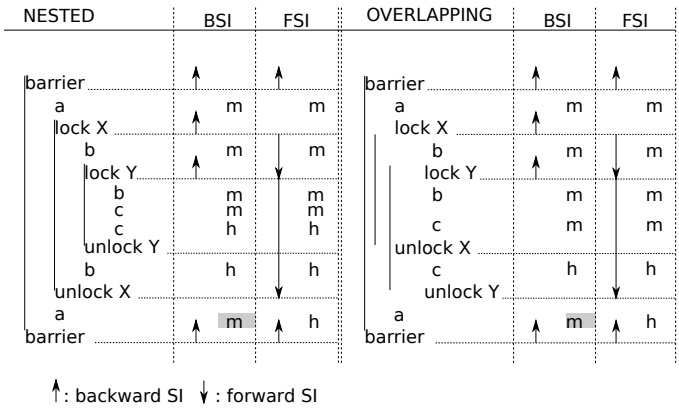


Fig. 3. Self-invalidation strategies in nested (left) and overlapping (right) critical sections.

line, the access to b would invalidate also a, thus causing an extra miss after the critical section. A solution to this extra misses would be to have per-word (instead of per-line) bits in cache. We have not explored this solution because the critical sections commonly consist in few accesses, and therefore, we do not expect many false-positives.

4.1 Nested and Overlapping Critical Sections

We have seen how FSI works for isolated critical sections, but what happens if we have nested or overlapping critical sections?

In this case we cannot just start the FSI at the outermost lock and finish it at the outermost unlock. This can give rise to some strange behaviors due to lack of self-invalidation that can complicate the programming model. One can easily show this by counter-examples, but in the interest of space and clarity we omit this from our discussion.

Our solution is to *restart* the FSI on every lock, but the FSI region ends only if the number of unlock operations matches the number of lock operations. To account for this, we require a single counter initialized to zero which is incremented on lock operations and decremented on unlock operations. The FSI region ends when the value of the counter before the last unlock is "one".

Figure 3 shows how the FSI policy and BSI behave on nested and overlapping critical sections. In this figure the fate of variables "a", "b", and "c" in the cache is shown (miss or hit) depending on the self-invalidation policy used. Backward self-invalidation is indicated by an arrow pointing to the past (upwards) and forward self-invalidation with an arrow pointing to the future (downwards). While BSI is a one-shot operation, FSI is continuous and ends at the tip of the arrow. Shading highlights the transition from a miss to a hit for the different policies.

Nested critical sections (Figure 3, left): BSI performs a self-invalidation of "a" (actually all cache contents) when acquiring the first lock, which will result in a miss on the subsequent access of variable "a." Similarly, when acquiring the second lock, BSI also invalidates "b." FSI avoids to self-invalidate "a," since it is not inside a critical section and the FSI region finishes just before the last unlock. Although variable "b" is self-invalidated on its first access, it also has to be self-invalidated on its second access. This is achieved by resetting the *access* bits when acquiring the second lock. Since there are no more lock operations, the third access to "b," it will result in a cache hit.

Overlapping critical sections (Figure 3, right): This pattern occurs when we are already in a critical section, and a new lock is taken but not released in this critical section. The new lock is typically released in *another* critical section. In the general case, however, the new lock can be released at any point in the code. This is known as "hand-over-hand locking" in literature, and is commonly used, e.g., for fine-grained locking of lists since it requires locking of only two elements of the list, instead of the entire chain from the head to the point of modification. For overlapping critical sections, BSI behaves similar to how it behaves for nested critical sections and is very conservative. FSI improves over BSI again by avoiding the second self-invalidation of "a."

5 Forward Self-Downgrade

In the previous section we discussed self-invalidation without regard to the downgrade policy of the coherence protocol (directory-indirection or self-downgrade). With a directory-indirection policy, read misses go to the directory and obtain the latest value from the writer (or find the latest value in the LLC –last level cache– if it was evicted from the writer). However, a self-downgrade protocol has to explicitly put back all the writes in the LLC *before* crossing a synchronization point: reads cannot indirect to the last writer but expect to find the latest data in the LLC. This downgrade process relies on DRF synchronization (which is already assumed for self-invalidation).

Thus, similarly to self-invalidation, self-downgrade is initiated *conservatively* before every release operation[1] in the program (e.g., unlock, barrier, or signal), ensuring that writes are made globally visible before the corresponding reads. This is overkill. Following the same reasoning as in Section 3, we can show that, when exiting a critical section that it is only intended to guarantee atomicity, we just need to make globally visible the data written inside it:

1) Lock/unlock only need to make visible the accesses inside the critical section that they protect. Exiting a critical section (`unlock`) makes the data written inside it globally visible so that the next critical section that acquires the lock will be able to see them.[2]
2) Critical sections are often not intended to guarantee ordering for their surrounding accesses (Figure 1). This means that data written before a critical section do not have to be visible just because of the execution of the critical section. Thus, there is no need to self-downgrade writes performed before entering a critical section.
3) It is only when there is a separate synchronization point ordering a write and a read, that the write needs to be made visible to the read (Figure 2).

1. Assuming, again, that no differentiation is made between critical sections and other synchronization.

2. *When* the data are actually made visible depends on the consistency model. In Release Consistency [23], as assumed in the above discussion, exiting a critical section makes the writes visible. In Lazy Release Consistency [24] and Entry Consistency [25], entering a critical section makes the writes of the *previous* critical section visible.

Normally, self-downgrade concerns past writes: when we self-downgrade at a synchronization point we make globally visible writes that already happened. Analogously to self-invalidation, let us call this backward self-downgrade (BSD). We define a new type of self-downgrade, called *forward* self-downgrade (FSD), that selects what writes are going to be made visible at the next synchronization point. FSD is also intended to be used in critical sections. It starts on critical section entry (lock) and marks the data written within the critical section. At critical section exit (immediately preceding the unlock), only the marked data are downgraded. FSD does not affect what happens outside the critical section.

An efficient implementation of self-downgrade is based on write-throughs via a coalescing write buffer [5]. The effect of the write buffer is to delay write throughs for a small period of time. For example, VIPS [5] sets an upper limit to the delay of a write through in the write buffer (a timer). When this limit is reached the write through is sent to the LLC. This approach has two advantages. First, multiple writes on the same data that happen closely in time can be coalesced, reducing write-through traffic. Second, write throughs are paced to the LLC throughout execution (as the write buffer ejects older entries to fit new ones), so that at synchronization points only a small and bounded write-through overhead remains: that of emptying the write buffer and writing back the modified data.

