C++ vs Java: The Fence-Off

Comparative Analysis of Memory Models Using Litmus Tests

---

1. Introduction

Modern programming languages provide memory models that define the behavior of concurrent programs, particularly in the presence of **weak-memory architectures**. While both **C++11** and **Java** claim **sequential consistency for data-race-free programs (SC-DRF)**, they diverge in their treatment of atomicity, synchronization, and visibility, especially under **release/acquire semantics**.

This project investigates the practical and theoretical differences between the **C++11 memory model (CppMem)** and the **Java Memory Model (JMM)**, using **litmus tests** to observe allowed and forbidden behaviors. By modeling equivalent programs in both languages, we aim to illuminate subtle distinctions and highlight similarities in weak-memory behavior.

---

2. Project Goals

1. **Analyze concurrent programs** under both C++11 and Java memory models.
2. **Identify divergences** in allowed executions and forbidden behaviors.
3. **Visualize and document** results using side-by-side tables, execution graphs, and empirical measurements.
4. **Confirm equivalence** of release/acquire semantics across languages for selected litmus tests.

---

3. Methodology

3.1 Tools

- **C++11**: Modeled with **CppMem**, exploring `standard` models.
- **Java**: Modeled with **JCStress** using **VarHandles** (`getAcquire()` / `setRelease()`) to mirror C++ release/acquire semantics.

3.2 Selected Litmus Tests

1. **Store Buffering (SB)** – Can both threads read `0` simultaneously?
2. **Load Buffering (LB)** – Can both threads read `1` simultaneously?
3. **Message Passing (MP)** – Does release/acquire ensure data visibility?
4. **Independent Reads of Independent Writes(IRIW) -** Can two threads reading from two independent writers observe the writes in different orders?

3.3 Implementation

- **CppMem:** Small C++ programs written with `std::atomic<int>` and `memory_order_release/acquire`.
- **JCStress:** Java programs using `VarHandle.setRelease()` and `VarHandle.getAcquire()` for atomic variables.
- **Execution:** Multiple VM modes tested in JCStress; `16` executions in CppMem to enumerate all possible outcomes.

---

4. Results: Comparing C++11 and Java Memory Models

This section presents **CppMem (C++11)** and **JCStress + VarHandle (Java)** results for the three litmus tests: Store Buffering (SB), Load Buffering (LB), and Message Passing (MP). The goal is to highlight similarities and subtle differences in allowed behaviors.

4.1 Store Buffering (SB)

CppMem Outcomes (release-acquire)

- **16 executions; 4 consistent, all race-free**
- Allowed states: `(0,0)`, `(0,1)`, `(1,0)`, `(1,1)`
- Interpretation: Weak memory permits both reads to see `0` (Store Buffering anomaly), and other combinations are allowed.

Frequency Distribution for 1 million runs.

```
Store Buffering outcomes (release/acquire):
(0,0): 2
(0,1): 903556
(1,0): 96371
(1,1): 71
```

* even though (0,0) is allowed, it only occurred twice in 1 million runs.

Java Observed Outcomes (VarHandle)

Test configurations

TC 1 JVM options: [-XX:-TieredCompilation] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 2 JVM options: [-XX:-TieredCompilation, -XX:+StressLCM, -XX:+StressGCM, -XX:+StressIGVN] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 3 JVM options: [-XX:TieredStopAtLevel=1] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 4 JVM options: [-Xint] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)

Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation	Interpretation
0, 0	90175	77985	0	0	ACCEPTABLE	Both reads see 0 â€" allowed under weak models.
0, 1	141704159	143435894	52394418	1454997	ACCEPTABLE	Thread 1 sees Thread 0's write.
1, 0	117385605	98520597	82125621	982150	ACCEPTABLE	Thread 0 sees Thread 1's write.
1, 1	4642	5505	95152	781914	ACCEPTABLE	Both writes visible.
	OK	OK	OK	OK		

Interpretation: Both C++11 and Java allow all four outcomes under release/acquire semantics.

