The Evolution of the C++ Memory Model

Jim Eckerlein
Technical University of Munich
jim.eckerlein@tum.de

Abstract

We present the past and current state of the *C++ Memory Model*. Compared to other language features, the memory model is especially hard to comprehend. As we want to address a wide audience, no prior knowledge about memory models is generally assumed. Instead, this work starts by motivating the need for a memory model. We then advance to describe how this concept is realized in C++ and how it evolves throughout the standard's revisions.

Keywords C++, Memory Model, Atomic Operations, Memory Order

1 Introduction

A memory model has two tasks: Firstly, it defines what units of memory the model is arguing about. C++ calls this units *memory locations*. Since their exact definition is less interesting, we defer it to the end of this work. Intuitively, each variable is stored in its unique memory location, which in turn is made up of one or more bytes. Secondly - which makes up the interesting part of a memory model, is how a program interacts with these memory units at runtime. The latter's definition especially becomes non-trivial in the existence of multiple threads.

2 Motivation for Memory Models

Compilers are free to transform the source code to improve speed or reduce memory footprint, as long as they maintain the illusion that the original program ran. This property of code transformations is provable in a single-threaded program because compilers know when a program can access specific memory locations. However, this curtain of illusion is cast aside when using a debugger to step through a compiled program. Code segments can be observed to run in a different order than authored, and variables are missing or are being introduced. We want to provide two examples of such source code transformations:

Originally

```
a = 2;
b = "Hello world";
a = 42;
```

Optimized

```
a = 42;
b = "Hello world";
```

The first example shows two writes to a variable. Since no read occurs between the writes, the leading write operation can be omitted entirely. The running program is not able to detect the lacking write.

Originally

```
for (int i = 0; i < N; ++i) {
  x += f(i);
}
```

Optimized

```
register int r1 = x;
for (int i = 0; i < N; ++i) {
  r1 += f(i);
}
x = r1;
```

The second example consists of a loop repeatedly accessing the same accumulator variable. In order to reduce data traffic to the cache, the compiler will allocate a register serving as an accumulator instead. After the loop finishes, the register content is flushed to the actual variable.

Similar reasoning applies to the hardware, as it does not run instructions in the exact order as emitted by the compiler. Various optimizations such as branch prediction or store buffering will result in instruction reordering effects. To a single-threaded program, these effects will always be invisible. One relatively intuitive hardware optimization is the store buffer. Because store operations require more work than load operations, the memory unit queues them up in a store buffer. Example: A processor executes a program P first writing to a location w and than reading from another location r. The write operation is buffered, and then r is read from memory. Since the store buffer is processed asynchronously, the value to w is written to the actual memory location, possibly long after r is read. Thus the processor did not execute the program P but another program in which both operations are swapped. In general, any deviation in the execution from the originally authored program either by the compiler, processor,

cache, or any other component can be reasoned about as reorderings at source code level.

As mentioned above, these transformations are unnoticeable to the programmer in a single-threaded program. It is only when multiple threads access common data that this illusion can no longer be maintained. Essentially, a thread accessing another thread's data without any precautions will experience something similar to debugging a program, such as witnessing code segments running out of order or seeing unfinished writes. For example, consider the following two threads executing the respectively given operations:

Thread 1	Thread 2
x := 42	if (a)
a := 1	r := x

Thread 1 is supposed to generate some data into x and to signal completion by setting y to one. Thread 2 periodically tests y and accesses x upon success. However, due to reordering by the compilers or out-of-order execution of machine instructions, an execution might be:

Thread 1	Thread 2
x := 42	r := x
a := 1	if (a) { }

The guarding mechanism of a is now bypassed, and if thread 2 executes between thread 1's operations, it reads data it was not supposed to. This scenario could be realized by both threads running truly concurrent on a multi-core device or by thread 1 being preemptively interrupted. The programmer's responsibility is to enforce the intended execution by issuing ordering constraints.

3 C++98

C++ inherited a single-threaded standard from C. Like in C [9], there is no notion of multithreading in the ISO standard ratified in 1998. For a program to use multiple, communicating threads, hardware- and platform-dependent APIs like POSIX had to be used. Compilers were unaware of those operations, so programmers inserted assembly directives to ensure correct memory ordering. With time it became apparent that threads and their communication cannot be implemented as a library [4], thus making it the standard's responsibility to define a proper memory model.