In this context, BSD is equivalent to emptying the write buffer on all synchronization points with release semantics. The benefit of FSD, i.e., filtering the writes that go to the LLC during an unlock, is twofold: first it reduces the delay of completing an unlock to a minimum, and second it does not interrupt write coalescing by prematurely emptying the whole write buffer. For example, when executing a small critical section to increment a counter (Figure 1), BSD would unnecessarily empty the whole write buffer on the `unlock` operation (interrupting the coalescing of unrelated data and delaying the operation), whereas FSD would achieve a correct result only by writing through the updated value of `count` on the `unlock` operation.

The implementation of FSD requires a *write* bit per write-buffer entry that is reset when entering FSD. Writing to a cache line sets its *write* bit. Ending FSD, all the entries in the write buffer with the *write* bit set are self-downgraded, and their corresponding *write* bits are reset. Note that during the self-downgrade operation only the dirty data (e.g., words) in each block are writen-back, by sending *diffs* to the next cache level [5].

5.1 Nested and Overlapping Critical Sections

Analogously to FSI, one needs to define the behavior of FSD for nested and overlapping critical sections. The behavior is similar. FSD uses a counter (the same as the FSI policies) that is incremented on `lock` and decremented on `unlock`. As long as this counter is not zero, we mark writes in the write buffer (by using the *write* bit) for immediate downgrade to the LLC at the *next* `unlock`. Every time we encounter an `unlock` operation the self-downgrade is performed for the marked blocks and the write bit is reset. Marking stops when the counter drops to 0. As usual, BSD empties the whole write buffer.

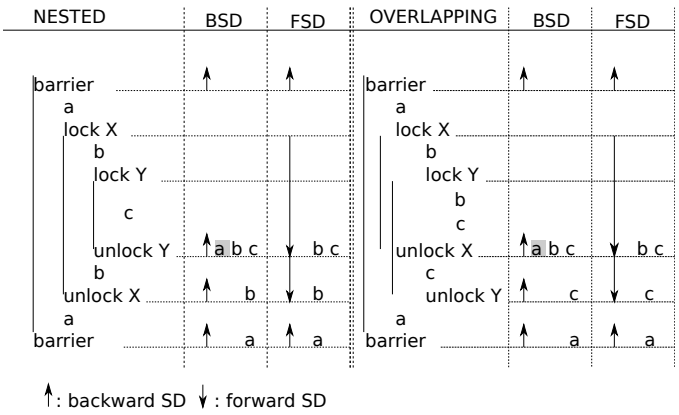


Fig. 4. Self-downgrade strategies for nested (left) or overlapping (right) critical sections.

Figure 4 shows what is made globally visible at various synchronization points, in relation to the writes shown in the code (represented simply with the variable name). The figure considers both BSD and FSD. BSD empties the write buffer on every `unlock`, thus writing back twice variable "a". FSD avoids the first self-downgrade of "a" since the first write is out of the critical section. On the other hand, "b" is written back twice in the case of nested locks and "c" is written back twice in the case of overlapping locks with both policies. This is achieved in the FSD policy by cleaning the *write* bit when a block is self-downgraded.

6 Initiating Self-Invalidation and Self-Downgrade on Synchronization

Synchronization is commonly offered via a language interface, as a set of libraries (e.g., Pthreads), by the operating system, or as part of a programming model. In these cases, the software can easily expose synchronization to the hardware. In Intel Haswel™, locks can be tagged for speculative lock elision by special instruction prefixes (REPN) preceding atomic instructions [26]. The same mechanism can serve for self-invalidation and self-downgrade. More enticing, however, is to recognize synchronization transparently to the software. This has already been proposed in several works: for speculative lock elision for critical-section synchronization [27], for management of multithreaded systems [28], and even for data race detection [29], etc.

Detecting synchronization can be accomplished by the programmer, compiler, or hardware (given the necessary resources [27], [28], [29]). The actual mechanism is beyond the scope of this paper. Here we are concerned with interaction of synchronization with the coherence layer. In particular, our synchronization operations include special instructions that backward/forward self-invalidate/self-downgrade the appropriate memory blocks in the local cache.

Synchronization constructs such as `barrier` and `signal/broadcast/wait` require BSI/BSD. In particular, `BSD` is required just before entering the `barrier` and `BSI` is required when exiting it. `Signal` and `Broadcast` are preceded by `BSD` and `wait` is followed by `BSI`. For a more detailed description about where these instructions are placed we refer the reader to our previous work [22].

```
lock(l);
  BSI;

  /* CS */

  BSD;
unlock(l);
```

```
lock(l);
  depth++;
  FSIDBegin;

  /* CS */

  FSIDEnd;
  depth--;
unlock(l);
```

Fig. 5. Lock/unlock with both BSI/BSD (left) and FSI/FSD (right).

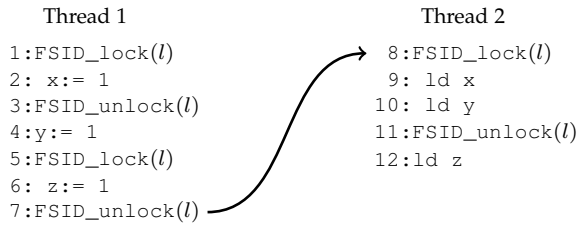


Fig. 6. Example of access orderings which are or are NOT enforced by FSI/FSD locks. In Thread 2, only the load of x is guaranteed to see the value written by Thread 1.

Regarding critical sections, this work opens the possibility of performing either backward or forward self-invalidation/self-downgrade. Figure 5 shows the instructions needed by the lock and unlock primitives for both BSI/BSD and FSI/FSD. In FSI/FSD, `FSIDBegin` acts as both `FSIBegin` and `FSDBegin` if `depth` is 1, but as only `FSIBegin` if `depth` is greater (as discussed in the analysis of nested and overlapping critical sections). Analogously, `FSIDEnd` acts as `FSIEnd` and `FSDEnd` if `depth` is 1, but as only `FSDEnd` if `depth` is greater.