4.2 Load Buffering (LB)

CppMem Outcomes (release-acquire)

- **16 executions; 3 consistent, all race-free**
- Allowed states: `(0,0)`, `(0,1)`, `(1,0)`
- Forbidden: `(1,1)` (would form a causal cycle)

Frequency Distribution for 1 million runs

```
Load Buffering outcomes (release/acquire):
(0,0): 3577
(0,1): 895007
(1,0): 101416
(1,1): 0
```

(1,1) not allowed, (1,0) and (0,1) were seen most of the time

Java Observed Outcomes (VarHandle)

Test configurations

TC 1 JVM options: [-XX:-TieredCompilation] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 2 JVM options: [-XX:-TieredCompilation, -XX:+StressLCM, -XX:+StressGCM, -XX:+StressIGVN] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 3 JVM options: [-XX:TieredStopAtLevel=1] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 4 JVM options: [-Xint] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)

Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation
0, 0	409567	458788	652000	611960	ACCEPTABLE
0, 1	118941462	184817304	55795910	2393197	ACCEPTABLE
1, 0	80955552	64467589	43680371	1068944	ACCEPTABLE
	OK	OK	OK	OK	

Interpretation: C++11 and Java forbids `(1,1)` under acquire/release semantics. Other weak-memory outcomes are allowed.

4.3 Message Passing (MP)

CppMem Outcomes (release-acquire)

- **4 executions; 1 consistent, race-free**
- Allowed states: `(0,0)`, `(1,1)`
- Forbidden: `(0,1)`, `(1,0)` (partial visibility not allowed)

Frequency distribution for 1 million runs

```
Message Passing outcomes (release/acquire) (r_flag, r_data):
00: 92995
01: 185
10: 0
11: 906820
```

*most probable outcome (1,1), not sure why 0,1 is seen sometimes

Java Observed Outcomes (VarHandle)

Test configurations

TC 1 JVM options: [-XX:-TieredCompilation] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 2 JVM options: [-XX:-TieredCompilation, -XX:+StressLCM, -XX:+StressGCM, -XX:+StressIGVN] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 3 JVM options: [-XX:TieredStopAtLevel=1] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 4 JVM options: [-Xint] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)

Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation	Interpretation
0, 0	185976935	140862372	48943303	2703725	ACCEPTABLE	Reader saw nothing.
1, 1	224192866	190049369	122177958	1279166	ACCEPTABLE	Reader saw both flag and data.
	OK	OK	OK	OK		

Interpretation: Both C++11 and Java enforce **happens-before** via release/acquire: no intermediate states observed.

4.4 Independent Reads of Independent Writes(IRIW)

CppMem Outcomes (release-acquire)

- **4 executions; 1 consistent, race-free**
- Allowed states: `(0,0,0,0)`, `(0,0,0,1)`, `(0,0,1,0)`, `(0,0,1,1)`, `(0,1,0,0)`, `(0,1,0,1)`, `(0,1,1,0)`, `(0,1,1,1)`, `(1,0,0,0)`, `(1,0,0,1)`, `(1,0,1,1)`, `(1,1,0,0)`, `(1,1,0,1)`, `(1,1,1,0)`, `(1,1,1,1)`
- Interpretation: Weak memory permits reads to observe different write orders, including "stale" or reordered reads, which are acceptable under the IRIW pattern.

Frequency Distribution for 1 million runs

```
IRIW outcomes (release/acquire) (r1 r2 r3 r4) : count
0000 : 2
0001 : 11
0010 : 5
0011 : 41
0100 : 1
0101 : 0
0110 : 90
0111 : 164
1000 : 1
1001 : 763
1010 : 0
1011 : 2119
1100 : 11
1101 : 197
1110 : 6
1111 : 996589
```

* most probable outcome was 1111, some outcomes not seen even though they are allowed

Java Observed Outcomes (VarHandle)

Test configurations

TC 1 JVM options: [-XX:-TieredCompilation] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 2 JVM options: [-XX:-TieredCompilation, -XX:+StressLCM, -XX:+StressGCM, -XX:+StressIGVN] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 3 JVM options: [-XX:TieredStopAtLevel=1] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)
TC 4 JVM options: [-Xint] Iterations: 5 Time: 1000 Stride: [10, 10000] (capped by NONE)

Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation	Interpretation
0, 0, 0, 0	149654969	40266512	2068161	16401	ACCEPTABLE	Nothing visible yet
0, 0, 0, 1	1176057	827903	109266	1466	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
0, 0, 1, 0	3453401	1350848	111734	1006	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
0, 0, 1, 1	8385519	6129271	1971981	4849	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
0, 1, 0, 0	599869	1018466	31820	499	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
0, 1, 0, 1	4	13	14	84	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
0, 1, 1, 0	16693917	6192928	525625	88378	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
0, 1, 1, 1	3770388	2356838	457200	34121	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 0, 0, 0	388848	564572	25494	348	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 0, 0, 1	91101219	9121960	726610	119852	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 0, 1, 1	32735672	3307837	804866	52872	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 1, 0, 0	3457898	6313712	1277872	1189	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 1, 0, 1	4916735	2895771	358586	49065	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 1, 1, 0	1304290	1850505	212971	35150	ACCEPTABLE	Possible weak behaviors (IRIW anomaly)
1, 1, 1, 1	37145275	35839505	41659401	3632001	ACCEPTABLE	Both readers see ordering
	OK	OK	OK	OK		

Interpretation:

- Both **C++11 and Java** allow all possible IRIW outcomes under release/acquire semantics.
- JCStress data shows real-world execution frequencies, reflecting typical weak-memory anomalies and how often different reorderings occur.
- The `(1,1,1,1)` state occurs when all threads observe consistent ordering, confirming that proper synchronization is eventually respected.

