On the tamability of the Location Consistency memory model

Charles Wallace
Computer Science Dept.
Michigan Technological University
Houghton, MI, USA

Guy Tremblay
Dépt. d'informatique
Université du Québec à Montréal
Montréal, QC, Canada

José N. Amaral
Computing Science Dept.
University of Alberta
Edmonton, AB, Canada

Abstract

Gao and Sarkar have proposed Location Consistency *(LC), the weakest memory model described in the literature to date for shared memory architectures. The advantage of LC is that it does not require invalidation/update messages across processors, thus dispensing with cache snooping or directories. Gao and Sarkar have argued that, although LC is strictly weaker than* Release Consistency *(RC), it is equivalent to RC for all programs that are* data-race free. *However the LC specification is silent about which reorderings of consistency related operations and program statements are allowed. In this paper we argue that under permissive reordering rules, LC is not equivalent to RC, even for data race free programs. On the other hand, if strict reordering rules are followed, then LC is equivalent to RC for data race free programs but the distinction between LC and RC is greatly diminished and many of the advantages of LC are lost.*

Key words: *Multiprocessors, Shared Memory, Weak Memory Models*

1 Introduction

On a shared memory multiprocessor machine, a *memory consistency model* is a contract between programs and the underlying machine architecture. This contract constrains the order in which memory operations appear to be performed with respect to one another, *i.e.*, the order in which the operations become visible to processors [6]. By constraining the order of operations, a memory consistency model determines which values can be returned by each read operation. The implementation of a memory consistency model in a machine with caches requires a *cache protocol* that invalidates or updates cached values when these values can no longer be returned to read operations.

The most common memory consistency model, *sequential consistency* (SC) [12], ensures that memory operations performed by the various processors are *serialized* (*i.e.*, seen in the same order by all processors). This serialization results in a model that is similar to the familiar uniprocessor model. Under SC there is always a unique most recent write to a location and all other values stored in the system for that location must be either invalidated or updated. Thus, a major drawback of SC on a multiprocessor machine is the high level of interprocessor communication required by the cache protocol.

Because of the requirement that all write memory operations be serialized into a single total order, the SC model is quite restrictive and is thus said to be a *strong* memory model. *Weaker* memory models have been proposed to relax the requirements imposed by SC. Examples include *release consistency* [9], *lazy release consistency* [11], *entry consistency* [4], *DAG consistency* [5], and *commit, reconcile and fences* (CRF) [17]. Relaxed memory models place fewer constraints on the memory system than SC, which permits more parallelism and requires less interprocessor communication but complicates reasoning about program behavior.

All these models, SC as well as relaxed models, have the *coherence* property. In a coherent memory model, all writes become visible to other processors, and all the writes (by any processes) *to any given location* are seen in the same order by all processors.

In 1994, Gao and Sarkar proposed the *Location Consistency (LC) memory model* [7], one of the weakest memory models proposed to date. LC is unusual in that it does not ensure coherence.[1] Under LC, memory operations performed by multiple processors need not be seen in the same order by all processors. Furthermore, because LC allows the coexistence of multiple legal values for the same location, there is no need to invalidate or update *remote* cached values; instead, LC only requires (local) *self-invalidation*. Hence the LC model has the potential to reduce consistency-related interprocessor traffic.

In a previous paper [18], we used the *Abstract State Machine* methodology [3] to give formal operational semantics for both the LC memory model and the associated cache protocol proposed by Gao and Sarkar [8]. Using these formal models, we were able to prove that the cache protocol does satisfy the memory model and, also, that the cache protocol is strictly stronger than the abstract memory model — in other words, the protocol does not allow certain behavior allowed by the model.

Gao and Sarkar claim that LC has the following "Equivalence Property": "the LC model is equivalent to the RC [Release Consistency] model for all program executions that are free of data races (access anomalies)" [7]. Since previous work has shown that RC is equivalent to Sequential Consistency (SC) for such programs [9], Gao and Sarkar conclude that programs without data races, when executed under the LC model, will always produce SC behavior. Since Gao and Sarkar's model is relatively informal, so is their proof of this property. As we will see, whether this property of LC holds depends strongly on the ordering constraints associated with synchronization operations. In this present paper, we argue that if the full flexibility of the LC model is allowed, then the Equivalence Property *does not hold*.