Finally, all memory accesses employed to implement the synchronization operations (e.g., lock/unlock, barrier, and signal/broadcast/wait) that may "race" with other accesses should always bypass the private cache [22]. In the same manner, "racy" stores should perform a write-through directly to the shared cache. This ensures the fast propagation of writes which is essential to achieve efficient synchronization. Although bypassing the private cache for spin-loops can lead to a considerable amount of network traffic and LLC accesses, efficient solutions to this problem have been recently proposed, such as exponential back-off [5], [11], spurious self-invalidation [12], [16], and Callbacks [22]. In this paper, the self-invalidation protocols evaluated implements spin-waiting with Callbacks.

7 Semantics of FSI/FSD Locks

In this section we give an informal description of the semantics of critical sections implementing the FSI/FSD technique as described in Section 6. In essence, the semantics of FSI/FSD locks resemble the semantics of relaxed atomic operations in C++. The key difference is that each critical section can contain multiple instructions, complex control flow, and multiple memory accesses. Therefore, FSI/FSD locks offer a more general solution than relaxed atomic operations.

We will assume here that an FSI/FSD critical section, as described in Figure 5 (right) is initiated by a call to a locking primitive `FSID_lock`(l) and terminated by a call to a corresponding primitive `FSID_unlock`(l) for some lock object l. The implementation of these primitives is assumed to contain the necessary operations for acquiring/releasing the lock, updating the `depth` counter, and initiating/terminating FSI/FSD, as indicated in Section 6. We allow critical sections to be nested or overlapping. In an execution, for a call operation a to `FSID_lock`(l) and a corresponding call b to `FSID_unlock`(l), we consider a and b to define a critical section (l, a, FSID). We consider each operation which is executed between such a and b to be a part of the critical section (l, a, FSID). Correspondingly, we will refer to a critical section for the lock object l, using BSI/BSD synchronization as (l, a, BSID), when it is initiated by the lock operation a. In the discussion, we will let a variable, typically f, take the value FSID or BSID, in critical sections (l, a, f) where the particular synchronization method is not relevant in the context.

We say that a store s to a memory location x is *visible* to a critical section (l, a, f) when (i) for all stores s' to x in the critical section (l, a, f) it holds that s precedes s' in memory order[3], and (ii) when for all loads r' to x in the critical section (l, a, f) it holds that r' loads the value of either s or some memory order-later store. Similarly, we say that a load r to a memory location x is *visible* to a critical section (l, a, FSID) when it loads the value of some store which is visible to (l, a, FSID).

We are now ready to state the intuitive effect of executing the primitives `FSID_lock`(l) and `FSID_unlock`(l).

Executing `FSID_lock`(l) blocks the executing thread until the lock object l is available, and then:

- acquires l, making it unavailable to other threads, and
- initiates a critical section (l, a, FSID), where a is the current lock operation, and
- ensures that every operation that is part of a critical section (l, a', f) for some a' that happened before a (i.e., the lock to a' was taken before the lock to a) in any thread will be visible to (l, a, FSID).

Executing `FSID_unlock`(l):

- terminates the latest currently pending critical section (l, a, FSID) by this thread, and
- ensures that every operation in (l, a, FSID) will be visible to every critical section (l, a', f) for any lock operation a' that happens after a (i.e., the lock to a' will be taken after the lock to a) in any thread, and
- releases l, making it available to other threads.

The main difference between the semantics for FSI/FSD critical sections, as given above, and BSI/BSD critical sections, is that the ordering guarantees made by FSI/FSD critical sections concern only operations which are *inside* the critical sections.

To better understand these semantics, we consider the example given in Figure 6. Note that this example is not DRF, since the write to y in Thread 1 is not protected by lock/unlock. Programmers writing correct DRF code

3. For store operations is the order in which new values are propagated to shared memory.

should not worry about this example. We only show this corner case to make clear which guarantees are or are not given by forward locks. The intuitive understanding of the programmer should basically be that FSID critical sections give atomicity with respect to other critical sections (i.e., weak atomicity [20], as ensured by Pthread lock/unlock critical sections).

In this example, the second critical section of Thread 1 (lines 5-7) is assumed to precede the critical section of Thread 2 (lines 8-11). When the operation `FSID_lock`(l) on line 8 is executed, its semantics guarantee that the operations in earlier critical sections (i.e., lines 2 and 6) are visible to the critical section in Thread 2. Hence, it is guaranteed that the load of `x` at line 9 will read the value 1, written by the store at line 2.

On the other hand, the store to `y` on line 4 is not part of a critical section for the lock object l, and so there is no guarantee that it will be visible for the load on line 10. That load may therefore read either the value 1, written on line 4, or some earlier value.

Symmetrically, the load of `z` on line 12 is not part of a critical section. Hence it is not guaranteed that the store on line 6 will be visible to it. The load of `z` may read either the value 1, written on line 6, or some earlier value.

8 Evaluation

The goal of our evaluation is to assess the impact of forward self-invalidation (FSI) and forward self-downgrade (FSD). We use two base cases. One is a conventional directory-based coherence protocol with MESI states. The other is a state-of-the-art coherence protocol using (backward) self-invalidation on every synchronization with acquire semantics and (backward) self-downgrade on every synchronization with release semantics —we call this protocol BSI-BSD. For an optimal performance, BSI-BSD employs a page-level private/shared classification of the data referenced by the applications performed by the operating system [5], [30], [31] and a Callbacks mechanism for spin-loops [22]. With this double comparison we show that (i) self-invalidation/self-downgrade protocols are a cost-effective alternative for scalable systems and that (ii) the proposed forward self-invalidation/self-downgrade notably improve performance and energy consumption compared to their backward counterparts.

8.1 Simulation Environment

We evaluate the performance of three cache coherence protocols (MESI, BSI-BSD, and FSI-FSD) using the Wisconsin GEMS simulator [32], a detailed simulator for multiprocessor systems. GEMS is fed with information gathered by a PIN tool [33], which offers detailed information about the instructions executed, memory references, and syncronization primitives. Synchronization primitives do not generate a trace, but provide to the simulator its functionality, such that spin-loops are modeled properly in the internals of the simulator. We model an in-order processor that along with the Ruby cycle-accurate memory simulator offers a detailed timing model. The cache coherence protocols evaluated in this work are modeled in detail using the SLICC domain specific language provided by GEMS. The interconnect is modeled with the GARNET network simulator [34]. The simulated system is a 64-core chip multiprocessor with the parameters shown in Table 1. Energy consumption is modeled with the CACTI 6.5 tool [35], assuming a 32nm process technology.