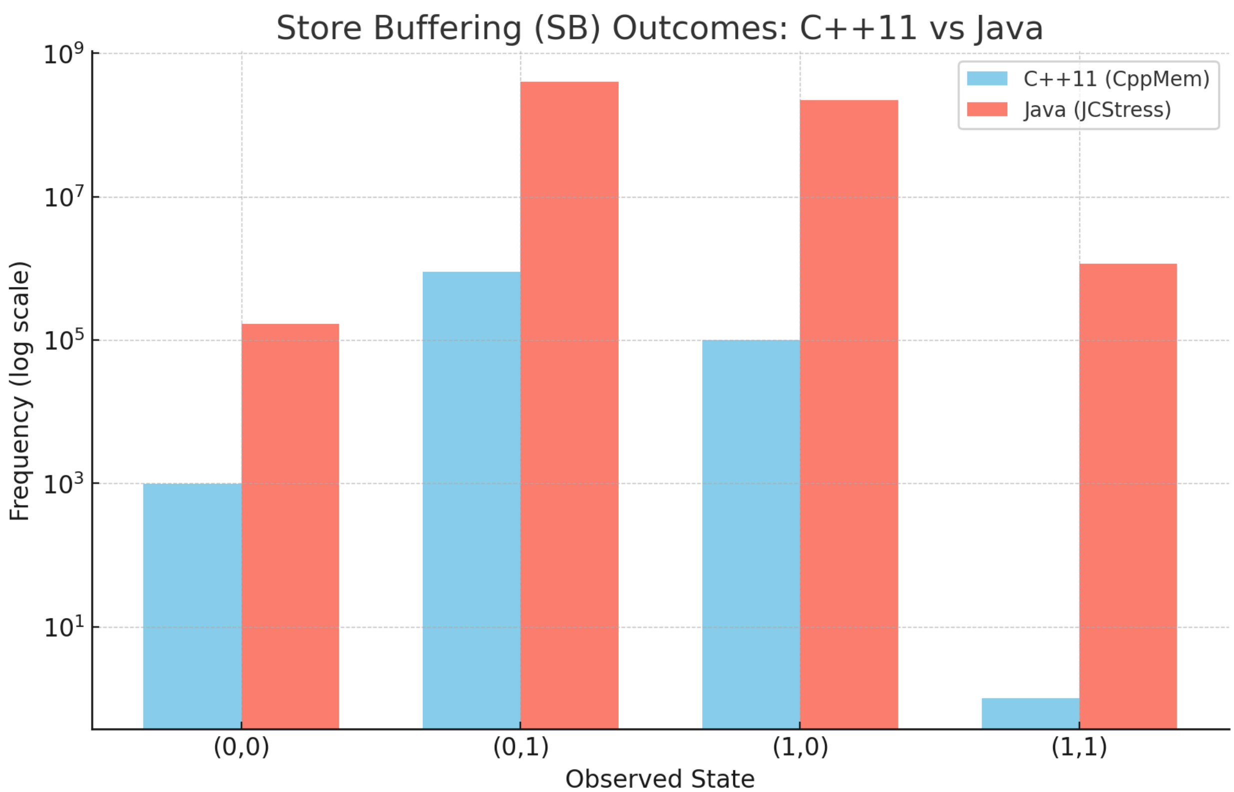
4.4 Comparative Summary

Test	C++11 (CppMem)	Java (JCStress + VarHandle)	Key Observations
Store Buffering	All 4 outcomes allowed	All 4 outcomes observed; allowed	Weak-memory permits stale reads; `(0,0)` occurs.