4 C++11

The C++ ISO standard ratified in 2011, known as C++11, introduced a memory model fit for multi-threaded programs. It did so building on the knowledge gained in defining a memory model for Java 5.0 [12]. To realize this, an atomic library was introduced that includes the type template `std::atomic<>` with operations that could specify ordering constraints:

- `memory_order_relaxed`
- `memory_order_consume`
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel`
- `memory_order_seq_cst`

In program diagrams we introduce a special syntax. The $x :=_{mo}$ syntax describes an store operation to x where memory ordering is constraint by *mo*, which can be either *rxd* (relaxed), *rel* (release), or *sc* (sequentially consistent). The x_{mo} syntax describes a read from a variable x where memory ordering is constraint by *mo*, which can be either *rxd* (relaxed), *acq* (acquire), *con* (cosume), or *sc* (sequentially consistent). Some operations combine reading and writing. They can be agumented by the constraints *rxd* (relaxed), *acqrel* (acquire-release), or *sc* (sequentially consistent).

4.1 Atomics

We partition accesses into two categories: Data access and atomic access. Data accesses are treated as regular pre-C++11 accesses. When multiple threads read from and write to the exact memory location, partially written values can be read. On the other hand, atomic accesses ensure atomicity; they are complete in a single step relative to other threads. In reality, data accesses can be non-atomic for many reasons. For example, a processor could implement a write operation as multiple instructions. It could depend on whether the memory location is aligned, where the definition for correct alignment depends on the architecture. Due to a hierarchical cache, writes can also take some time to propagate into the topmost cache level. A subset of the API for atomics is shown in the following listing:

```
template<class T> struct atomic {
  T load(std::memory_order mo);
  void store(T value, std::memory_order
    mo);
  bool compare_exchange_strong(T&
    expected, T desired,
    std::memory_order mo);
  bool compare_exchange_weak(T& expected,
    T desired, std::memory_order mo);
};
```

Listing 1. Atomics API

To avoid code cluttering with calls to `load` and `store`, C++ overloads the casting operator and the assignment operator. The memory order argument in all functions defaults to sequential consistency. The type argument to `std::atomic<>` must be trivially copyable, copy constructible, and copy assignable. The atomic variable as a whole then still is neither copyable nor moveable. While this still allows whole structures to be atomic, the C++ standard does not guarantee that such constructs are lock-free. To realize these template instantiations, the standard library implementation is allowed to use mutexes. Since atomics are used to avoid locking mutexes, in practice, there are small atomic variables, usually integers, acting as guards for much more complex data, which acts as a payload from this point of view.

4.2 Memory Ordering Constraints

The following subsections present all available reordering constraints. A less strict constraint incurs less runtime overhead and allows for significant optimization opportunities by the compiler and processor. A more stringent constraint states more guarantees, enabling one to argue about execution order and causality chains.

4.2.1 Acquire and Release Constraints

The unidirectional barriers called *acquire* and *release* restrict memory operations to be moved in only one direction across the barrier. *Release* designates a location where a thread finishes working on some piece of shared data and is ready to release its content to other threads. No operation must be moved *after* the barrier. *Acquire* on the other hand, designates a location where a thread assumes the ownership for some piece of shared data. No operation must be moved *before* the barrier. The following code illustrates how to use the unidirectional barriers by fixing the two-thread example presented during the motivation of a memory model. We repeat the scenario: Thread 1 generates data into x and signals completion by setting a to one. Thread 2 waits for data being ready. Again, the if-statement and read from x in Thread 2 are free to be reordered.

Thread 1	Thread 2
x := 42	if (a)
a := 1	r := x

Now we make a an atomic (implicit in the diagram) and augment the stores to a and loads from a with constraints:

Thread 1	Thread 2
x := 42	if (a_{acq})
a :=$_{rel}$ 1	r := x

This prohibits the data generation phase in thread 1 to leak beyond the release and the data-gathering phase in thread 2 to leak before the acquisition. As a result, the problematic reordering of the two statements in thread 2 is now no longer possible. Practical C++ code would look like this:

```
#include <atomic>

// Globally shared data:
std::atomic<int> guard;
std::vector<double> data;

// Thread 1:
data = generate();
guard.store(1, std::memory_order_release);

// Thread 2:
for (;;) {
  if
    (guard.load(std::memory_order_acquire)
    == 1) {
    process(data);
  }
}
```

Listing 2. Acquire-Release Use Case

A `std::atomic` type also provides functions that combine reading and writing into a single atomic operation, meaning that relative to other threads, such an operation must always be complete in a single step. Memory reordering around such operations can be controlled by either a *relaxed*, *acquire-release*, or *sequentially consistent* constraint. Examples of such functions are `fetch_add()` and `compare_exchange()`.