TABLE 1
System parameters.

Parameter	Value
Processor frequency	3.0GHz
Block size	64 bytes
Page size	4KB
Private L1 cache	32KB, 4-way
L1 cache access time	1 cycle
Shared L2 cache	512KB per bank, 16-way
L2 cache access time	Tag 6 cycles; tag+data 12 cycles
Memory access time	160 cycles
Network topology	2D mesh
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data message size	5 flits
Control message size	1 flit
Switch-to-switch time	6 cycles

We employ a wide variety of parallel applications in our evaluations. In particular, we evaluate the entire Splash-3 suite [36] (a modernized, data race free version of the Splash-2 suite [37]) with the recommended input parameters. Additionally, we run a wide range of benchmarks from the PARSEC benchmark suite [38], all of them with the *simmedium* input, except *Fluidanimate* and *Streamcluster*, which use the *simsmall* input due to simulation-time constraints. As recommended, we simulate the entire application, but collect statistics only from start to completion of their parallel section, the region of interest.

We manually identified which critical sections are used for synchronization and which ones are used only for atomicity. To aid us in that task, we employed a data race detection tool similar to the Fast&Furious tool [39], with the main difference being that our tool simulates the relaxed lock and unlock operations by only self-invalidating and self-downgrading the relevant cache lines, as explained in Sections 4.1 and 5.1. After running the applications with our verifying tool more that 1000 times, we did not found any behaviour forbidden by a sequential consistency model, i.e., all loads where returning the same values as a sequential consistency model.

8.2 Results

Results presented in this section are normalized to a directory protocol with MESI states. Applications are shown in increasing order of synchronization intensity: from the ones that do not employ locks (from *Blackscholes* to *Streamcluster*, in increasing order of non-lock synchronization operations) to the ones that execute a larger number of locks (*Fluidanimate* $\approx$ 5.3M locks). Averages are shown for both categories. Notice that, in the *no locks* category, FSI-FSD cannot be applied, and consequently, results are expected to be the

same as with the backward scheme. However, they are also reported for completeness.

8.2.1 BSID vs. FSID synchronization

In order to understand the advantages of FSI-FSD on the evaluated applications it is interesting first to characterize their synchronization operations and the self-invalidation and self-downgrade that they cause. Table 2 shows the number of self-invalidation/self-downgrade pairs of each type executed by each application. The first column shows the application. The second column shows the number of executed BSI-BSD pairs due to barrier and signal/broadcast/wait synchronization (BSID sync). The third column corresponds to the locks that are not relaxed and require BSI-BSD (BSID locks). Finally, the fourth column reports the number of relaxed locks that use FSI-FSD (FSID locks). As shown, most of the applications can safely use FSID locks without breaking the consistency model, which means that these locks were just employed by the programmer to guarantee atomicity. A special case of locks aiming to synchronize threads that can still use FSID, are the locks protecting critical sections that contain conditional variables, such as signal/broadcast/wait (which perform BSI/BSD). Although these locks theirselves do not enforce self-invalidation and self-downgrade of all data in cache, this is guaranteed by the conditional variable. Such critical sections are employed in *Dedup*, *Bodytrack*, *Cholesky*, *FMM*, and *Barnes*.

As mentioned, applications that do not employ locks are not affected by our proposal. On the other hand, the more locks the application executes, the more savings can be expected, given that these locks can use FSI-FSD. This is the case in most applications. The exceptions are *FMM* and *Radiosity*. In particular, in *Radiosity* a large amount of locks require BSI-BSD, which clearly limits the advantages of our proposal. On the other hand, in *FMM* most of the FSID locks contain conditional variables which would neglect the advantages of the FSI and FSD. For applications, such as *Barnes* and *Fluidanimate*, which execute a large amount of locks, all of them being FSI-FSD, significant improvements are expected.

TABLE 2
Pairs of BSI/BSD versus pairs of FSI/FSD instructions executed.

Benchmark	BSID sync	BSID locks	FSID locks
Blackscholes	0	0	0
Swaptions	0	0	0
FFT	320	0	0
Radix	490	0	0
LU	4224	0	0
LU-nc	4224	0	0
Streamcluster	723328	0	0
Water-Sp	832	0	705
Ocean-nc	22976	0	5184
Ocean	23296	0	5184
Dedup	960	0	9068
Raytrace	64	0	13457
Bodytrack	10550	0	15422
Water-Nsq	832	0	34432
Volrend	2688	0	38599
Cholesky	20700	0	81896
FMM	90618	387	135078
Barnes	13931	0	1064144
Radiosity	640	155355	5146016
Fluidanimate	2560	0	5338710

8.2.2 Impact on self-invalidation

The use of self-invalidation can affect the number of cache misses compared to a write-invalidation protocol, which in turn affects execution time. Coherence misses in MESI, caused by remote accesses, are replaced with self-invalidation misses in self-invalidation protocols. Figure 7 shows the breakdown of L1 cache misses for the evaluated applications.

BSI-BSD does well for a variety of benchmarks (*Blackscholes*, *Swaptions*, *FFT*, *Radix*, *LU*, *LU-nc*, *Ocean-nc*, *Volrend*, and *Cholesky*), reducing the number of misses compared to MESI. The number of synchronization points in these benchmarks is low and mostly dominated by barriers and signal/wait (BSID sync). There are two main reasons that can lower the miss ratio of BSI-BSD compared to MESI. First, store race-free operations are always hits in BSI-BSD. Second, store operations do not cause invalidations, which reduces misses due to false sharing.

In others, such as *Raytrace*, *Water-Nsq*, *Bodytrack*, self-invalidation protocols perform similarly to MESI. The reason is that cache misses in these benchmarks are dominated by cold, capacity, and conflict misses.