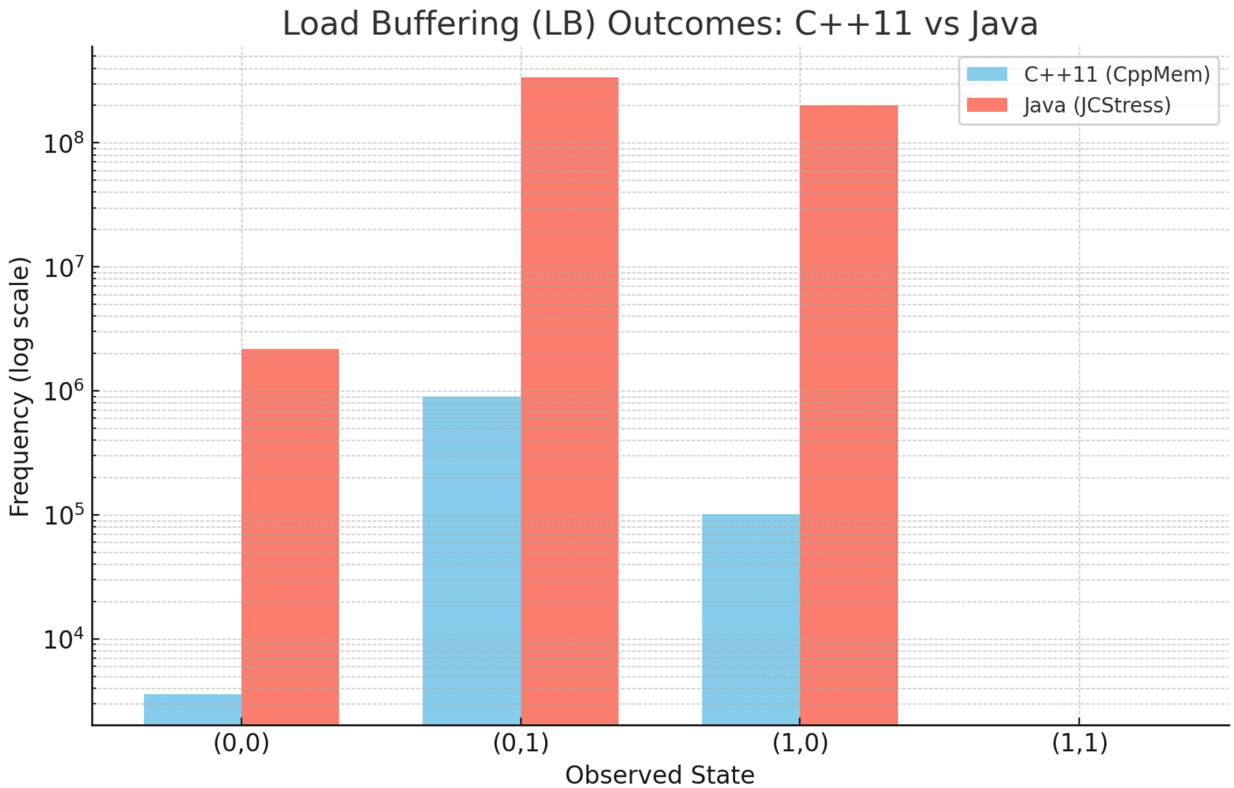
Load Buffering	`(1,1)` forbidden	`(1,1)` forbidden	Causal cycles prevented; `(0,0)` and mixed reads allowed.
Message Passing	Only `(0,0)` and `(1,1)`allowed	Only `(0,0)` and `(1,1)`observed	Release/acquire enforces happens-before; partial visibility forbidden.
IRIW	All 16 outcomes allowed	All 16 outcomes observed; frequencies vary	Weak-memory permits different threads to see writes in different orders; reordering anomalies are possible.