4.2.2 Consume and Release Constraints

The *comsume* constraint is a barrier that is weaker than the *acquire* barrier. Whereas an *acquire* constraint forbids any operation from being reordered before the barrier, a *consume* constraint lifts this restriction except for operations depending on the atomic value.

1	a := 42
2	**b := $\mathbf{x_{con}}$**
3	print(a)
4	c := a + b
5	print(c)

The bold operation represents the barrier imposed through the load operation augmented by a *consume* constraint. The compiler can move operation three above the barrier because *a* does not depend on *x*. This is unlike the *acquire* ordering, which would disallow this potential optimization. However, operation four cannot be moved before the barrier. *x* is said to carry a dependency through *b* to *c*. By transitivity, operation five can also not be moved before the barrier.

To give the programmer fine-grained control over dependency chains originating from consume operations, C++ defines the attribute `[[carries_dependency]]` and the function `std::kill_dependency()`. Performance gains can be observed on weakly ordered architectures (Refer to section 10) because compilers insert fewer memory fence instructions. That is because such processors do not reorder instructions working on registers before the register content is loaded, thus having a built-in notion of *data-dependencies*. As a production example, Linux employs consume semantics in read-copy-update (RCU) implementations, although not by using C++11.

4.2.3 Sequentially Consistent Constraint

Sequential consistency [11] is the strictest constraint. A program using exclusively sequentially consistent constraints will always run as if only one thread is running at a time. When viewed in isolation, each thread executes the program in authored order.

Sequentially consistency constraints insert bidirectional barriers. No other operation can propagate across the barrier either way. In addition, the standard requires that an atomic only accessed with the sequentially consistent constraint must experience the same sequence of values throughout the program's runtime in every thread. This example demonstrates the effect of this property:

Thread 1	$x :=_{rel} 1$
Thread 2	$y :=_{rel} 1$

Thread 3	while (y_{acq} = 0) { }
	if (x_{acq})
	  ++z_{acqrel}

Thread 4	while (x_{acq} = 0) { }
	if (y_{acq})
	  ++z_{acqrel}

Thread 3 waits for $y = 1$. When it does, it tests $x = 1$. Upon success, z increments. Thread 4 behaves antisymmetrically in that x and y exchange roles. *Acquire* and *release* constraints are employed. Thus, two threads can experience different modification orders of variables. The outcome of z not being incremented by either thread is thus possible. The scenario occurs if thread 3 first saw $x = 1$ followed by $y = 1$, while thread 4 experienced the opposite order of events. The order of changes to variables does not need to be consistent across threads. To enforce this property, sequentially consistent ordering constraints must be used in all cases. This program will always end with $z = 1$:

Thread 1	$x :=_{sc} 1$
Thread 2	$y :=_{sc} 1$

Thread 3	while (y_{sc} = 0) { }
	if (x_{sc})
	  ++z_{sc}

Thread 4	while (x_{sc} = 0) { }
	if (y_{sc})
	  ++z_{sc}

4.2.4 Relaxed Constraint

The relaxed constraint imposes no ordering requirements. Only the atomicity of each access is guaranteed. Both the compiler and the processor are free to reorder such operations. Atomics accessed using relaxed constraints are also called *weakly ordered atomics*. Use cases for this type of constraint involve scenarios where the program does not depend on the current value of the atomic but employs other usually delayed means of synchronization. In general, designing correct code using weakly ordered atomics is significantly more complex than writing lock-free code, which is already a tremendously difficult process.

We want to present automatic reference counting (ARC) as one use case for relaxed atomics. When retaining a reference and the counter is incremented, the resulting value is not used for further computation. It turns out that for incrementing the reference counter,

relaxed access suffices. Only when releasing the reference all prior writes must become visible. By using a release constraint while decrementing the counter, no relaxed operation may be reordered afterward.

```
1 void retain() {
2   count->fetch_add(1,
      std::memory_order_relaxed);
3 }
4
5 void release() {
6   if (count->fetch_sub(1,
      std::memory_order_acq_rel) == 1) {
7     // Deallocation
8   }
9 }
```

Listing 3. ARC as a use case for relaxed constraints.