There are, however, a number of benchmarks where self-invalidation does poorly (*FMM* and *Barnes*) or fails spectacularly (*Radiosity*, and *Fluidanimate*). The commonality in these programs is frequent and intensive synchronization, mainly using locks. It is *exactly* in these cases where forward self-invalidation dramatically betters backward self-invalidation. In fact, forward self-invalidation considerably reduces the number of misses caused by self-invalidation in applications like *Volrend* (32.1%), *Cholesky* (24.2%), *FMM* (16.9%), *Barnes* (63.1%), and *Fluidanimate* (65.4%) with respect to BSI. The only application that is self-invalidation-sensitive and can barely be optimized with FSI is *Radiosity* (0.9% improvements in cache misses). Although almost most of its locks are FSID, in the cases where a FSID lock is followed by a BSID lock, self-invalidation of the cache contents takes place, thus negating the advantages of the FSID lock. In general, the FSI technique is very effective, reducing the total number of cache misses by 12.2% (geometric mean) with respect to BSI for the applications that employ locks.

8.2.3 Impact on self-downgrade

Self-downgrade impacts the number of write-backs to shared memory, which in turn mainly affects network traffic and energy consumption in the cache hierarchy. In MESI, downgrades are caused by replacements or by coherence requests due to remote writes. They also can cause extra cache misses on store operations due to lack of write permission.

In BSI-BSD, self-downgrades never cause extra cache misses, since DRF store operations can always write in cache, but can dramatically increase the network traffic, as happens in the synchronization-intensive applications.

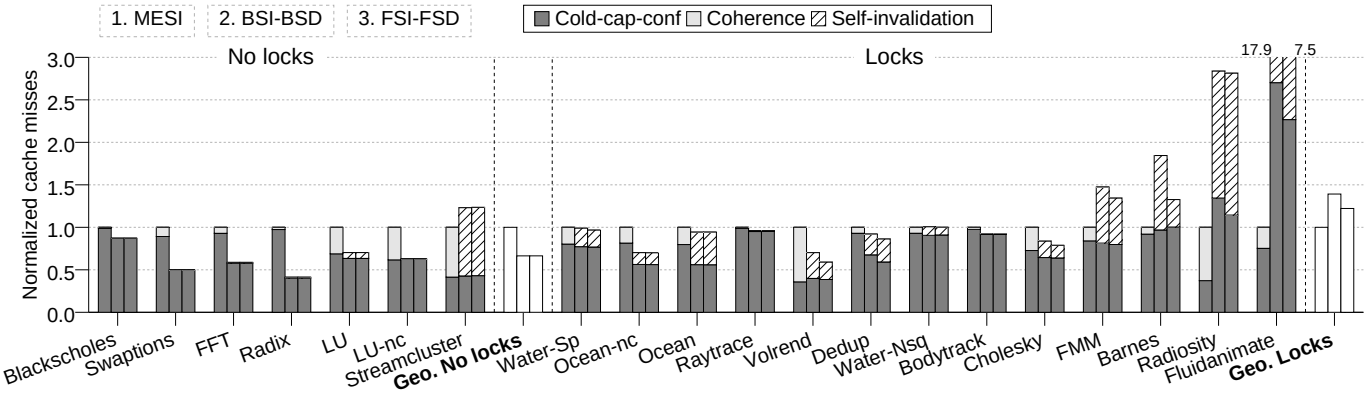


Fig. 7. Normalized L1 cache misses

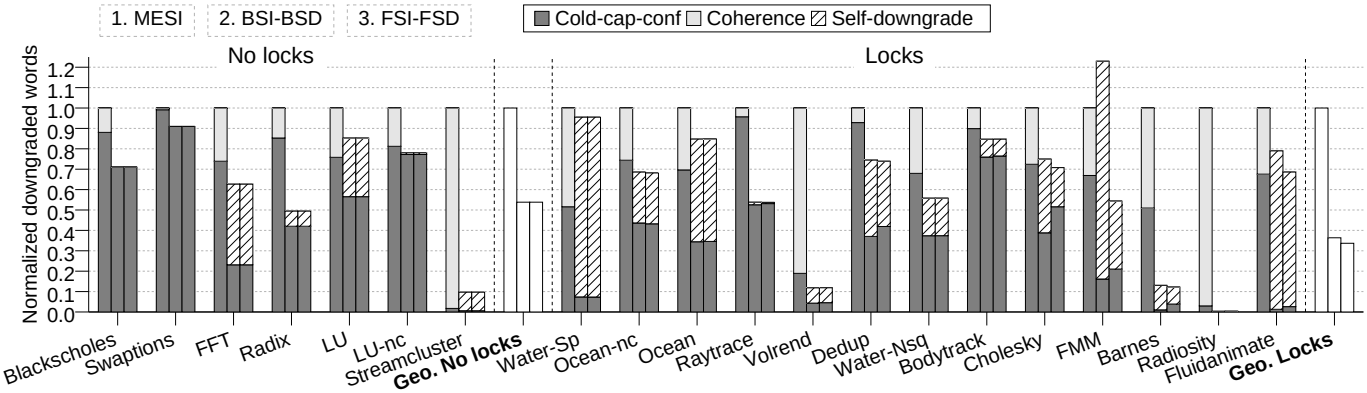


Fig. 8. Normalized downgraded words

Much like FSI does for misses, our forward self-downgrade approach effectively addresses this problem for write-backs by dramatically curbing the damage of backward self-downgrade in the same synchronization-intensive benchmarks. Forward self-downgrade elides the downgrade of written words out of the critical section, thus it does not interrupt write coalescing, as BSD, by perceptually emptying the write buffer on exiting a critical section.

Figure 8 shows the number of downgraded words in the evaluated applications. Downgrades due to coherence requests in MESI are translated to self-downgrades in self-downgrade protocols. Backward self-downgrade (BSD) reduces the number of downgraded words for all the applications except for *FMM*. It is in this application where FSD achieves major benefits, reducing the self-downgraded words by 68.8% with respect to BSD.

FSD achieves reductions in the number of self-downgraded words with respect to MESI for all benchmarks, obtaining an average reduction of 66.4% (geometric mean) for the applications that employ locks. This helps to reduce the energy consumption of the memory system.

8.2.4 Impact on execution time

In Figure 9 we show the combined effect of FSI and FSD on execution time. Variations in misses and downgrades manifest in execution time but in a much more subdued manner. As discussed, the more lock-intensive is the application, the worse performance is expected for backward self-invalidation/self-downgrade protocols. Indeed, *FMM*, *Barnes*, *Radiosity*, and *Fluidanimate* perform similar or worse than MESI when considering BSI-BSD. The rest of the applications perform better than MESI when considering a BSI-BSD protocol.