2 Synchronization and Operation Ordering in SC and RC

The LC memory model can be described using a simple programming model with four types of operation on memory locations: `read`, `write`, `acquire`, and `release`. Since in LC the content of a memory location is seen as a partially ordered set of values [8, 18], the semantics of these operations can be described as follows:

- A `read` retrieves one of the (legal) values associated with a location.
- A `write` adds a value to the set of values associated with the location. In any real system, the number of places available to store the values is finite. Therefore, a possible side effect is that a value previously associated with a given location may no longer be available.
- An `acquire` grants *exclusive* ownership of a location to a processor.[2] The exclusive ownership of a location thus imposes a sequential order on processor operations associated with that location. When acquiring a location, in order to gain access to the most recent value of that location as imposed by the sequential order, a processor also updates its own state by discarding any old value it has stored for the location (*self-invalidation*).
- A `release` operation takes exclusive ownership away from a processor. Any processor attempting to acquire a location currently owned by another processor must wait until the location is released by its current owner. If the releasing processor has written to the location, the release operation has the additional effect of making the value of its most recent write available to other processors. In this way, a processor that subsequently acquires the location will have access to the value of the global "most recent write".

Note that the `acquire` and `release` operations are assumed to come in pairs, *i.e.*, a processor must gain ownership of a location through an `acquire` before releasing that location. It is also *exclusively* through `release`/`acquire` pairs that the effect of a remote write is guaranteed to be made visible to a processor, as the acquiring processor is sure to see the value written by the releasing processor.

[1]However, it is not unique in this regard: *PRAM (Pipelined RAM) consistency* [13] also does not require coherence. The closely related *Processor Consistency* model [10] adds the coherence condition.

[2]In LC, contrary to SC, a processor without ownership of a location can perform `read` or `write` operations on that location, although their effect may not be globally visible.

A key motivation behind weak memory models, such as LC, is to relax the strict sequential order specified by a program and allow reordering of program instructions in order to improve parallelism. Such reordering can be done at compile-time (e.g., compiler optimizations and code reordering) or at run-time (e.g., dynamic instruction dispatching and execution).

Gao and Sarkar's description of the Location Consistency memory model does not explicitly specify which reordering operations are allowed with respect to `acquire` and `release` operations. To illustrate that this is an important issue, let us consider two possible assumptions for what kinds of reorderings around `acquire` and `release` operations could be allowed.

First, let us examine the program excerpt in Fig. 1. Under the SC model of execution, after all the statements in the figure are executed, it is not possible for the locations `a` and `b` to have the values `a=0` and `b=0`. If `a=0`, then the read of `y` by processor 1 must precede the write of `y` by processor 2. SC requires that the sequential "program order" of instructions by a single processor be maintained. Here, program order dictates that processor 1's write of `x` precedes its read of `y`, and processor 2's write of `y` precedes its read of `x`. It follows that processor 1's write to `x` must have preceded processor 2's read of `x`, and therefore `b` must be assigned the value `1`.

On the other hand, under the LC model, `a=0` and `b=0` is indeed possible: for a given processor, the two instructions can be reordered, since they pertain to *distinct* locations. In fact, a similar result would be possible under RC. In RC, operations may be reordered if they are not synchronization operations. Given that such reordering are possible, the result `a=0` and `b=0` is then also possible.

A result that is not SC-compliant is possible for RC because the program contains a *data race* between the instructions `x = 1` in Processor 1 and `b = x` in Processor 2 (and similarty for `y` and `a`). Informally, a data race exists between two operations o_1 and o_2 if the following properties hold for o_1 and o_2 in a sequentially consistent execution where program order is obeyed [1]:

- Operations o_1 and o_2 access the same memory location and one of them is a `write`.
- Operations o_1 and o_2 are not ordered by the program order relation nor by any intervening synchronization operations.
- One of o_1 or o_2 is a data operation, i.e., is not a synchronization operation.

Under RC, data races can be removed from a program by inserting appropriate synchronization operations, as shown in Fig. 2. Since data operations from distinct processors are now separated by synchronization operations, the program becomes free of data races — this program is also said to be "properly synchronized". The result `x=0` and `y=0` now becomes impossible, since `acquire` and `release` operations in the RC model are always sequentially ordered and, most importantly, since `release` and `acquire` operations impose additional ordering constraints on the other operations [6]. More precisely, in RC, a `release` operation ensures that *all* previous operations have completed before the release completes, e.g., a `release` will complete only when all previous `write`s have been made visible to other processors. Similarly, an `acquire` prevents *all* subsequent operations from proceeding until after the `acquire` has been performed (including having obtained the associated lock).

In the LC model, as described in [8] and formalized in [18], operations applying to a given location are assumed to have no ordering constraints relative to operations on *other* locations. This raises certain difficulties regarding the notions of data race and properly synchronized program, which we examine in the next section.