4.3 Fences

Fences are synchronization primitives which are not associated with an atomic variable. Every use of an atomic load or store can be replaced by fences:

Atomic operation	Equivalent using fences
$r \coloneqq a_{acq}$	$r \coloneqq a_{rxd}$ $fence_{acq}()$
$r \coloneqq a_{con}$	$r \coloneqq a_{rxd}$ $fence_{acq}()$
$a \coloneqq_{rel} r$	$fence_{rel}()$ $a \coloneqq_{rxd} r$
$r \coloneqq a_{sc}$	$r \coloneqq a_{rxd}$ $fence_{sc}()$
$a \coloneqq_{sc} r$	$fence_{sc}()$ $a \coloneqq_{rxd} r$

Since fences are not associated with specific memory locations, they have to impose more constraints on reordering to be able to state at least the same set of guarantees. A fence is thus more strict than an atomic operation with the equivalent memory ordering. An acquire load disallows any subsequent load to be moved before the barrier, while an acquire fence disallows any subsequent load to be moved before any preceding load. A release store disallows any preceding store to be moved after the barrier, while a release fence disallows any preceding load to be moved after any subsequent store. Compared to atomic operations, fences can be a pessimisation.

4.4 Allowed reorderings

Diagram 1 presents how atomic operations can be reordered with respect to surrounding code based on the given constraints. The strings represent atomic operations. The arrows show which way surrounding code can cross the barrier. Crossed arrows negate the permission. The angular arrows show how atomic operations may be reordered with respect to themselves.

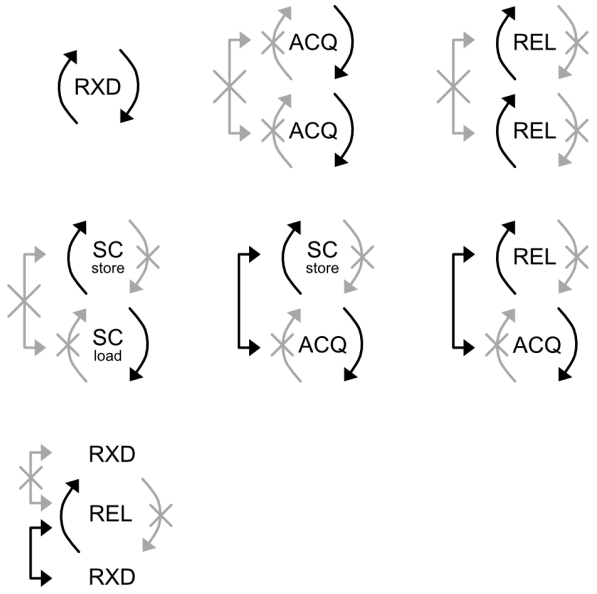

Figure 1. Allowed and prohibited reorderings

4.5 Compare-and-swap

The atomic library includes support for compare-and-swap (CAS), atomically testing a memory location for a given value. A new value is inserted upon success, and the expression evaluates truthfully. C++ offers two flavours, `compare_exchange_weak()` and `compare_exchange_strong()`. The weak version is allowed to fail spuriously, allowing for a more efficient implementation on some architectures. In general, the weak version will always be inside a loop. Conversely, the weak candidate might be preferable when a CAS operation is inside a loop anyway.

5 C++14

Unlike its preceding revision, C++14 mainly fixed existing issues rather than introducing new features.

5.1 Prohibiting Out-of-Thin-Air Reads

C++14 now explicitly prohibits a phenomena called *out-of-thin-air* reads, or *OOTA* for short [5, 6]. Under the effect of speculative execution, OOTA renders programs unusable that use relaxed accesses to form a causal cycle, such as this example:

Thread 1	Thread 2
$y :=_{rxd} x_{rxd}$	$x :=_{rxd} y_{rxd}$

The problem is when thread 1 speculatively loads 42 into y, followed by thread 2 quickly copying this value to x. Then, finally, the first thread's load arrives, confirming 42 as the correct value for y. Thus nothing has to be undone. But unfortunately, this execution injects a value into the execution the program never calculates.

A proposed solution was to restrict allowed reorderings of relaxed operations, but that would introduce performance penalties on weakly ordered architectures. Unfortunately, OOTA is a long-standing problem not only in C++ but also in Java. C++11 tried to outlaw OOTA by phrasing the specification accordingly. However, it turned out to be insufficient. C++14 mitigates this by replacing the complicated wording with a handwaving note that implementations should ensure OOTA shall happen for such circularly depending values. Mind that OOTA is fully understood and even less definable at that time.

6 C++17

Similar to the previous revision, C++17 did not introduce new features to the atomics library.