Because load misses matter significantly more for execution time than downgrades, execution time results are skewed towards the miss behavior of the corresponding techniques. Thus, FSI-FSD outperforms BSI-BSD mostly in the applications where self-invalidation is considerably reduced. In particular, significant reduction with respect to BSI-BSD is achieved in *Barnes* (16.4%) and *Fluidanimate* (39.2%). In *Ocean-nc* and *Ocean*, despite not reducing the overal number of cache misses, performance improvements are obtained. In both applications about 18% of self-invalidations and self-downgrades are turned to forward. The effect is that FSI-FSD reduces the number of invalidated and downgraded blocks at locks which are finally invalidated and downgraded at barriers. Since locks are more critical for performance than barriers, FSI/FSD achieves improvements in execution time. The end result is that, on average, FSI-FSD outperforms MESI for the applications using locks by 17.1% and a highly optimized version of BSI-BSD by 7.6%.

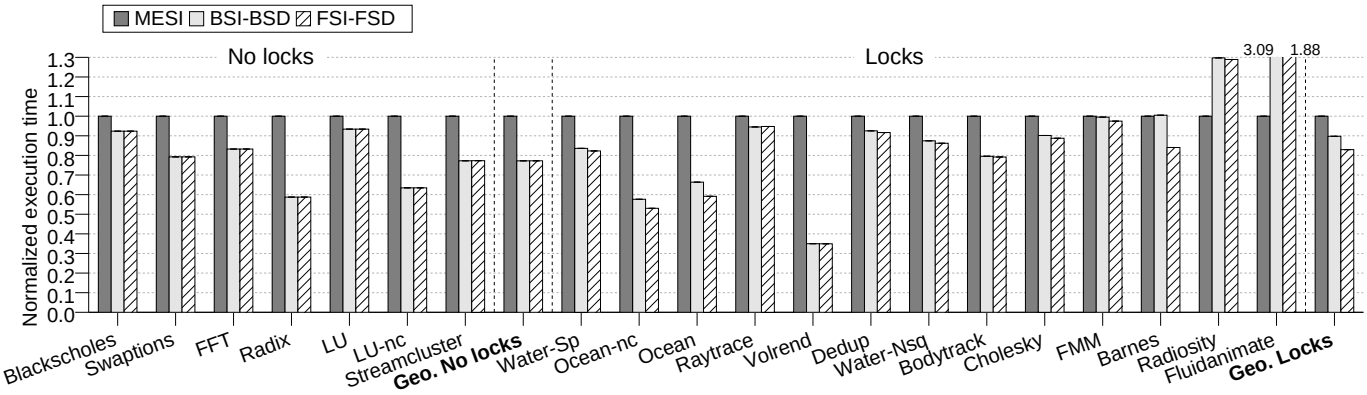


Fig. 9. Normalized execution time

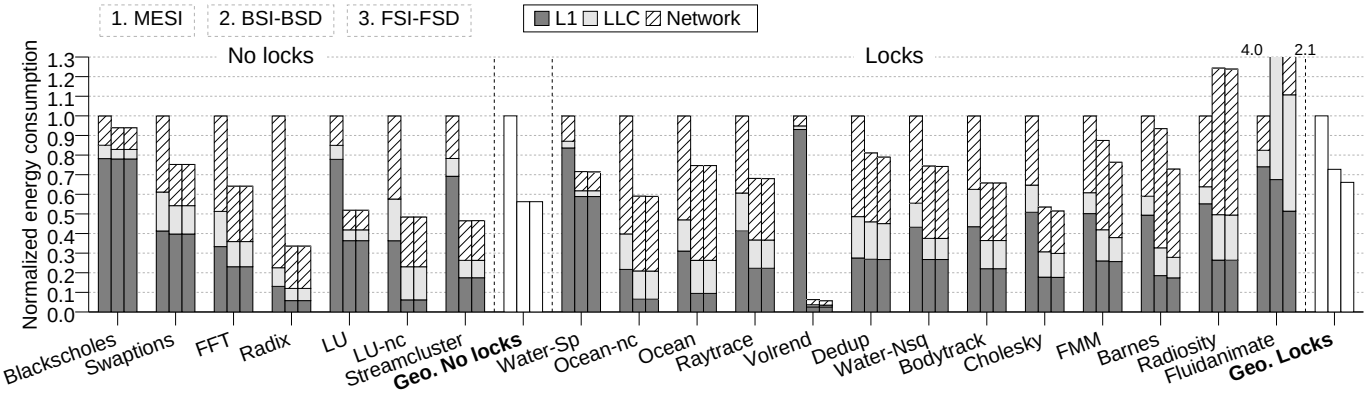


Fig. 10. Normalized energy consumption

8.2.5 Impact on energy consumption

In contrast to execution time, energy consumption tracks both the downgrade behavior and the miss behavior of the corresponding techniques. This is because both misses and downgrades cost in accesses to the shared cache and network traffic. The combined effect is shown in Figure 10, where we break the energy consumption in the consumption of the L1 cache (*L1*), the consumption of the last-level cache (*LLC*) with is the L2 in our system, and the consumption of the interconnection network (*Network*). The only outliers in a backward self-invalidation/self-downgrade protocol with respect to MESI are *Radiosity*, and *Fluidanimate*, but for every other application BSI-BSD does significantly better. FSI-FSD consistently lowers energy consumption over BSI-BSD. More importantly, FSI-FSD works best in the applications that do not do well with BSI-BSD when compared to MESI. Again, the exception is *Radiosity*, which cannot get full use of FSID locks.

Part of the reduction in energy of self-invalidation protocols compared to MESI comes from reducing the spin-waiting in L1, and shifting it to less frequent LLC accesses. However, in BSI-BSD the traffic generated due to extra L1 misses and downgrades lessens this advantage. Thanks to the extra reductions in network traffic in FSI-FSD, the energy consumption is reduced with respect to MESI by 33.9% and with respect to BSI-BSD by 9.1% for the applications that employ locks.

8.2.6 Discussion

To conclude the evaluation it is worth to note that the number of self-invalidation and self-downgrade actions are drastically reduced with FSI and FSD, respectively. The overall improvements in performance, however, are modest for the evaluated system and applications. Nevertheless, in novel Software-DSM systems based on self-invalidation like Argo [17], FSI and FSD can be invaluable, since self-invalidation is a major bottleneck in such systems.