3 Strict vs. Relaxed Interpretation of LC Synchronization Operations

Under the data race definition informally presented above, the program excerpt in Fig. 3 is data-race free because all conflicting data operations are now separated by synchronization operations. However, whether the execution under the LC model produces a result equivalent to RC and thus, since the program is properly synchronized, equivalent to SC depends on the exact interpretation of the `acquire` and `release` operations.

```
Processor 1                                   Processor 2
                Initial state:  x = y = 0
x = 1;                                        y = 1;
a = y;                                        b = x;
```

Figure 1. A program which, under RC or LC, is not SC

```
Processor 1                                       Processor 2
                    Initial state:  x = y = 0
acquire(lock);                                    acquire(lock);
x = 1;                                            y = 1;
a = y;                                            b = x;
release(lock);                                    release(lock);
```

Figure 2. A properly synchronized RC program which is SC

There are two possible interpretations for the semantics of `acquire` and `release` operations:

1. A *relaxed* interpretation: In the spirit of LC, where operations related to distinct locations are considered totally independent, only operations related with the acquired/released locations are constrained by the *blocking* properties of `acquire`/`release`.

2. A *strict* interpretation: In the style of RC, operations related with any locations are constrained by an `acquire` or `release` operation, regardless of the location on which the `acquire`/`release` is acting.

In our formal specification of LC [18], faithful to the original paper [8] and as confirmed in private discussion with one of the authors, the relaxed interpretation was assumed. However, based on this relaxed interpretation, although the program in Fig. 3 can be considered "properly synchronized" and free of data races, it can still produce a result not allowed by SC: `a=0` and `b=0` is indeed possible since the the sequence of three operations on `y` (resp. `x`) on Processor 1 (resp. 2) can be moved before the operations on `x` (resp. `y`).

Under this relaxed interpretation, however, properly synchronized programs do obey the weaker condition of coherence. A memory system is coherent if "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [9]. In other words, all writes to a location are seen in the same order by all processors. Under LC, if each memory access to a location l is enclosed between an appropriate pair of `acquire`/`release` operations, the program will then be properly synchronized in relation to l and a total order will be imposed on the accesses to l, leading to coherence.

On the other hand, if the strict interpretation of synchronization operations is taken, then the behavior of the resulting program would be equivalent to a properly synchronized RC program and thus to SC. However, this strict interpretation would require some significant changes to the LC model, since operations pertaining to distinct locations could now be related with one another. This more strict interpretation would thus preclude some of the reordering opportunities offered by LC's initial weak approach to reordering. Furthermore, synchronization operations become costly under this interpretation: an `acquire` or `release` on one location may require remote cache invalidations or updates for other locations.

The fact that the behavior of `acquire` and `release` in the RC model differs from the one in LC based on the relaxed interpretation can also be confirmed by the work of Shen et al. [17], who present definitions of RC-style `acquire` and `release` in terms of their CRF model. For example, in their model, an `Acquire(s)` from RC can be defined as follows:

```
Acquire(s) = Lock(s);
PostFenceR(s);
Reconcile(*)
```

```
Processor 1                                         Processor 2
                   Initial state:  x = y = 0
acquire(x);                                         acquire(y);
x = 1;                                              y = 1;
release(x);                                         release(y);
acquire(y);                                         acquire(x);
a = y;                                              b = x;
release(y);                                         release(x);
```

Figure 3. A properly synchronized LC program which is *not* SC

The key element in this definition is the `PostFenceR(s)` operation, which constrains execution so that all reads of location `s` that "[precedes] the fence [must] be completed before any memory access [...] following the fence can be performed" [17].

4 Beyond simple `acquire/release`

When considering what kinds of dependencies (or ordering constraints) the `acquire` and `release` operations impose in a program, the dual-purpose role of these operations — lock/unlock and fence — becomes evident. In their LC paper, Gao and Sarkar introduced an "optimization" that they claim to be applicable only to the LC cache protocol. For that optimization, they use a `lock/unlock` pair to protect critical sections, and introduce a `refresh`, a `writeback`, and a `sync_writeback` operations to implement the consistency related operations. A `refresh(x)` operation performs a cache invalidation of `x`, as does the `acquire` operation. A `writeback(x)` operation triggers a writeback to main memory, with a synchronization signal returned upon its completion. A `sync_writeback` completes only after all such synchronization signals, from the same critical region, have been received.

It seems that the separation of the purposes of `acquire/release` is not only desirable as an optimization for the cache protocol, but is also required in the memory model itself in order to provide clear semantics and clearer rules for code motion. This separation of functions leads to the following constraints on reordering:

- No operation should be allowed to move across a `lock` or `unlock` operation because allowing such moves would be equivalent to moving code into or out of critical sections, a clear violation of programmer's intent.
- The only code motion restrictions on `refresh`, `writeback`, and `sync_writeback` should be related to operations that reference the same location.