6.1 Discouriging Consume Semantics

Consume semantics have proved to be difficult to be implemented for both compilers and programmers despite efforts made to make the constraint more accessible [14]. C++'s notion of *data-dependency* is rather clumsy. Dependency chains are truncated by the compiler at function boundaries because their implementations might be unknowable or unmodifiable at compile-time [13]. To avoid the compiler lifting to acquire semantics, the programmer has to clutter the codebase with `[[carries_dependency]]` annotations and calls to `std::kill_dependency()`. The semantics have been revised multiple times [15], but since the debate is still open and in progress, *consume* has been declared temporarily discouraged [7].

7 C++20

As of writing this work, C++20 is the latest available revision. The standard atomic library now supports floating-point variables used as atomic data.

7.1 Repairing Sequential Consistency

Lahav et al. discovered problems with current compilation schemes on weakly ordered processors when combining release ordering with sequentially consistent ordering [10]. Such processors might execute the operations in an order that is not compliant with the definition of sequential consistency. Enforcing the standard would introduce overhead and eliminate the motivation for *acquire-release* to a significant extent. The settled approach is to rephrase the standard in a way that breaks backward compatibility in a few edge cases.

8 Memory Locations

The C++ standard generalizes a device on which a C++ program is executed to an abstract machine. Storage available to the abstract machine is discretized into cells called *bytes*, each one having its own unique address. Each byte, consisting of a contiguous bit sequence, must be able to represent each character used in the C++ grammar, as well as any code-unit present in the Unicode UTF-8 encoding. The number of bits is accessible to the programmer at compile-time through the macro `CHAR_BIT`. Almost all commercially available hardware uses 8 bits per byte, although some embedded systems, especially digital signal processors (DSP), use bytes wider than 8 bits.

Memory locations are disjoint and contiguous sets of bytes, each one forming a subset of the available storage. Memory locations are occupied by objects of scalar type: `int`, `char`, `float`, `double`, `bool`, enumerations, or pointers. In addition, various language features such as references or virtual functions might occupy additional memory locations which are not directly accessible to the programmer.

Integral members of a structure may be decorated with a bit-length, allowing for fine-grained alignment control. A consecutive sequence of such decorated members called bit fields forms a single memory location. Such a sequence may be subdivided using zero-length bit fields, which act as separators. The boundaries of subobjects such as members of a structure type or array elements also act as separators.

```
1 struct {
2   int a;
3   float b;
4   struct {
5     double c;
6   } s1;
7
8   int b1 : 2;
9   int b2 : 3;
10  int b3 : 0;
11  int b4 : 2;
12  struct {
13    int b5 : 4;
14  } s2;
15 };
```

Listing 4. Memory Locations

In this structure declaration `a`, `b` and `c` are mapped onto individual memory locations. Furthermore, the bit fields `b1` and `b2` are combined into a single memory location. `b3` acts as a separator. `b4` and `b5` occupy different memory locations, because one is member of a suboject.

For the sake of completeness, we want to mention the fact that there is also an *Object Model*. Whereas the memory model defines how objects are mapped to storage or, in other words space, the object model defines a mapping of objects to time. It thus specifies properties such as lifetime and storage duration. We consider the C++ object model out of scope for this work.

9 Atomic vs. Volatile

Volatility and atomicity are orthogonal concepts in C++, contrasting to the languages Java and C#, where the keyword `volatile` assumes the role of C++'s keyword `atomic`.

The primary motivation of `volatile` is access to memory whose values can change at any time without the program intention. Consequently, reads and writes from such memory locations are effectively unoptimizable. Furthermore, volatile accesses must not be moved across neighboring side effects, thus enforcing ordering constraints independent of ordering restrictions imposed by memory barriers. In addition, volatile variables do not enforce atomicity. Instead, a variable has to be declared as `volatile std::atomic<>`. Volatile variables are thus unsuited for inter-thread communication.

	Load		Store	
	Regular	Atomic	Regular	Atomic
x86	mov	mov	mov	xchg
IA64	ld	ld.acq	st	st.rel; **mf**
POWER	ld	**sync**; ld; cmp; bc; isync	st	**sync**; st
ARM v7	ldr	ldr; **dmb**	str	**dmb**; str; **dmb**
ARM v8	ldr	ldra	str	strl

Table 1. Instructions for reodering constraints.