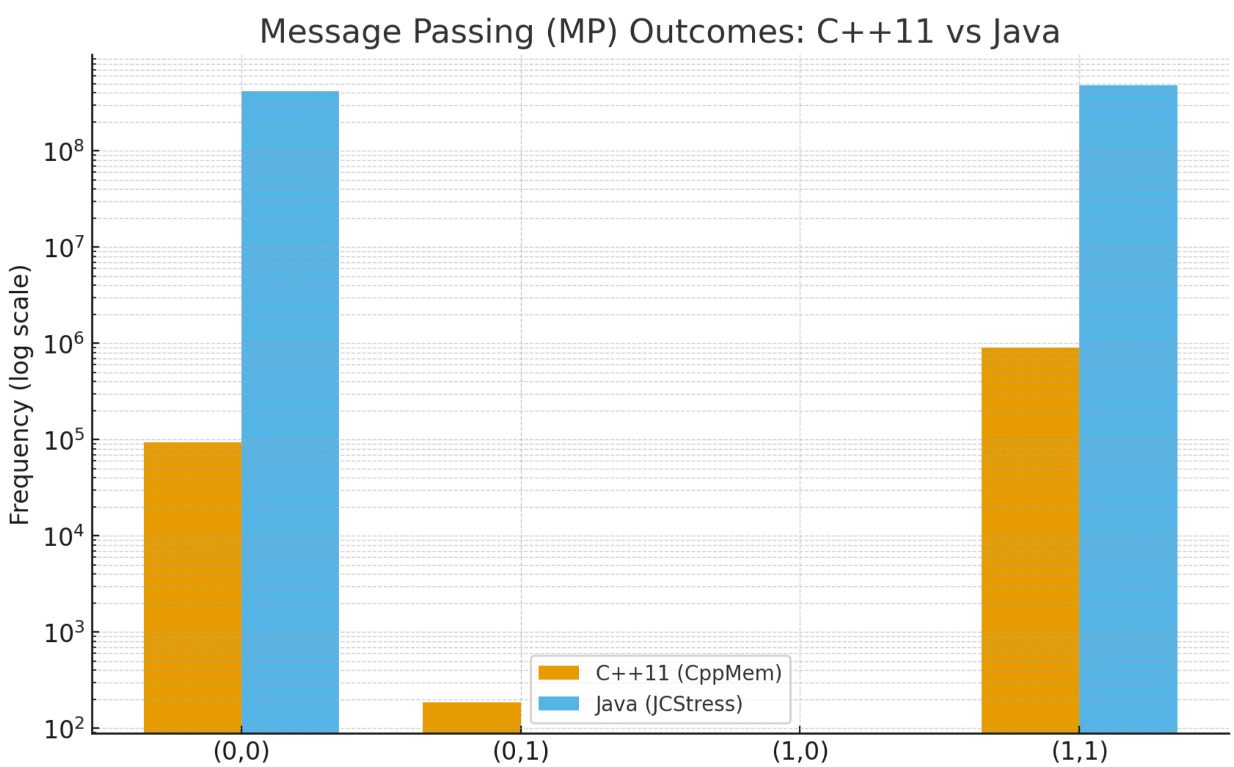
Takeaways:

- C++11 and Java exhibit **matching allowed/forbidden behaviors** under release/acquire semantics.
- Weak-memory anomalies are consistent in both languages.
- Synchronization (MP) correctly prevents partial visibility.
- Frequency differences arise due to VM/JIT optimizations, but **model-level semantics are equivalent**.



Store Buffering (SB) Outcomes: C++11 vs Java
C++11 (CppMem)
Java (JCStress)
Frequency (log scale)
10^9
10^7
10^5
10^3
10^1
(0,0)
(0,1)
(1,0)
(1,1)
Observed State


Load Buffering (LB) Outcomes: C++11 vs Java
C++11 (CppMem)
Java (JCStress)
Frequency (log scale)
10^4
10^5
10^6
10^7
10^8
(0,0)
(0,1)
(1,0)
(1,1)
Observed State


Message Passing (MP) Outcomes: C++11 vs Java
C++11 (CppMem)
Java (JCStress)
Frequency (log scale)
10^2
10^3
10^4
10^5
10^6
10^7
10^8
(0,0)
(0,1)
(1,0)
(1,1)

5. Discussion

This project confirms that **release/acquire semantics** in Java and C++11 are largely equivalent in their guarantees:

1. **Store Buffering and Load Buffering** illustrate that **weak-memory behaviors** are possible in both languages, including stale reads and interleaved updates.
2. **Message Passing** demonstrates that **release/acquire enforces ordering and visibility**, ensuring proper synchronization.
3. Differences primarily appear in **execution frequency**, influenced by VM optimizations, scheduling, and stress modes; however, theoretical allowed/forbidden outcomes align perfectly.
4. This supports the SC-DRF guarantee: properly synchronized programs behave consistently, while weakly synchronized programs can exhibit anomalies allowed under relaxed semantics.

---

6. Conclusion

The study demonstrates a practical **cross-language comparison** of memory models:

- **C++11 and Java release/acquire semantics** allow identical weak-memory outcomes in litmus tests.
- **JCStress + VarHandle** is a reliable tool to empirically validate Java memory model behavior.
- **CppMem** provides a theoretical baseline for comparing weak-memory behaviors.
- The results highlight **where developers must be careful with weak-memory effects** and illustrate the importance of formal modeling for concurrent programs.

---

7. Files Attached

- C++ program called “StressTest” for running each litmus test 1 million times and printing the frequency distribution. (contains a README file for running the program)
- A Jcstress program for running the litmus test along with the result in index.html for each test. (contains a README file for running the program)