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

Alberto Ros
Universidad de Murcia
aros@ditec.um.es

Carl Leonardsson
Uppsala Universitet
carl.leonardsson@it.uu.se

Christos Sakalis
Uppsala Universitet
chrissakalis@gmail.com

Stefanos Kaxiras
Uppsala Universitet
stefanos.kaxiras@it.uu.se

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 and self-downgrade on synchronization points. This work examines how self-invalidation and self-downgrade are performed in relation to atomicity and ordering and shows that they 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 but not thread synchronization.

Keywords

Cache coherence; memory consistency; self-invalidation; critical sections; atomicity

1. INTRODUCTION

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 [5, 2, 3, 6]. With the addition of self-downgrade, the directory can be eliminated [6] and virtual cache coherence becomes feasible at low cost, without reverse translation [4].

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. Most of these proposals offer sequential consistency for data-race-free (SC for DRF) programs [1]. 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. Self-invalidation is therefore initiated on synchronization. Unfortunately, exposing all synchronization to the hardware, causes indiscriminate self-invalidation on lock, barrier, and wait constructs.

PACT '16 September 11-15, 2016, Haifa, Israel

ACM ISBN 978-1-4503-4121-9/16/09.

DOI: http://dx.doi.org/10.1145/2967938.2974050

We show that self-invalidation does not need to be applied conservatively. Our key observation is that `lock` / `unlock` synchronization (i.e., ensuring atomic execution of a critical section) is often *not* intended to constitute DRF synchronization for its *surrounding code*. 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 prior to its execution. Using similar reasoning, 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.

2. ORDERING VERSUS ATOMICITY

Ordering. Some synchronization primitives, such as 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 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 [1].

Atomicity. On the other hand, some other synchronization primitives do not inherently establish order. A common example is the critical section 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, 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 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. A typical

Thread 1	Thread 2

`lock(l);`
`  count += my_count;`
`unlock(l);`
	`lock(l);`
	`  count += my_count;`
	`unlock(l);`

Figure 1: Critical sections used for atomicity.

Thread 1	Thread 2

`x = 1;`
`lock(l);`
`  flag++;`
`unlock(l);`
	`lock(l);`
	`  local = flag;`
	`unlock(l);`
	`if (local)`
	`  print(x);`

Figure 2: Critical sections used for ordering.

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.

3. PROPOSAL

We propose new synchronization primitives for locks: Forward Self-Invalidation (FSI) and Forward Self-Downgrade (FSD). In a critical section, the accesses between `lock` and `unlock` are always self-invalidated/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 (Figure 1) or thread-ordering (Figure 2). Hence, our new primitives allow for applications to be optimized by preventing self-invalidation/downgrade of variables surrounding critical sections for the case when the locks only provide atomicity. When the programmer needs to take care of accesses surrounding thread-ordering critical sections, conservative self-invalidation/downgrade is employed.

Forward Self-Invalidation (FSI). 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.

Forward Self-Downgrade (FSD). When exiting a critical section, only the data written inside it need to be made globally visible. An efficient implementation of FSD is based on write-throughs via a coalescing write buffer that delays write throughs for a small period of time [6]. Our implementation uses 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.

4. RESULTS

Our proposal is evaluated for an extensive set of benchmarks from Splash3 and PARSEC, where locks have been modified appropriately to take advantage of FSI and FSD where possible. Our evaluation using the GEMS simulator for 64-core multiprocessors show that our techniques significantly limit penalties occurring in synchronization-intensive benchmarks. Results report 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 [6] that employs Callbacks [7] for efficient spin-waiting (7.6% in execution time and 9.1% in energy consumption).

5. CONCLUSIONS

We found that self-invalidation and self-downgrade, that conventionally affect what happened in the past, can be substantially improved if we turn them forward in time. This is possible for critical sections that do not ascertain thread-order without changing program semantics. This way, important performance and energy improvements are obtained over traditional and state-of-the-art self-invalidation cache coherence protocols.

6. ACKNOWLEDGMENTS

This work has been jointly supported by the "Fundación Séneca – Agencia de Ciencia y Tecnología de la Región de Murcia" under the 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).

7. REFERENCES

[1] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 2–14, June 1990.

[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)*, 60(4):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)*, pages 155–166, Oct. 2011.

[4] S. Kaxiras and A. Ros. A new perspective for efficient virtual-cache coherence. In *40th Int'l Symp. on Computer Architecture (ISCA)*, pages 535–547, June 2013.

[5] 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)*, pages 48–59, June 1995.

[6] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–252, Sept. 2012.

[7] 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)*, pages 427–438, June 2015.