10 Hardware implementation

The concept of memory models originates from the hardware level. Colossal progress has been made since the nineties [1–3, 8]. Processors fall somewhere in the spectrum between being strongly or weakly ordered. A strongly ordered processor states many guarantees about out-of-order execution and thus facilitates formal program verification. As an example for a strongly ordered architecture, x86 does not distinguish between regular and atomic load instructions. Loading an aligned 32-bit integer is always atomic; it does not have to be wrapped in a `std::atomic<>`. Unfortunately, this implies that non-atomic read operations pay for guarantees which are not required for correct execution. Weakly-ordered processors benefit from lower overhead, which is a reason for their supremacy in lower-power devices. Unfortunately, heavyweight fences have to be used to implement sequentially consistent operations. An exception is ARM v8, the first architecture to natively support the C++11 memory model by exposing load-acquire and store-release instructions.

Table 1 shows which instructions compilers use to implement atomic load and store operations, as well as their non-atomic counterparts. Instructions issuing memory fences are in bold.

11 Conclusion

Every programming language that exposes threads and allows them to communicate over shared mutable memory needs to define a memory model. C++11 introduced a memory model to enable programmers to write portable multithreaded code, supporting both locks in the form of mutexes and lock-free primitives in the form of atomics. Only atomic memory is suitable to coordinate inter-thread communication. Code does not necessarily run in authored order at runtime due to various effects resulting from compilation optimizations and out-of-order execution. This fact only becomes visible to the programmer when writing multithreaded programs. To ensure such programs still function valid, particular constraints have to be inserted, informing the compiler and the processor about prohibited reorderings. Since the memory order affinity of individual architectures varies wildly, each type of constraint is a compromise between efficiency and deterministic behavior. Subsequent C++ revisions repair defects and improve the definition, mirroring the steady progress in research and understanding of this complex topic.

References

[1] Sarita Adve. 1993. *Designing Memory Consistency Models for Shared-memory Multiprocessors*. University of Wisconsin–Madison.

[2] Sarita Adve and Kourosh Gharachorloo. 1996. Designing Memory Consistency Models for Shared-memory Multiprocessors: A tutorial. *IEEE Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

[3] S.V. Adve and M.D. Hill. 1989. *Weak Ordering: A New Definition and Some Implications*. Number no. 902 in Computer sciences technical report. University of Wisconsin-Madison, Computer Sciences Department. https://books.google.de/books?id=8KeYPgAACAAJ

[4] Hans-J. Boehm. 2005. Threads Cannot Be Implemented As a Library. https://web.stanford.edu/class/cs240/readings/p261-boehm.pdf. Accessed 2022-01-31.

[5] Hans-J. Boehm. 2013. N3710: Specifying the absence of "out of thin air" results. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html. Accessed 2020-12-2.

[6] Hans-J. Boehm. 2013. N3786: Prohibiting "out of thin air" results in C++14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm. Accessed 2020-12-2.

[7] Hans-J. Boehm. 2016. P0371R1: Temporarily discourage memory order consume. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r1.html. Accessed 2020-12-2.

[8] Michel Dubois, Christoph Scheurich, and Faye Briggs. 1998. Memory Access Buffering in Multiprocessors. *ACM Sigarch Computer Architecture News* 14, 320–328. https://doi.org/10.1145/17356.17406

[9] B.W. Kernighan, D.M. Ritchie, and C.L. Tondo. 1988. *The C Programming Language*. Prentice Hall. https://books.google.de/books?id=161QAAAAMAAJ

[10] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. [n. d.]. Repairing Sequential Consistency in C/C++11. http://plv.mpi-sws.org/scfix/paper.pdf. Accessed 2020-12-3.

[11] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28* 9 (September 1979), 690–691. https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/

[12] Jeremy Manson, William Pugh, and Sarita V. Adve. 2004. *The Java Memory Model*. University of Maryland. https://books.google.de/books?id=kOjzvQAACAAJ

[13] Paul E. McKenney and Lawrence Crowl. 2008. N2643: C++ Data-Dependency Ordering: Function Annotation. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2643.html. Accessed 2020-12-2.

[14] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, Olivier Giroux, and Lawrence Crowl. 2016. P0098R1: Towards Implementation and Use of memory_order_consume. http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2016/p0098r1.pdf. Accessed 2020-12-2.

[15] Paul E. McKenney, Michael Wong, Hans Boehm, Jens Maurer, Jeffrey Yasskin, and JF Bastien. 2016. P0190R2: Proposal for New memory order consume Definition. http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2016/p0190r2.pdf. Accessed 2020-12-2.