A remaining interesting question is what should be the granularity of a `lock` operation. Under the relaxed interpretation of LC, per-location lock operations are insufficient if stricter SC-like consistency is desired. On the other hand, a single global lock, requiring invalidations and writebacks of all locations, seems contrary to the spirit of LC.[3] A possible compromise would involve defining lock operations on *sets* of locations. For any set `S` of locations, an `acquire(S)` operation would not complete until exclusive ownership of all locations in `S` were obtained. In addition, cache entries for each location would also be invalidated. Likewise, a `release(S)` operation would involve writebacks to all locations in `S` and releases of the associated locks. In this way, the programmer could select particular locations requiring synchronization, thereby avoiding synchronization of all locations. In the worst case, if the locations requiring synchronization cannot be determined, a global `acquire(*)` and `release(*)` could be performed. Two questions arise with regard to this approach: (1) how easy it

[3] It should be noted, however, that the memory model for Java proposed by Manson and Pugh [15] uses exactly this interpretation of LC for its semantics of non-volatile variables. While this does constrain LC greatly, the lack of a coherence condition on unsynchronized operations does present a potential performance benefit.

would be for programmers to choose the correct set of locations for each synchronization; (2) how feasible it would be to implement these atomic multi-location synchronization operations in hardware.

5 Conclusion

Location Consistency (LC) is an interesting weak memory model for multiprocessor machine, since the implementation of its cache protocol can be done efficiently (no need for invalidation/update messages across processors). However, there are still a number of issues to be addressed before it can be used as the basis for high-level languages memory model, for instance: What are the legal reordering associated with LC's synchronization operations? Are `acquire`/`release` sufficient as synchronization operations and what should their granularity be?

Our goal for future research is to continue investigating the properties of LC and, among other things, the advantages and drawbacks of working *without* coherence. Clearly, getting a new memory model right is no easy task — see all the difficulties encountered in defining Java's memory model [16, 14, 15]. Investigating how LC, or variants of it, could be used for defining Java's memory model is thus also one of our key reseach direction.

References

[1] Adve, S., Hill, M.D., Miller, B.P., Netzer, R.H.B.: "Detecting data races on weak memory systems"; Proc. ISCA (1991), 234–243.

[2] Adve, S.V., Gharachorloo, K.: "Shared memory consistency models: a tutorial"; Research Report 95/7, Digital Western Research Laboratory (1995).

[3] ASM home page; `http://www.eecs.umich.edu/gasm/`.

[4] Bershad, B., Zekauskas, M., Sawdon, W.: "The Midway distributed shared memory system"; Proc. IEEE COMPCON (1993), 528–537.

[5] Blumofe, R.D., Frigo, M., Joerg, C.F., Leiserson, C.E., Randall, K.H.: "An analysis of DAG-consistent distributed shared-memory algorithms"; Proc. ACM SPAA (1996), 297–308.

[6] Culler, D.E., Singh, J.P., Gupta, A.: "Parallel computer architecture: a hardware/software approach"; Morgan Kaufmann (1999).

[7] Gao, G.R. and Sarkar, V.: "Location consistency: Stepping beyond the barriers of memory coherence and serializability."; ACAPS Technical Memo 78, School of Computer Science, McGill University (1994).

[8] Gao, G.R., V. Sarkar: "Location consistency — A new memory model and cache consistency protocol"; IEEE Trans. on Computers 49, 8 (2000), 798–813.

[9] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: "Memory consistency and event ordering in scalable shared-memory multiprocessors"; Proc. ISCA (1990), 15–26. Also in Computer Architecture News 18, 2 (1990).

[10] Goodman, J.R.: "Cache consistency and sequential consistency", Technical report 1006, Computer Science Dept., University of Wisconsin–Madison (1989).

[11] Keleher, P., Cox, A.L., Zwaenepoel, W.: "Lazy release consistency for software distributed shared memory". Proc. ISCA (1992), 13–21. Also in Computer Architecture News 20, 2 (1992).

[12] Lamport, L.: "How to make a multiprocessor computer that correctly executes multiprocess programs"; IEEE Trans. on Computers C-28, 9 (1979), 690–691.

[13] Lipton, R.J., Sandberg, J.S.: "PRAM: A scalable shared memory"; Technical Report CS-TR-180-88, Princeton University (1988).

[14] Maessen, J.-W., Arvind, Shen, X.: "Improving the Java memory model using CRF"; Proc. OOPSLA (2000), 1–12.

[15] Manson, J., Pugh, W.: "Multithreaded semantics for Java"; CS Technical Report 4215, University of Maryland (2001).

[16] Pugh, W.: “Fixing the Java memory model”. In ACM 1999 Java Grande Conference (1999), 89-98.

[17] Shen, X., Arvind, Rudolph, L.: “Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers”; Proc. ISCA (1999), 150–161.

[18] Wallace, C., Tremblay, G., Amaral, J.N.: “An Abstract State Machine specification and verification of the Location Consistency memory model and cache protocol”; Journal of Universal Computer Science 7, 11 (2001), 1088–1112.