9 Related Work

Our work builds on a number of previous works that paved the way for simplifying coherence [2], [3], [4], [5], [6], [7], [11], [22], [40], [41].

From the VIPS works [5], [7], [22], [42] we borrow the concept of integrating self-invalidation and self-downgrade. However, that work suffers from conservative self-invalidation and self-downgrade especially in synchronization-intensive workloads. In this work we address this shortcoming and improve the overall performance and energy efficiency by exploiting program properties.

The DeNovo work [3], [6], [11], [13] is also close to our work. We see our works as complementary. Below we briefly touch on some details of the related work that matter in our context.

9.1 Invalidation Signatures

The approach proposed by Ashby *et al*. [2] relies on inexact information gathered in Bloom filters to selectively self-invalidate data that *might* have changed but spare data that are known *not* to have changed. Signatures are created at the cores and are communicated to all other cores during barriers. This approach is restricted to barrier synchronization because the Bloom filters cannot be reset easily if it is not known that *every* core has observed all prior invalidation signatures. This problem is solved in DeNovoND for lock synchronization, using a hardware queue lock. A further problem is that it is very expensive to bulk-self-invalidate using a Bloom filter signature because all the tags must be matched against the signature. This either takes excessive time or excessive hardware [2]. In contrast, our approaches are simple and hardware-efficient: backward self-invalidation resets the valid bits of all (shared) cache lines, and forward self-invalidation operates on a per-access basis consulting a single bit (Section 4).

9.2 DeNovo Touched Bits

Touched bits is an interesting concept introduced in the DeNovo work to reduce self-invalidation. Touched bits give a reprieve to cache lines that are touched during an epoch (the period of execution between two barriers) from being self-invalidated at the end of the epoch. In short, if a cache line is accessed in a epoch, it cannot have been changed by another core during that same epoch —otherwise it would be a data race. Touched bits are orthogonal to our approach: i.e., sieving what will be self-invalidated after crossing a barrier. For critical sections, forward self-invalidation subsumes their function as it restricts invalidation to critical section accesses which cannot be the same as accesses preceding the critical section. The touched bit concept requires either data-race-free semantics at a cache-line granularity (since it cannot handle false sharing), or alternatively tracking of touched items on a per-word (as DeNovo does) or per-byte basis in all cache lines (something that is quite expensive). However, our emphasis throughout this work has been to i) propose an approach for the general case (with as few restrictions to the software as possible, i.e., allowing false-sharing) and ii) propose low cost solutions, by not requiring a per-word bit per cache line.

9.3 DeNovoND and Touched-Atomic Bits

One of the main contributions of DeNovoND over DeNovo is the implementation of a "synchronization" protocol for critical sections. DeNovoND recognizes that in a critical section, only data accessed in other critical sections under the same lock should be invalidated (although this is only valid for the "atomicity" locks but not for "ordering" locks, as described in Section 3). Lacking 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 are encoded in a signature and compounded from core to core in the same order the lock is taken. However, the DeNovoND approach needs to track signatures for the accesses in critical sections and transfer the signatures from unlock to lock and needs to explicitly clean signatures which is still a complication. In case signatures are not employed all the accesses performed in within critical regions have to be self-invalidated. Differently, we support general programs by considering both atomicity and ordering locks, identifying them in the code. Additionally, for atomicity locks we opt for invalidating what *will* be accessed within a critical section, and not what was accessed in the critical section in the past.

Touched-atomic bits are introduced by DeNovoND to self-invalidate each block inside a critical section only once. Although FSI bits have a similar effect inside critical sections in our approach, the key of FSI is avoiding invalidation *outside* critical sections. It is actually outside critical sections where FSI obtains improvements with respect to BSI.

Finally, DeNovoSync [11] helps to reduce the self-invalidation overhead of DeNovoND by delaying reads with a backoff mechanism.

9.4 Self-invalidation in GPUs

Some proposals based on self-invalidation have been also proposed for GPUs. TC [43] employs self-invalidation in the private caches based on expiration times. The problem of this approach is the difficulty in predicting expiration times in hardware.

More similar to our work, other proposals perform self-invalidation on acquire synchronization. Sinclair *et al*. [13] reduces the impact of self-invalidation by not invalidating read-only regions of data detected at compile time. Our FSI mechanism would benefit from this optimization in the case of backwards locks, similarly to BSI, thus improving the benefits of our solution with respect to a MESI protocol.

hLRC [15] delays the self-downgrade and the self-invalidation. Atomic variables are tracked by the L2 cache. When an atomic variable changes the owner, the cache losing the ownership needs to perform a self-downgrade while the cache acquiring the ownership needs to perform a self-invalidation. Although this lazy self-invalidation/self-downgrade can remove some invalidations/downgrades, in CPUs running applications with contended locks changing ownership even if the lock is not acquired, self-invalidation can be triggered more frequently.

Finally, other solutions explore self-invalidation-based coherence in heterogeneous CPU-GPU systems [14], where the selective self-invalidation techniques can be less aggressive in the throughput-oriented GPU than in the latency-oriented CPU. In general, FSI-FSD would improve even more the performance obtained by the BSI-BSD techniques used in these approaches.

9.5 Extended-DRF Regions

SPEL [40], [41] is a proposal that defines extended-DRF code regions as multiple DRF regions of code, separated by code that contains synchronization operations, that once aggregated keep the DRF properties. SPEL proposes the identification of these regions at compile time given the

semantics assumed by OpenMP applications. An automatic detection of extended-DRF regions has been recently proposed for more Pthreads applications [44].

In practice, in a extended-DRF region every lock and unlock operation can be relaxed, and therefore, use FSI and FSD. This work therefore proposes new lock constructs that have more relaxed semantics (regarding the code outside the critical section) being able to improve the performance of pure self-invalidation, self-downgrade cache coherence protocols.

9.6 Lock-Based Consistency Models

The consistency model proposed in this paper is based on the idea that order between shared memory accesses is only maintained when enforced using critical sections. This behavior is strictly weaker than that of Release Consistency [23], since in Release Consistency locks can be used to enforce order even between accesses occurring outside of the critical sections.

Iftode *et al*. [45] introduce Scope Consistency. Similarly to our model, Scope Consistency relaxes Release Consistency by enforcing weaker guarantees on ordering for memory accesses occurring outside of critical sections. However our semantics are different from Scope Consistency and, in fact, more relaxed. For example, in Figure 6, the load on line 12 is allowed to read the initial value of z, while under Scope Consistency, it would be forced to read the value written on line 6.

Our model is more similar to Entry Consistency [25], but easier to use for the programmer since it does not require explicit identification of shared data in the program nor its association with particular lock objects.

10 Conclusions

Self-invalidation and self-downgrade are a two-edged sword. On one hand they can cut through complexity and deliver significant savings in area, energy consumption, and execution time, but on the other, if not used frugally, can be damaging. Our goal in this work is to methodically re-examine their use in relation to the synchronization (critical section or barrier and signal/broadcast/wait) of a program. We found that conventional self-invalidation and self-downgrade that affect what happened in the past, can be substantially improved if we turn them forward in time, specifically for critical sections that do not ascertain thread-order and thus allow for this more relaxed behavior, i.e., critical sections with relaxed semantics.

Our evaluation shows that forward self-invalidation and self-downgrade can be safely used, obtaining important performance and energy improvements over traditional and state-of-the-art self-invalidation cache coherence protocols. More importantly, this new approach aggressively addresses the synchronization-intensive workloads where prior self-invalidation and self-downgrade protocols exhibited their greatest weakness while consistently maintaining the advantages in all other workloads. While not every programmer will be able to use the proposed locks as easily as the fully synchronizing ones, they are necessary for achieving the best performance, especially for the people who are implementing the higher level synchronization algorithms.

Acknowledgments

This work has been jointly supported by the "Fundación Séneca – Agencia de Ciencia y Tecnología de la Región de Murcia" under project "Jóvenes Líderes en Investigación" 18956/JLI/13, the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2015-66972-C5-3-R, the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center, and the Swedish VR (grant no. 621-2012-5332). The authors would like to thank SNIC and UPPMAX for using their resources.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 189–200.

[9] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *14th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2014, pp. 427–440.

[10] 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.

[11] 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.

[12] M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 292–304.

[13] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient gpu synchronization without scopes: Saying no to complex consistency models," in *48th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 647–659.

[14] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building heterogeneous unified virtual memories (uvms) without the overhead," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016.

[15] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy release consistency for gpus," in *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–14.

[16] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for tso," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 165–176.

[17] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas, "Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory," in *24th Int'l Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, Jun. 2015, pp. 3–14.

[18] 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.

[19] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[20] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[21] ISO, *ISO/IEC 14882:2015 Information technology — Programming languages — C++*. International Organization for Standardization, 2015.

[22] A. Ros and S. Kaxiras, "Callback: Efficient synchronization without invalidation with a directory just for spin-waiting," in *42nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2015, pp. 427–438.

[23] 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.

[24] P. J. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *19th Int'l Symp. on Computer Architecture (ISCA)*, May 1992, pp. 13–21.

[25] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The midway distributed shared memory system," Carnegie Mellon University, Pittsburgh, PA, USA, Technical report 865207, Jan. 1993.

[26] "Intel(r) architecture instruction set extensions programming reference," *http://software.intel.com/en-us/intel-isa-extensions*, 2012.

[27] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *34th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2001, pp. 294–305.

[28] T. Li, A. R. Lebeck, and D. J. Sorin, "Spin detection hardware for improved management of multithreaded systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 17, no. 6, pp. 508–521, Jun. 2006.

[29] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, "Dynamic recognition of synchronization operations for improved data race detection," in *2008 Int'l Symp. on Software Testing and Analysis (ISSTA)*, Jul. 2008, pp. 143–154.

[30] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 184–195.

[31] 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 Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[32] 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.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[34] 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.

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

[36] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[37] 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.

[38] 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.

[39] A. Ros and S. Kaxiras, "Fast&furious: A tool for detecting covert racing," in *6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA) and 4th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM)*, Jan. 2015, pp. 1–6.

[40] A. Ros and A. Jimborean, "A dual-consistency cache coherence protocol," in *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2015, pp. 1119–1128.

[41] ——, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 11, pp. 3101–3115, Nov. 2016.

[42] S. Kaxiras and A. Ros, "Efficient, snoopless, soc coherence," in *25th IEEE International System-on-Chip Conference (IEEE SOCC)*, Sep. 2012, pp. 230–235.

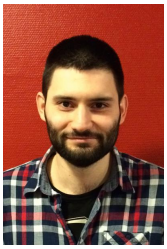
[43] I. Singh, A. Shriraman, and W. W. L. Fung, "Cache coherence for gpu architectures," in *19th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2013, pp. 578–590.

[44] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros, "Automatic detection of extended data-race-free regions," in *15th IEEE / ACM Int'l Symp. on Code Generation and Optimization (CGO)*, Feb. 2017, pp. 14–26.

[45] L. Iftode, J. P. Singh, and K. Li, "Scope consistency: A bridge between release consistency and entry consistency," in *8th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun. 1996, pp. 277–287.

Alberto Ros received the PhD degree in computer science from the University of Murcia, Spain, in 2009, after being granted with a fellowship from the Spanish government to conduct the PhD studies. He hold postdoctoral positions at the Universitat Politècnica de València and at Uppsala University. Currently, he is Associate Professor at the University of Murcia. He has co-authored more than 60 research papers in international journals and conferences. His research interests include cache coherence protocols, memory hierarchy designs, and memory consistency for multicore architectures.

Carl Leonardsson received the PhD degree in computer science from Uppsala University in 2016, specialized in automated verification in the context of relaxed memory.



Christos Sakalis obtained his BSc in Computer Science from Aristotle University of Thessaloniki in 2013 and his MSc in Computer Science from Uppsala University in 2015. He is currently pursuing a PhD at Uppsala University, researching approximate computing. Before that he spend some time at Codeplay (Edinburgh) working with compilers and autovectorization. He is one of the creators of the Splash-3 benchmark suite, a modern version of the well established Splash-2 suite.

Stefanos Kaxiras is a full professor at Uppsala University, Sweden. He holds a PhD degree in Computer Science from the University of Wisconsin. Previously he held positions at Bell Labs (Lucent) and the University of Patras, Greece. Kaxiras' research interests are in the areas of memory systems, and multiprocessor/multicore systems, with a focus on power efficiency. He has co-authored more than 100 research papers, awarded 16 US patents, participated in five major European research projects, and received grants from Sweden's VR and VINNOVA agencies. He is a Distinguished ACM Scientist and IEEE member.